Buffer Feedback Scheduling: Runtime Adaptation of Ubicomp Applications

Christian Decker[1], Michael Beigl[2], Till Riedel[1], Albert Krohn[1], Tobias Zimmer[1]

[1]Telecooperation Office (TecO), University of Karlsruhe
[2]Distributed and Ubiquitous Computing (DUS), University of Braunschweig
{cdecker,riedel,krohn,zimmer}@teco.edu, beigl@ibr.cs.tu-bs.de

Abstract. In this paper we propose an operating system design for Ubicomp applications that are implemented on embedded sensor platforms. The OS provides support for both periodic sensor sampling and sequential appliation logic. Core component is a lightweight real-time runtime system guaranteeing predictable real-time behavior of periodic sampling processes. The design utilizes a novel method, called buffer feedback scheduling (BFS), to adapt the system under unpredictable workload. Processes are automatically coordinated and expensive hardware accesses are reduced when the feedback indicates that the results do not contribute to the application. Real-time behavior is guaranteed throughout the entire runtime. Theoretical analysis and implementation in a Ubicomp application study on the Particle Computer sensor platform demonstrate a significant performance step-up when utilizing BFS.

1 Introduction

Within Ubicomp, small, battery powered embedded sensor devices are a state-of-the art technology for detecting activity and situation information on an object or in the environment. Adding embedded sensor devices to things converts "dumb" and passive objects to smart and reactive subjects. A first example of such an object was the Mediacup[1], a coffee cup with embedded tiny sensor electronics. The cup was able to recognize conditions within the cup - e.g. if cup is full, coffee is cold etc. - and to react accordingly - e.g. reporting to the coffee machine the coffee consumption status. In general, embedded sensor systems are able to process raw sensor information to high-level situation information directly on the object and to trigger reactions. Software performing such processes must be able to handle

- periodic sampling of sensor information - e.g. the amount of liquid in a cup,
- processing of sensor information - e.g. to conclude on liquid status,
- reaction on events - e.g. to trigger further actions like start a new brew.

The quality of typical applications on embedded sensor systems is highly dependent on the correct recognition of situations. It requires the correct periodic

execution of the sensor sampling since those samples provide the information basis for the recognition. Fragmented sets of sensor information would otherwise require an additional resource intensive pre-processing for the applications. An OS supporting applications on embedded sensor devices must therefore be able to handle two general sets: Periodic processes and data-driven processes. This paper will introduce an OS concept that enables optimized coordination between the two process sets through feedback scheduling. It will contribute to the quality of the data processing and recognition algorithms and therefore provides an optimal basis for Ubicomp applications on embedded sensor devices.

The paper is organized as follows: In section 2, we analyze the behavior of Ubicomp applications on embedded sensor systems. Therefrom, we derive our system design in section 3. Section 4 and 5 introduce the formalization and theoretical model of feedback scheduling. The paper demonstrates the new system design in an implementation in section 6 and a case study in section 7 of the Remembrance Camera - a multi-sensor application for an embedded sensor platform. It follows a view on related work before the paper concludes in section 9.

2 Ubicomp Application Analysis

This section will briefly analyze typical Ubicomp applications based on experiences collected with several Ubicomp applications. It allows us to derive requirements for an OS approach supporting such applications. We found that applications like MediaCup, DigiClip[2], eSeal[3] and AwareOffice[4] incorporate continuous sensor perception of environmental conditions and activities - e.g. movements patterns - with data processing for context and in-situ recognition. Only periodically acquired sensor data and their processing enable accurate, detailed reaction in a timely fashion, especially in very mobile settings where an overarching processing back-end is not permanently available. For such applications, we conclude on the following common characteristics:

- the complete application runs on an embedded sensor device
- inputs are obtained from multiple, periodically sampled sensors
- data-driven processing implements context recognition and reaction

While periodic sensor samplings are under control of the developer, data-driven processing depends on acquired sensor input. An example is a rule-based expert system: The number of cycles to evaluate the rule base and each single rule's evaluation runtime depends on the facts coming in. The overall runtime behavior is not known at the design time and sensor samplings from unpredictable environmental conditions consequently cause a highly dynamic data processing behavior. An appropriate OS concept for Ubicomp applications has to separately support the different natures of periodic and data-driven processes. Runtime management of both is a primary design goal. Therefore, we derive the following requirements: Firstly, periodic sensor sampling is required to be guaranteed without any interference. This is usually referred to as a non-preemptive real-time scheduling. Interference or interruption while accessing hardware might

have fatal consequences, e.g. deadlocks, distorted sampling, and assertion violations. Secondly, since data processing is highly varying, coordination between periodic and data processing parts needs runtime support. Thirdly, the application's runtime behavior needs to be adapted to the current data-dependent computation effort through feedback into the scheduling process.

