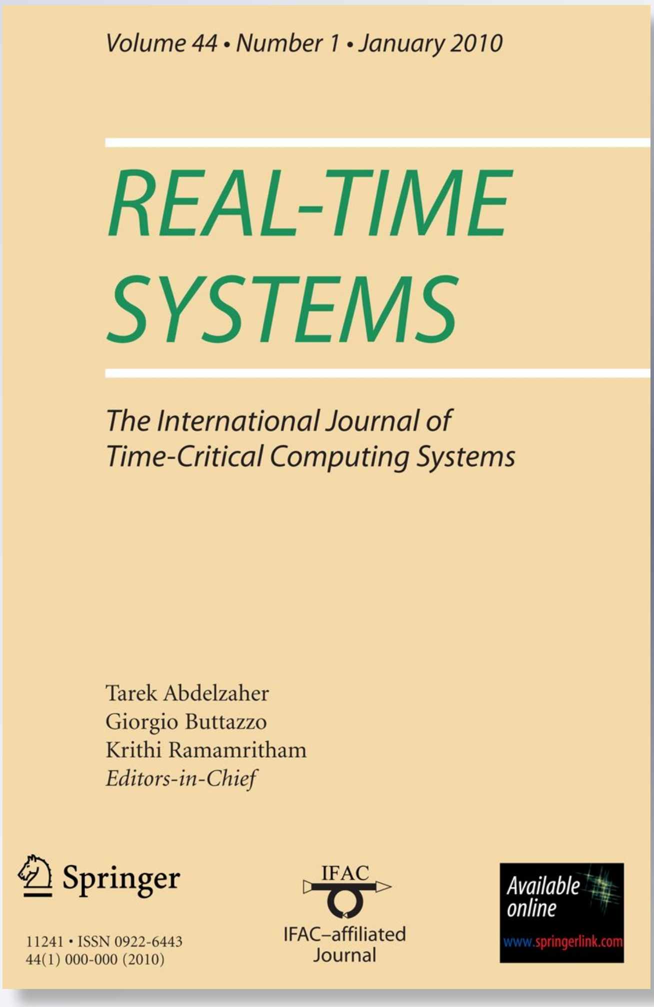
Energy-optimal Batching periods for asynchronous multistage data processing on sensor nodes: foundations and an mPlatform case study

Dong Wang, Tarek Abdelzaher, Bodhi Priyantha, Jie Liu & Feng Zhao

Real-Time Systems
The International Journal of Time-Critical Computing Systems

ISSN 0922-6443
Volume 48
Number 2

Real-Time Syst (2012) 48:135-165
DOI 10.1007/s11241-011-9138-5



Real-Time Syst (2012) 48:135–165
DOI 10.1007/s11241-011-9138-5

Energy-optimal Batching periods for asynchronous multistage data processing on sensor nodes: foundations and an mPlatform case study

Dong Wang · Tarek Abdelzaher · Bodhi Priyantha · Jie Liu · Feng Zhao

Published online: 5 October 2011

Abstract This paper derives energy-optimal batching periods for asynchronous multistage data processing on sensor nodes in the sense of minimizing energy consumption while meeting end-to-end deadlines. Batching the processing of (sensor) data maximizes processor sleep periods, hence minimizing the wakeup frequency and the corresponding overhead. The algorithm is evaluated on mPlatform, a next-generation heterogeneous sensor node platform equipped with both a low-end microcontroller (MSP430) and a higher-end embedded systems processor (ARM). Experimental results show that the total energy consumption of mPlatform, when processing data flows at their optimal batching periods, is up to 35% lower than that for uniform period assignment. Moreover, processing data at the appropriate processor can use as much as 80% less energy than running the same task set on the ARM alone and 25% less energy than running the task set on the MSP430 alone.

Keywords Real-time systems · Energy optimization · Sensor network · Heterogeneous platform

1 Introduction

In this paper, an optimal batching algorithm is proposed for asynchronous multistage data-processing on sensor nodes, where optimality refers to minimizing energy consumption subject to deadline constraints. Sensor data processing may include outlier

D. Wang (✉) · T. Abdelzaher
Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA
e-mail: dwang24@illinois.edu

B. Priyantha · J. Liu · F. Zhao
Networked Embedded Computing, Microsoft Research, Redmond, WA 98052, USA

detection, filtering, statistical analysis, correlation, spectrum analysis, CRC computation, and encryption. These operations can be grouped into computational stages. Each stage has a constant amount of state to keep, leading to a constant, data-size-independent overhead, in addition to a processing time that depends on the amount of data to process. By operating on data batches (as opposed to on individual data items), the algorithm maximizes processor sleep durations in between processing bursts, hence minimizing data-size-independent overhead, and maximizing energy-efficiency.