3 System Design

Based on the application analysis, we decompose a Ubicomp application in a periodic part - responsible for the access to the sensor hardware - and an application logic part. This separates the two concerns: periodic sensor data acquisition and data processing. At the lower layer of our system design (figure 3) services acquire

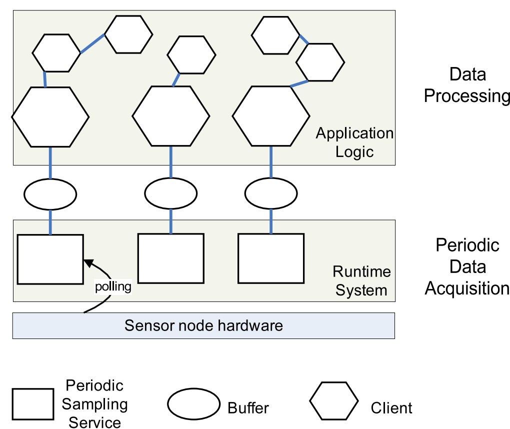

Fig. 1. System design of a Ubicomp application on a wireless sensor node

the sensor data. Services are executed periodically and encapsulate the access to the sensor hardware. Services are non-preemptive, i.e. once started a service runs to completion. A service cannot be called by another service or any other part of the application logic. Hence, a runtime system is responsible to drive services. Independent from application logic, the runtime system is required to strictly guarantee that services are executed according to a given period. This imposes real-time constraints on the services. The higher layer of the design comprises data-driven client functions that process input data from periodic services. A client is bound to one service via an intermediate buffer. Clients may form complex applications structures by calling other client functions that are not bound to a service. Clients are non-periodic, but run due to the availability of data. The buffers between the two layers decouple periodic processes from the data-driven application logic. Consequently, buffers form a data-based platform abstraction for the application.

Our system design yields the following advantages: The system clearly separates periodic processes from data driven processes. Independent, periodic real-time services guarantee a steady data acquisition process of the environment. Data-driven application processes can be seamlessly combined through buffers forming a data-based abstraction. Finally, the design eases the application development since periodic tasks can be delegated to services and let the developer focus on the actual application logic.

4 Formalizing the System

The runtime system should guarantee periodic execution of all services. As a consequence, real-time scheduling is applied. In this section will introduce terminology and formal background. Services are equivalent to non-preemptive real-time tasks. We use the term service, because we want to emphasize their role as mediators between the hardware platform and the application.

A periodic service s_i is described by tuple (C_i, T_i) (C_i: computation time, T_i: period). C_i is assumed to be the service's worst-case execution time (WCET) and known in advance. $D_i = T_i$ is the deadline of each service, i.e. a service has to run to completion before its next period begins. Services should not be preempted by other services. A service execution begins at a first arrival time $a_{i,0}$ and is repeated every period at $a_{i,n} = nT_i$. If no arrival time is given, the set of services S_p is said to be non-concrete. From every non-concrete set any concrete one, where services are associated with arrival times, can be generated.

Client functions f_j consume the data produced by the services. They poll the buffers and continue their execution due to the availability of data. Decoupling through buffers makes the clients independent from periodic services and allows preemption by services. The clients' computation time is unknown. Their minimum period is $T^f = \sum_j C_j^f$, where C_j^f is the jth client's unknown computation time.

Problem Formulation. From the highly dynamic computation times of clients two problem cases arise. The first case is data dropping. This situation appears in an overload situation where the clients' computation times become larger. As a consequence, a client cannot serve its buffer until the next service execution begins. The previously sampled data from the associated service is then not processed and consequently dropped. In figure 2 service s_i produces data for long running client f_j. The client is interrupted and finally misses to process its buffer content. The problem is avoided if the condition $\forall i, j : \sum_j C_j^f < T^f \leq T_i$ holds. Otherwise, the service with the minimum period will recur before its buffer content could be processed. The second problem case is client idling (figure 3). When clients' computation times decrease, they access the buffers several times until new values are provided by services. The idle case can be avoided for any client by fulfilling the following condition: $\forall i, j : T^f \geq T_i > \sum_j C_j^f$. Consequently we state the following problem:

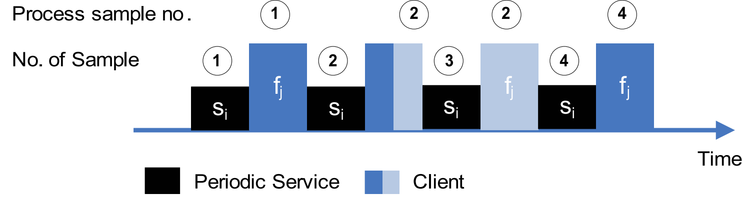

Fig. 2. Data drop: Sampling no.3 is not processed due to large computation time of f_j

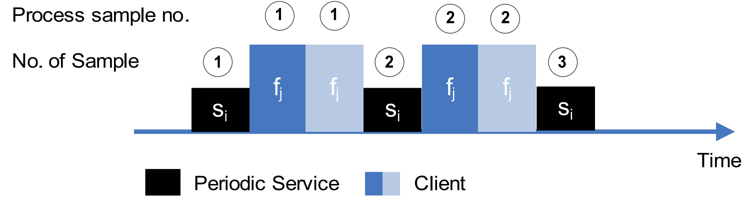

Fig. 3. Client idling:f_j processes samplings no.1&2 repetitively until a new value arrives

Problem 1. In unpredictable environments, where computation time C_j^f is unknown and highly dynamic, find periods T_i of the real-time services, that following conditions hold:

1. $\forall i, j : \min_i \{T_i\} = T^f > \sum_j C_j^f$ (period adaptation)
2. $\forall i : s_i$ finishes at latest at its deadline $D_i = T_i$ (preserve real-time)

The varying computation times of the clients require an adaptation of the services' periods. Our approach is to *expand* and *compress* a service period according to the change of the client computation time. An additional component - a controller - performs this adaptation during the system's runtime.

5 Buffer Feedback Scheduling

Data drop and client idling can be efficiently measured at the buffers between the periodic services and the clients. Each buffer is annotated by a single bit, which flips between 1 and 0 when the service and the client alternately access it. The bit remains in its current state, if a service or a client repeatively access the buffers in a non-alternating sequence. In this case, a counter is incremented for data drops and idling periods respectively. The counter values are fed back in the runtime system where a controller adapts the services' periods for the scheduler. Therefore, we call this adaptation method buffer feedback scheduling (BFS). The goal of BFS is to keep both counters at 0. The figure 4 depicts the basic principle of the BFS. The problem statement from section 4 requires, that the period adaptation preserves the real-time capability of the system. As a consequence, we firstly analyze the scheduling behavior under adaptation. Secondly, for the

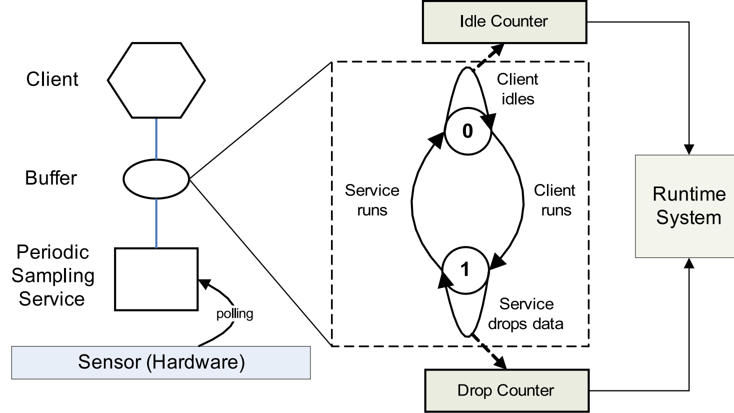

Fig. 4. Buffer drives 1-bit automaton. Feedback on idling and drops via counters.

feedback adaptation itself, we have to close the loop between the actual buffer performance and the service period and synthesize an appropriate controller for implementing the period adaptation.

5.1 Scheduling Analysis

Scheduling of real-time tasks is a well-investigated research topic. This section will extend this research by our approach of adapting service periods. We show, that this does not alter any statements about schedulability and the schedule. Consequently, period adaptation does not impose any scheduling overhead and is therefore well-suited for resource-constrained sensor node platforms.

The authors in [5] proved two conditions that are sufficient and necessary for scheduling a set of non-concrete, non-preemptable periodic tasks. Our considerations in section 4 show that this can be directly applied to our service approach. We will formulate all results using the term services instead of tasks. With the prerequisite that the set of periodic services $S_p = \{(C_1, T_1), (C_2, T_2), ..., (C_n, T_n)\}$ is sorted in non-decreasing order, i.e. $T_i \geq T_j$ if $i > j$, conditions from [5] are

(1) $\sum_i \frac{C_i}{T_i} \leq 1$

(2) $\forall i, 1 < i \leq n; \forall L, T_1 < L < T_i : L \geq C_i + \sum_{j=1}^{i-1} \left\lfloor \frac{L-1}{T_j} \right\rfloor C_j$

The condition (1) requires that the processor is not overloaded. The inequality in the condition (2) provides a least upper bound for processor demand, which can be realized in the interval L. As a consequence, we state the following corollary.

Corollary 1. *If the set of periodic services* $S_p = \{(C_1, T_1), \ldots, (C_n, T_n)\}$ *is schedulable according to condition (1) and (2), then the new set* $S_p^* = \{(C_1, T_1 + k), \ldots, (C_n, T_n + k)\}$*, where each period is increased by a constant time* k*, is schedulable according to condition (1) and (2).*

To prove the corollary, we first prove the following lemma.

Lemma 1. *If $g(k), h(k)$ are two linear functions with $\frac{\partial g}{\partial k} = \frac{\partial h}{\partial k}$ and $\forall k, k > 0$, then $\frac{g(k)}{h(k)}$ is monotonic.*

Proof. We prove this lemma constructively utilizing derivation $\frac{\partial}{\partial k}\frac{g(k)}{h(k)}$. Let $g(k) = k + x$ and $h(k) = k + y$, then $\frac{\partial}{\partial k}\frac{g(k)}{h(k)} = \frac{y-x}{(x+k)^2}$. For $x, y > 0$ and $x < y$, $\frac{\partial}{\partial k}\frac{g(k)}{h(k)} > 0$ and therefore $\frac{g(k)}{h(k)}$ is monotonically increasing. For $x, y > 0$ and $x > y$, $\frac{\partial}{\partial k}\frac{g(k)}{h(k)} < 0$ and therefore $\frac{g(k)}{h(k)}$ is monotonically decreasing. For $x, y > 0$ and $x = y$, $\frac{\partial}{\partial k}\frac{g(k)}{h(k)} = 0$ and therefore $\frac{g(k)}{h(k)}$ is simultanously monotonically increasing and decreasing.