A trivially optimal batching algorithm is to wait for an amount of time equal to the data processing deadline less the time it takes to process one data batch. The accumulated batch is then processed all together. In this design, each processing stage waits until the previous stage has finished the batch. Each stage is triggered immediately by the completion of the predecessor stage(s).

This paper explores an alternative application design, where appropriately-sized data processing stages run *asynchronously* as independent periodic tasks, reading data from input buffers when they wake up and depositing into output buffers before going back to sleep. Admittedly, this design consumes more energy than the one above, because stages are decoupled by data buffering, essentially breaking up one big input buffer into many smaller interstage buffers. Invocation rates of individual stages are correspondingly increased to keep the smaller buffers from overflowing. However, this design is motivated by simplicity. For example, (i) it is lock free as no synchronization is needed among stages, (ii) it allows separating complex computation into small "independent" components, and (iii) it leads to fewer bugs since simplicity of design contributes to a more reliable implementation. In a data processing graph where individual stages run independently, the question of assigning periods to different stages becomes important. This question is akin to breaking up the end-to-end data processing deadline among stages in a way that maximizes energy savings while maintaining independence (Cao et al. 2010). The optimal period assignment algorithm described in this paper solves the above problem.

If more than one processor is present, a related question is where to run each data processing stage. Although higher-end processors consume more power when active, some are disproportionately faster than their lower-power counterparts. This means that they consume less energy per byte, as the higher power is consumed for a much shorter period, leading to a smaller energy product. A problem is the data-size-independent overhead, which is often also processor-speed-independent (e.g., wakeup cost and saving data to flash). Given enough batching, a break-even point is reached where the increased energy-efficiency in processing the batch outweighs the larger data-size-independent overhead. Task to processor assignment therefore depends on whether or not the optimal batching period is larger than the breakeven point.

We implement our optimal batching algorithm on mPlatform Lymberopoulos et al. (2007), a heterogeneous sensor node platform consisting of one higher-end processor (ARM) and one lower-end microcontroller (MSP430). It represents a next generation of sensor node platforms, evolving from earlier platforms that used to include a low-end processor alone. The ARM, on mPlatform, is more power-consuming and has a higher idle power and startup overhead. However, it is also more energy-efficient when utilized continuously. We show that up to 25% savings can be achieved in energy consumption when using our optimal batching algorithm compared to the case when everything runs on the lower-end microcontroller alone. The mPlatform is thus not only suitable for high-end sensor network applications, where the ARM helps with large computational requirements, but also suitable for very low-power applications, where traditional low-power platforms are unable to deliver the right energy-efficiency.

The remainder of the paper is organized as follows. Section 2 describes related work. Section 3 presents the task model and problem formulation. Section 4 describes optimal batching algorithm. Evaluation results from empirical measurements are presented in Sect. 6. Finally, the paper concludes with Sect. 7.

2 Related work

There exists extensive research on system-level power optimization for embedded and real-time systems. Earlier studies are limited to single processor systems, and using frequency and voltage scaling as power control "knobs" under application performance or time constraints (Shin et al. 2000; Krishna and Lee 2000; Acquaviva et al. 2001; Zhong and Jha 2004). Wakeup energy and time delay cost can be substantial in duty-cycled embedded devices. Benini et al. (2002) highlighted the notion of *break-even time* to accommodate the nontrivial energy cost of waking up a processor from sleep. Multiprocessor and distributed real-time scheduling are significantly more complicated than single processor cases. One needs to consider both task to processor assignment and processor state control (Khemka and Shyamasundar 1997; Luo 2000). In general, the problem of minimizing energy consumption of dependent tasks under hard real-time constraints is NP hard for heterogeneous multiprocessors. Baruah (2004) considers the task allocation problem on heterogeneous multiprocessor platforms without task precedence constraints nor hardware configurability. In Jin et al. (2005), Sivanthi and Killat (2004), and Zheng et al. (2005) the objectives are to maximize, respectively, the throughput, the minimal task slack, and task extensibility. An integer linear programming formalism has been proposed in Goraczko et al. (2008) to compute the schedule.