Proof. First, we prove condition (1) of corollary 1:
From $T_i + k > T_i$ it follows directly that $\frac{C_i}{T_i} > \frac{C_i}{T_i+k} \forall k, k > 0$. As a result $\sum_i \frac{C_i}{T_i+k} < \sum_i \frac{C_i}{T_i} \leq 1$.
We now prove condition (2): Note, that k is also applied to L in this condition, so that the constraint changes to $\forall L, T_1 + k < L + k < T_i + k$. It is enough to focus on the expression $\left\lfloor \frac{L-1}{T_j} \right\rfloor$ for the following cases:

Case 1:$(L - 1 < T_j)$ Increasing the periods T_j by a constant k leads to following: $\forall k, k > 0 : \lim_{k\to\infty} \frac{L-1+k}{T_j+k} = 1^-$, i.e. for all k with $k > 0$, the expression $\frac{L-1+k}{T_j+k}$ converges to 1 from the left side. With lemma 1, it converges monotonically increasing. As a result, $\left\lfloor \frac{L-1+k}{T_j+k} \right\rfloor = 0$ for all $k > 0$.

Case 2:$(L - 1 > T_j)$ Increasing the periods T_j by a constant k leads to following: $\forall k, k > 0 : \lim_{k\to\infty} \frac{L-1+k}{T_j+k} = 1^+$, i.e. for all k with $k > 0$, the expression $\frac{L-1+k}{T_j+k}$ converges to 1 from the right side. With lemma 1, it converges monotonically decreasing. As a result, $\left\lfloor \frac{L-1+k}{T_j+k} \right\rfloor < \left\lfloor \frac{L-1}{T_j} \right\rfloor$ for all $k > 0$.

Case 3:$(L - 1 = T_j)$ Increasing the periods T_j by a constant k leads to $\left\lfloor \frac{L-1+k}{T_j+k} \right\rfloor = \left\lfloor \frac{L-1}{T_j} \right\rfloor$ for all $k > 0$.

As a result we find (properties on S_p, i, L as stated in condition (2)): $L \geq C_i + \sum_{j=1}^{i-1} \left\lfloor \frac{L-1}{T_j} \right\rfloor C_j \geq C_i + \sum_{j=1}^{i-1} \left\lfloor \frac{L-1+k}{T_j+k} \right\rfloor C_j, \forall k, k > 0$ meaning S_p^* is schedulable.

In the last step we have to prove that the schedule will not change after the transition from $S_p \to S_p^*$. According to Jeffay et al. in [5], the non-preemptive earliest deadline first (EDF) scheduling algorithm will schedule any concrete set generated from an unconcrete one. We apply this to the set S_p^* and state the following corollary:

Corollary 2. *If the unconcrete sets of periodic services S_p and S_p^* are schedulable according to condition (1) and (2) and the EDF algorithm has created a schedule $sched_{S_p}$ out of a concrete set generated from the unconcrete S_p one, then the same schedule is also valid for S_p^*, i.e. $sched_{S_p^*} = sched_{S_p}$.*

Proof. At the end of a service execution the global time is $t = a_{i,n-1} + C_i$. EDF selects then a service s_j with the closest deadline, i.e. s_j must fulfill the following condition: $\forall j, n : t < \min_{j,n} \{a_{j,n} + T_j\}$. Remember that we set $D_i = T_i$. The absolute deadline of the nth instance of s_j is $D'_{j,n} = a_{j,n} + T_j$ and therefore the EDF condition becomes to $t < \min_{j,n} \{D'_{j,n}\}$. In S^*_p, the absolute deadline of s^*_j is $D^{*'}_{j,n} = a_{j,n} + T_j + k$. Therefore, $t < \min_{j,n} \left\{D^{*'}_{j,n}\right\} = \min_{j,n} \{a_{j,n} + T_j + k\} = \min_{j,n} \{a_{j,n} + T_j\} + k$. Since $k = const.$, EDF selects the same service as it would do for S_p and we obtain $sched_{S^*_p} = sched_{S_p}$.

To conclude, we have proven that a period adaptation preserves the real-time behavior of the system. It guarantees that condition (2) of the problem formulation holds. It even does not impose any overhead because corollary 2 proves that adaptation will not alter the schedule. This is the fundament for controller synthesis in the following section.

5.2 Controller Synthesis

The controller is responsible for the services' periods adaptation. The adaptation is required to hold $\min_i \{T_i\} = T^f$ from condition (1) of the problem formulation. Figure 5 shows the placement of the controller within the runtime system. The

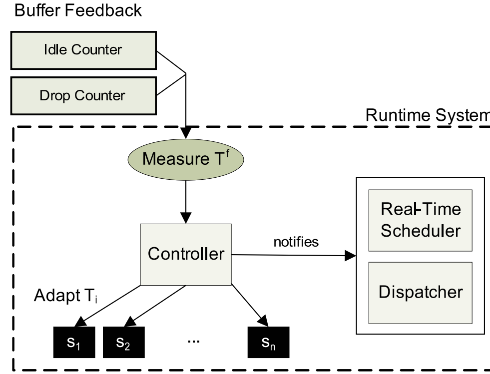

Fig. 5. Feedback controller within the runtime system

controller is triggered by non-zero drop and idle counters from the buffers. It then starts to measure the actual client period T^f and adapts the services' periods according to condition (1). Afterwards, it notifies the dispatcher for computing the new arrival times of the services before the real-time scheduler enqueues them for the next execution. In detail, the controller will compute the new period T'_i as follows:

$$T'_i = T_i + k \text{ with } k = T^f - \min_i \{T_i\} \text{ (expansion, drop counter} > 0) \quad (1)$$

$$T_i' = \max\{T_{min}, T_i - k\} \text{ with } k = \min_i \{T_i\} - T^f \text{ (compr., idle counter > 0)} \quad (2)$$