Dataflow programming models have long been used in signal processing and control applications (Lee and Messerschmitt 1987; Benveniste et al. 2003). Recently, static and dynamic dataflow models have been proposed to program sensor networks (Chu et al. 2007; Girod et al. 2006; Whitehouse et al. 2006), since they match well with the data streaming abstraction of the application domain. Typical dataflow scheduling optimize for throughput (Ha and Lee 1997; Chao and Sha 1997), dynamic memory usage (Buck 1993), or code size (Bhattacharyya et al. 1999). Our task model is also inspired by time-triggered architectures and languages, such as Henzinger et al. (2001). In these models, tasks are woken up periodically to process their inputs and produce their outputs. However, unlike Giotto, which uses a single buffer and allows newly generated data to override older, unconsumed data, our model keeps a traditional FIFO queue model for communication between tasks. This matches the application requirements for most sensor networks, where each collected piece of data needs to be processed.

Energy optimal dataflow scheduling has been explored in the context of buffer management. The most relevant work is on static or dynamic buffer insertion for multi-media application (Lu et al. 2002; Cai and Lu 2004). The idea is that by buffering the inputs, one can scale down the processor to a lower power state, since buffer insertion is not computationally intensive. Our work is different in that we take advantage of power diversity in heterogeneous processors and address the wake up energy cost.

Essentially the same idea of leveraging batching to amortize the energy overhead of high power components has been explored in the context of heterogeneous radio systems (Lymberopoulos et al. 2008; Sengul et al. 2008). Large data packets are stored and sent in bulk to take advantage of the energy efficiency of the high power radio, while short control packets are transmitted over the low power radio to ensure fast delivery. However, the work in this paper applies the idea to a dual processor platform under real time constraints, and thus formalizes the energy optimization problem from the perspective of allocating proper batching periods to tasks on heterogeneous processor boards.

The well-known concept of batching has been used in many situations to improve system performance (Pavlovski and Boyd 1999; Youn et al. 2008). In the context of sensor network, batching has mostly been done at MAC layer to study the traffic effect on network energy consumption (Ning and Cassandras 2007). MAC protocols batch multiple packets to share a single preamble or common header and send batched packets in a large chunk. This can avoid retransmission of the same header and reduce the number of times to access the shared medium in a competitive way. Our work introduces the batching idea to a heterogeneous sensor node platform and exploits batching to amortize the wake up overhead of high power processors.

There is also increasing interest in the research of reducing the energy of PC by adding a secondary low power processor (Agarwal et al. 2009; Shih et al. 2002; Sorber et al. 2005). The basic idea is to use the secondary processor to impersonate the CPU of PC to fulfill some simple tasks while putting PC into sleep mode for great energy savings. On the contrary, our approach uses two processors alternately in a smart way to achieve energy optimization.

3 Model and problem statement

In this section, we formulate the problem of finding the optimal batching period for each data processing stage, given a particular task-to-processor allocation, such that energy savings are maximized subject to deadline constraints. Later, we discuss how to compute the task-to-processor allocation. In practice, allocation is determined by the nature of the task. For example, the MSP430 is more energy-efficient than the ARM at simple mathematical and logical instructions, whereas the ARM is more energy-efficient at complex floating-point operations. In some cases, it also depends on period.

A point of departure in this paper from most prior schedulability literature lies in its use of a *data-centric* task model, where data are a first-order abstraction, and where the amount of data determines the amount of computation.

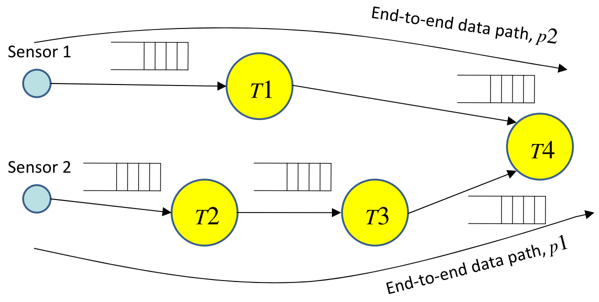


Fig. 1 An example of two paths

Consider a sensor network node with multiple sensors. Each sensor generates data at a given rate, creating a data flow. Data flows stream through multiple stages of processing on the node, each performed by its own periodic task. Thus, the topology mentioned in the paper is the topology of multistage processing tasks in a single node. Following a periodic task model, tasks will execute once somewhere within each period (as opposed to being executed exactly on period boundaries). Formally, we define the notion of a *path*, p, as an ordered set of periodic tasks that process the data stream sequentially in the order these tasks appear on the path. We say that a pair of consecutive tasks T_j and T_i on the path share a (directional) *link* $T_j \rightarrow T_i$, and that T_j is T_i's immediate predecessor. For example, Fig. 1 shows a task set with two paths, $p_1 = (T_2, T_3, T_4)$ and $p_2 = (T_1, T_4)$. Tasks T_3 and T_4 are an example of a pair of tasks that share a link ($T_3 \rightarrow T_4$), where T_3 is the immediate predecessor of T_4. Let R_{ji} denote the average rate of data transfer across the link $T_j \rightarrow T_i$. Data are transferred asynchronously. The producer deposits data into a shared buffer. The consumer then reads from that buffer at a later time.