The constant T_{min} marks the lower bound of the period for that the scheduling analysis guarantees that all services will hold their deadlines. In case of compression, the periods should never set below this bound.

5.3 Controlled Buffer Feedback Scheduling

In this section we will compare the controlled BFS system with the non-controlled system under variable client workloads. The results were achieved through simulations using the Ptolemy II framework[1]. We vary the clients' computation times by applying step loads - a sudden change of the computation time. The behavior is investigated for both step-up and step-down loads in both the controlled and non-controlled case. Results are compared using the following definition of the accumulated data drop ratio: $DropRatio(t) = \frac{\sum_t dropCounter(t)}{\sum_t serviceExecutions(t)}$. We define the cumulative idle ratio as the time spent for idling in relation to the runtime of the system: $IdleRatio(t) = \frac{\sum_t idleCounter(t) \cdot clientTime(t)}{t}$. In both figures 6 and 7

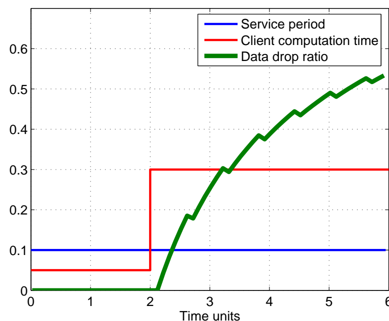

Fig. 6. Non-controlled step-up behavior. Service period remains constant and the drop ratio increases.

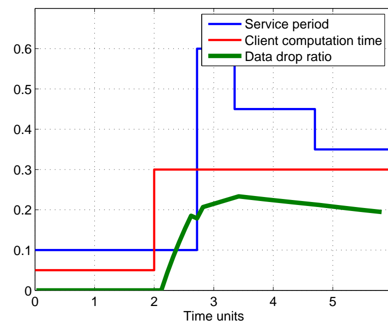

Fig. 7. BFS-controlled step-up behavior. Service period is expanded and the data drop decreases.

the step load occurs at time 2 and changes from 0.05 to 0.3. The services' periods are initially set to 0.1. The system was previously analyzed according to the conditions in section 5.1 in order to ensure that services will hold their deadlines. When the step occurs, the drop ratio increases shortly afterwards. In the non-controlled case, the service period remains at 0.1, data drop occurs causing an increasing drop ratio. In this example, the services run 6 times until one date is processed by the clients. As a result, the drop ratio will asymptotically approach 0.83. On the other side, the controlled BFS system adapts to the new situation

[1] http://ptolemy.berkeley.edu/ptolemyII/

and expands the services period. It reaches the final period of 0.35 at 4.7 time units. The reaction to the step is delayed because the measurement of T^f firstly starts after the first data drop is detected. Since the step occurs at an arbitrary point in time, the T^f-measurement is biased due to the overlap of the original and the increased clients' computation times. This causes the overshot. Maximum drop ratio in this example is 0.24 and is reached at time 3.4 and decreased afterwards. For comparison: At the same time the non-controlled system had a drop ratio of 0.31. Although, there is no data drop anymore, the drop ratio is still positive, but approaching 0. This is due to the drop ratio definition because it represents the accumulated drop ratio throughout the entire runtime. The

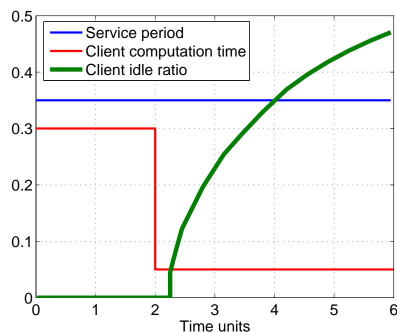

Fig. 8. Non-controlled step-down behavior. Service period remains constant and idle ratio increases.

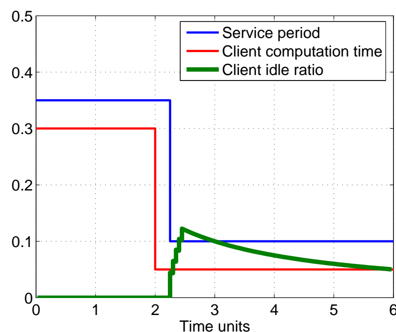

Fig. 9. BFS-controlled step-down behavior. Service period is compressed and idle ratio decreases.

figures 8 and 9 depict the client idling in the non-controlled and controlled case. The step-down from 0.3 to 0.05 is applied at time 2. In the non-controlled case, the idle ratio increases shortly after the step and remains increasing. According to the chosen parameters, the system executes 6 times the clients before a new value is produced by the services. In this example, the idle ratio will asymptotically reach 0.71. In the controlled BFS system idling is handled by compression of services' periods. There is no undershot of the period because services should keep a minimum period of 0.1 to be still real-time schedulable (see equation (2)). Although the BFS controller denotes idling very early, reaction is delayed because the services were already scheduled and had to wait until their next period. Meanwhile, the idle ratio grows up to 0.12. The system has then adapted and the idle ratio decreases. However, it is still positive because it represents the cumulative idling ratio.