Let P_i denote the period of task T_i. We call it the *batching period* to emphasize the fact that this period does not stem from physical requirements such as control loop stability or sampling rate. It is simply the period chosen over which data are buffered before they are processed in batch by T_i. When task T_i is invoked, it reads all the data from each of its input buffers, processes the data, and deposits results into its output buffer(s). The amount of data read by task T_i every period is thus equal, on average, to the sum $\sum_j P_i R_{ji}$, carried over the set of its immediate predecessors, $j \in Pred_i$.

After processing all the data in its queues, task T_i stops and waits until the next period. The average computation time of task T_i, on processor k, denoted by C_i^k, is the sum of a fixed, data-independent component, $c_{i,0}^k$ (e.g., wakeup cost and saving state to flash), and a component that grows linearly with data size. In other words:

$$C_i^k = c_{i,0}^k + \sum_{j \in Pred_i} c_{ji}^k R_{ji} P_i \tag{1}$$

where c_{ji}^k is a constant that reflects the time it takes to communicate, read and process each unit of data that accumulated from predecessor T_j. Observe that since rates, R_{ji} are fixed, the above equation can be rewritten as:

$$C_i^k = c_{i,0}^k + c_i^k P_i \tag{2}$$

where c_i^k is a constant. In theory, one might be tempted to add other terms to (2), reflecting algorithms of nonlinear complexity. Most algorithms that operate on data streams, however, use incremental forms that operate on fixed-size updates (e.g., one sample or one window of data at a time). They have the same complexity per update. Hence, while the computation might have arbitrary complexity in other parameters, it is linear in the number of updates processed, and hence linear in the input data size or the batching period.

If the energy it takes to execute $c_{i,0}^k$ and c_i^k is a_i^k and b_i^k, respectively, the total amount of energy E_i^k needed, *on average*, each time task T_i runs on processor k, is:

$$E_i^k = a_i^k + b_i^k P_i \tag{3}$$

Observe that b_i^k may include the cost of communicating data from the other processor, if the respective stages are not allocated to the same one. Assuming that the processors sleep when not executing any tasks and that the sleep energy is negligible,[1] the average power W_i^k consumed by processor k on executing task T_i is:

$$W_i^k = \frac{a_i^k}{P_i} + b_i^k \tag{4}$$

Note that a_i^k and b_i^k are two processor dependent parameters in our model, where a_i^k denotes the data-independent energy cost of the processor k (e.g., processor wakeup cost and cost of saving states to flash) and b_i^k is the data-dependent average power consumption of the processor k (e.g., data computation and communication cost). Some higher-end processor (e.g., ARM) has a higher a_i^k value, but lower b_i^k value for some tasks (e.g., ARM is more energy efficient at complex operations and long data types) due to its *disproportionately* faster speed (Lymberopoulos et al. 2007). However, the energy savings of using the higher-end processor is only possible when enough amount of data has been accumulated and processed in batch to offset its higher a_i^k overhead. Meanwhile, data usually have to traverse the processing path within an end-to-end deadline in order to meet the application responsiveness requirements or node buffer constraints. Therefore, we formulate the problem with the goal to find the optimal batching period for each task on the data path that minimizes the total (average) power consumption of the sensor node while respecting the end-to-end deadline of the data processing on the node.

Given each task, T_i, $1 \le i \le n$, executing on processor k_i, and the paths p_l, $1 \le l \le m$, defined on those tasks, it is desired to find the optimal batching period P_i, for each task, T_i, to minimize W, the total (average) power consumption:

$$W = \sum_{1 \le i \le n} \left(\frac{a_i^{k_i}}{P_i} + b_i^{k_i} \right) \tag{5}$$

subject to end-to-end time constraints on data paths. Data on path p must traverse the path from sensor to final output (e.g., the radio buffer to send data to the destination)

[1]It is trivial to extend this assumption to the case where sleep energy is significant but, in practice, it is usually negligible.