6 Implementation

We implemented the real-time runtime system and the BFS controller on the Particle Sensor platform [6]. The implementation was carried out using the Small

Devices C Compiler (SDCC) for the PIC18 platform. Services contain parameters like their period, arrival time, states (ready, waiting), pointers to their output buffers and service functions. Latter actually implement the service functionality. The runtime system manages all services in two queues for waiting and ready services. A service is waiting, if it has not yet reached its arrival time. Once it has reached it, the scheduler enqueues it in the ready queue according to the EDF policy. The dispatcher brings a service to its execution by calling the service function of the first service in the ready queue. When no service is ready, the control is given to the clients for data processing. The buffers manage the access automaton and provide feedback for the BFS controller which is triggered as soon as dropping or idling is detected. Dispatcher, scheduler and BFS controller all work on a 32bit time format of the Particle's real-time clock. The accuracy is limited by the 32kHz clock to about 31 microseconds. The tables 1 and 2 list the memory footprint and the computation effort of the runtime system.

Component	ROM	RAM
Scheduler	5.556 kB (4.2%)	0.161 kB (4,0%)
Dispatcher	3.600 kB (2.7%)	0.052 kB (1.3%)
Queue	2.498 kB (1.9%)	0.022 kB (0.6%)
Controller	1.870 kB (1.4%)	0.021 kB (0.5%)
Byte-Buffer		34 bit
Service		104 bit
Sum	13.524 kB (10,5%)	0,256 kB (6.4%) excl. services and buffers

Table 1. Memory footprint of the runtime system

Function	Cycles
Dispatcher	
insert_waiting, 5 services	1250
getTime	670
timeGreater	140
Scheduler	
schedule_EDF, 4 services	1903
compareDeadlines	421
BFS Controller	
Compression, 5 Services	7371
Expansion, 5 Services	6481

Table 2. Computation effort in cycles on the PIC18f6720

7 Case Study: Remembrance Camera

The Remembrance Camera can be considered as a typical example for an embedded sensor system in Ubicomp. The Remembrance Camera is a camera-based device that is worn by a person throughout the day as a personal wearable device and automatically takes pictures of interesting events experienced from that person. The camera consists of two components: An embedded sensor board for recognition of activities and for data processing and a miniature digital camera to record pictures under control of the sensor board. We used the Particle Computer platform as a typical candidate of an embedded sensor system with ultra-low power consumption for continuous monitoring of activities through sensors. Low power design was required because of usability since the camera should be of a very small outline and able to operate throughout the day without

battery recharging. The sensor hardware is attached to a small digital camera, the Apitek PenCam, and can be worn in a shirt's pocket or on a necklet (total size about 4.5x4x15 cm, see figure 10). The aim of the Remembrance Camera

Fig. 10. Remembrance Camera (with Particle sensor node)

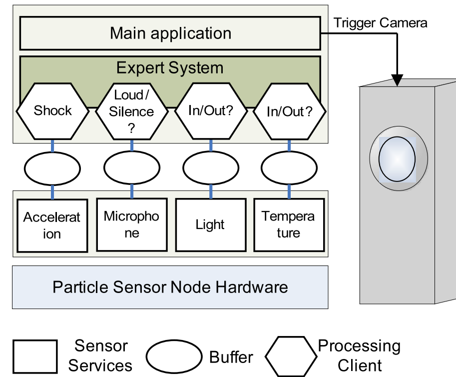

Fig. 11. Remembrance Camera recognition architecture

system is to enable a person to recall all important events of a day using taken pictures. Rather than having a continuous recording of all pictures of events - which would require a user to scan through a lot of pictures - we liked to present only the most important pictures of the day to the user based on a recognition of important events of a day. This recognition was performed stand-alone by the camera device using sensors and activity recognition methods. At synchronization points pictures and recognition information could be uploaded and viewed on a PC. Events are detected through sensor recognition methods using embedded sensor hardware, sensor detection firmware and higher-level activity recognition and fusion methods (see figure 11). Acceleration, light, microphone and temperature sensors are used as inputs. These sensors are connected to *sensor services* performing the periodic sampling of sensor data. Sensors are used as input to the data *processing clients* performing the recognition algorithms. The recognition in each of the clients uses a rule-based expert system[7]. It operates on a ring-buffer of time-stamped sensor data acquired by sensor services and performs various operations under varying computation times depending on sensor input. As a consequence, the periods of clients are different due to variations of the sensor data. Output of the clients, listed in table 3, is finally used by the main application for recognizing the person's situation (table 4). The Remembrance Camera main application uses a change in the situation to detect an important event, which then triggers the camera to take a picture. The complete system is written in C and embedded on the Particle sensor node (CPU:PIC18F6720@20MHz, 4kRAM, 128kFlash).

Sensor service	Client output (Activity)
acceleration	shock, calm/excited
light	inside/outside a building
temperature	inside/outside a building
microphone	loud, talking, silence

Table 3. Sensors on the Particle platform and extracted information

Situation	Inputs Activities
working	{calm, inside, silence}
meeting	{inside, talking}
running	{shock, outside}
standing (outside)	{outside, calm}
work pauses	{shock, inside}
work interruptions	{excited, inside, talking or loud}

Table 4. Situations and activities used by the main application for taking a picture

Scheduling performance using the BFS controller. The Remembrance Camera application is sensitive to continuous and regular sensor detection. Missing information or large delays within the recognition affects the application and may lead to misinterpretation of the situation. Handling such exceptions requires more time and memory intensive algorithms that would have exceeded the platform's performance. Therefore, it is important for the quality, that the recognition runs on a frequent basis and that (almost) no samplings are dropped. This is achieved by the BFS controller. In figure 12 we have included a scheduling

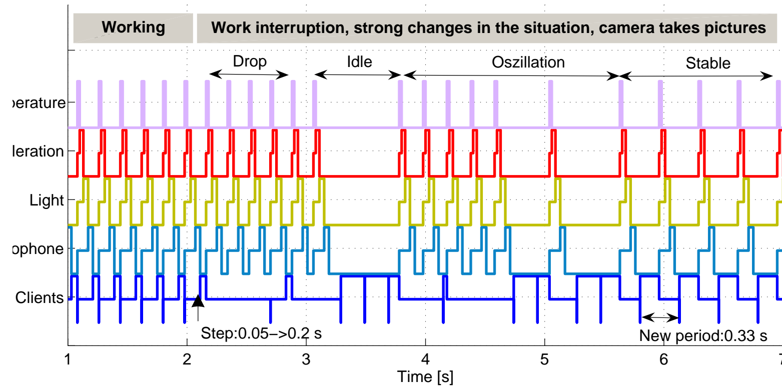

Fig. 12. Scheduling trace of BFS controlled services when situation changes from *working* to *work interruption*

trace for the detection of a situation change from *working* to *work interruption*. The situation recognition is handled by the processing clients while the sensor input is provided by the sensor services. All services run every 0.18 seconds (s) - the initial service period. The new situation *work interruption* arising at time 2s

leads to a higher effort for recognition. For this reason, the clients' computation time increases from 0.05s to 0.2s and therewith the clients' period. As a result, services run faster than data processing, data drop occurs and the system starts to adapt the period of the services. After 3.7s the BFS controller stabilizes the new period at 0.33s. Figure 13 depicts the progression of the period adaptation

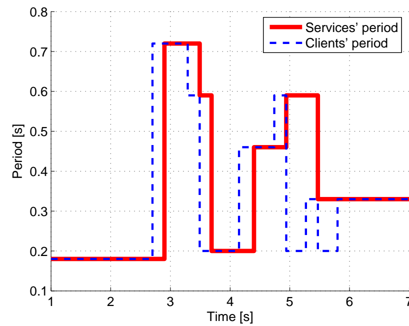

Fig. 13. BFS controlled services' and clients' periods during the adaptation. After adaptation the situation recognition (clients) runs with period 0.33s.

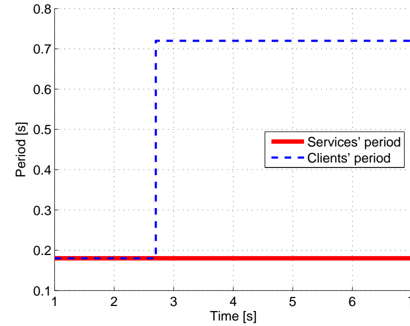

Fig. 14. Non-controlled services' and clients' periods. The situation recognition (clients) runs with period 0.72s more than twice as slow as in the controlled case.

in more detail. When the clients' period change, the services' period follows until both reach their final period of 0.33s. During the adaptation the periods overshoot the correct ones. This is compensated by the reverse reaction. As a result, the periods are temporarily unstable causing an oszillation until the final value has been stabilized. Nevertheless, the scheduling analysis in section 5.1 shows that the real-time contraint is never violated. Figure 14 shows both periods when the controller is disabled: Clients, which are responsible for the Remembrance Camera's situation recognition, are called more seldom - every 0.72s. As a result, we find that when utilizing the BFS controller the clients are executed more than twice as often as in the non-controlled case. For the Remembrance Camera application, the BFS controller yields a significant performance step-up.

User study. The overall quality of the Remembrance Camera application is highly dependent on underlying technical constraints, especially the expert systems that itself is very dependent on quality of the scheduling and operation of the sensor services. We evaluated the overall quality in a small initial user study and will report a short excerpt of this study. In the study 7 persons from 20 to 50 years, all of them with only minor knowledge of computer systems, were asked to carry the Remembrance camera during the daytime. Candidates were interviewed 3 days after performing the test on the value of the system. Among the questions there were 1) if they think the system performs as expected, es-

pecially if it draws a good picture of the day and if the collection of pictures contains unnecessary information and 2) if they found the information useful. All candidates found that the system draws a good picture of the day while almost half of them mentioned that there are unnecessary pictures taken while the day. All users agreed that the information is useful and most of them would wish to use the system on a daily basis. Although overall this is a very good result for the Remembrance Camera application, we were curious to see why half of them found unnecessary information among the pictures. From an additional analysis we found that in these cases the person was in a home environment all the day and not in an office environment like the other users. A more in depth analysis revealed that our initial expert system was not trained to this environment and therefore performs non-optimal. We therefore assume, that the underlying sensor service and scheduling system performed optimal like in the cases with persons in the office environment and that our service scheduling approach provides a good basis for activity recognition systems like the Remembrance Camera.

8 Related Work

Besides of our work other scheduling approaches are already implemented on embedded sensor systems. OSs like TinyOS[8] and SOS[9] incorporate non-preemptive FIFO schedulers, where tasks are sorted in order of their calling by the application logic. The system is kept responsive to various inputs by deferring procedure calls that are invoked through periodic processes. Several OSs for microcontrollers like FreeRTOS[10] and XMK[11] guarantee preemptive real-time scheduling. However, none of these approaches copes with the dynamic nature of data processing on sensor devices. WCETs need to be specified in advance leading to a static runtime behavior for dynamic workloads. Furthermore, decomposing the application in periodic-only tasks is generally not suited for data-driven processing. Our approach of separation of periodic and data-driven processes preserves the sequential semantic of the application. Varying processing loads are handled dynamically by feedback scheduling. Stankovic et al. investigated feedback scheduling for real-time OSs[12]. The model was completely periodic and adapted the computation time of the tasks according to service levels. In contrast, Buttazzos elastic task model [13] is closer to our approach since it compresses and expands tasks periods in order to handle over- and underutilization of the system. Apart from purely periodic modelling, the system is limited to preemptive tasks. With buffer feedback scheduling (BFS) we have extended this feedback control research on data driven processes with no a-priori knowledge on the computation times.

9 Conclusion and Outlook

In this paper we presented a new OS concept for supporting Ubicomp applications on embedded sensor systems utilizing adaptation through feedback scheduling. Adaptation can accurately modify parameters, e.g. service periods, in order

to automatically coordinate processes and achieve a better performance under uncertain or unknown conditions. Our approach - buffer feedback scheduling (BFS) - modifies service periods and especially addresses dynamic and unpredictable workloads caused by the data processing parts of an application. BFS achieved a significant performance step-up for an Ubicomp applicaion allowing a two times faster recogition rate. BFS automatically coordinates periodic and data-driven processes based on runtime information and does not rely on any a-priori specification of the coordination behavior.

Future work includes research on new controllers incorporated in the runtime system. Feedforward controllers are an interesting option, because they act preventively before the system runs in a data drop or idling situation. The system would change from reactive to proactive enabling a distributed coordination between diverse applications on embedded sensor systems within a network.

Acknowledgments

The work presented in this paper was partially funded by the EC through the project CoBIs (contract no. 4270) and by the Ministry of Economic Affairs of the Netherlands through the project Smart Surroundings (contract no. 03060).

References

1. Beigl, M., Gellersen, H.W., Schmidt, A.: Mediacups: experience with design and use of computer-augmented everyday artefacts. Computer Networks **35**(4) (2001)
2. Decker, C., Beigl, M., Eames, A., Kubach, U.: Digiclip: Activating physical documents. In: 4th IEEE IWSAWC. (2004) 388–393
3. Decker, C., Beigl, M., Krohn, A., Robinson, P., Kubach, U.: eseal - a system for enhanced electronic assertion of authenticity and integrity. In: Pervasive. (2004)
4. Zimmer, T., Beigl, M.: AwareOffice: Integrating Modular Context-Aware Applications. In: 6th IEEE IWSAWC. (2006)
5. Jeffay, K., Stanat, D.F., Martel, C.U.: On non-preemptive scheduling of periodic and sporadic tasks. In: Proceedings of 12th IEEE RTSS'91. (1991) 129–139
6. Decker, C., Krohn, A., Beigl, M., Zimmer, T.: The particle computer system. In: ACM/IEEE Information Processing in Sensor Networks (IPSN). (2005) 443 – 448
7. Fischer, M., Kroehl, M.: Remembrance camera. Term Thesis (2006)
8. Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D.E., Pister, K.S.J.: System architecture directions for networked sensors. In: ASPLOS. (2000)
9. Han, C.C., Kumar, R., Shea, R., Kohler, E., Srivastava, M.: A dynamic operating system for sensor nodes. In: Proceedings of MobiSys '05, New York, NY, USA, ACM Press (2005) 163–176
10. Barry, R.: FreeRTOS - a free RTOS for small embedded real time systems. http://www.freertos.org/ (2006)
11. Shift-Right Technologies: eXtreme Minimal Kernel (xmk) - a free real time operating system for microcontrollers. http://www.shift-right.com/xmk/ (2006)
12. Stankovic, J.A., He, T., Abdelzaher, T., Marley, M., Tao, G., Son, S., Lu, C.: Feedback control scheduling in distributed real-time systems. In: Proceedings of RTSS'01. (2001)
13. Buttazzo, G.C., Lipari, G., Abeni, L.: Elastic task model for adaptive rate control. In: Proceedings of IEEE RTSS '98, Washington, DC, USA, IEEE Computer (1998)