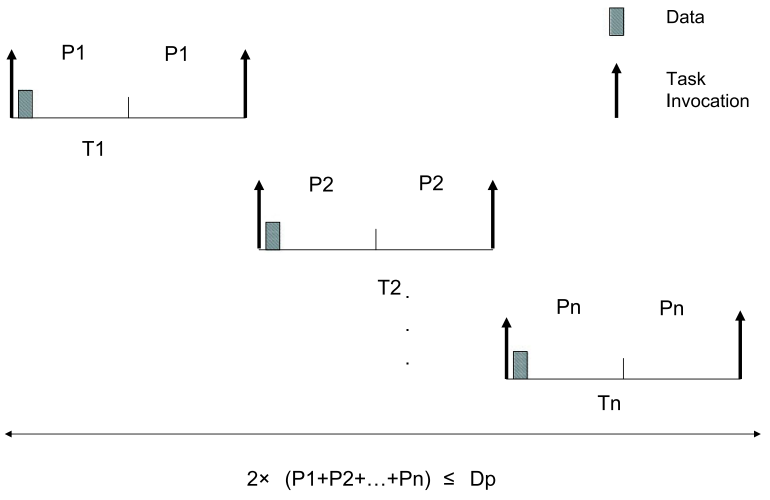


Fig. 2 End to end deadline constraints

within an end-to-end delay, D_p. Consider the flow of one byte of data within a sensor node. This byte, having been generated by a sensor, will wait for the next invocation of the first task on its path. In a system where tasks execute *independently* once per period, the maximum separation between two task invocations is upper bounded by two periods, which is the maximum waiting time of the packet on the next task. Once the task operates on its input data, it produces a result, which in turn may have to wait for up to two periods on the next task. An explanation of such case is shown in Fig. 2. Hence, for the end-to-end path deadline, D_p to be met, the batching periods must satisfy the constraint $2\sum_{i:T_i\in p} P_i \leq D_p$. This constraint is rewritten more conveniently to say that the sum of the batching periods must add up to no more than half the end-to-end deadline:

$$\sum_{i:T_i\in p} P_i \leq D_p/2 \tag{6}$$

This completes the formulation of the optimization problem.

4 Optimal batching periods

The problem formulated in the previous section can be easily solved using the method of Lagrange multipliers (Vapnyarskii 2001). First, we formulate the Lagrange function, L, to be minimized, which is defined as:

$$L = \sum_{i=1}^{n}\left(\frac{a_i^{k_i}}{P_i} + b_i^{k_i}\right) + \sum_{p=1}^{m}\lambda_p\left(\sum_{i:T_i\in p} P_i - D_p/2\right) \tag{7}$$

where λ_p is the Lagrange multiplier. The first summation term in (7) is the objective function to be optimized as discussed in Sect. 3. The second summation term represents the end-to-end deadline constraints to be satisfied by the batching periods.

Let us denote the optimal batching period of task T_i by P_i^*. Setting the derivative $dL/dP_i = 0$ at $P_i = P_i^*$ yields:

$$P_i^* = \sqrt{\frac{a_i^{k_i}}{\sum_{p:T_i \in p} \lambda_p}} \tag{8}$$

Similarly, obtaining the derivative $dL/d\lambda_p$ yields:

$$\sum_{i:T_i \in p} P_i^* = D_p/2 \tag{9}$$

The solution to the system of (8) and (9) can be computed numerically using the following pseudocode, which will converge to the optimal periods given a sufficiently small constant K:

```
loop
    $\forall i : P_i = \sqrt{\frac{a_i^{k_i}}{\sum_{p:T_i \in p} \lambda_p}}$
    $\forall p : \delta_p = K(\sum_{i:T_i \in p} P_i - D_p/2)$
    $\lambda_\mathrm{p} = \lambda_\mathrm{p} + \delta_\mathrm{p}$
end loop until $\delta_p$ is small enough
```

Below, we derive an analytic solution for any non-acyclic aggregation graph topology. Data aggregation or fusion is the most common function of sensor networks. To derive results for arbitrary directed acyclic graphs, we first consider the chain and star topology. For notational simplicity, since the results in this section are for a particular task allocation, we omit below the processor index from the processor-dependent constants $a_i^{k_i}$ and $b_i^{k_i}$. Hence, we shall use a_i and b_i to refer to the corresponding energy overheads of task T_i on the processor that T_i runs on.

4.1 The chain topology

In this section, we consider a set of n tasks, $T_1, \ldots, T_n$, that form a single path, p. Note that, the results are trivially generalizable to multiple independent paths, since they are applicable to each path separately. For a chain topology, since each task is precisely on one path, (8) reduces to:

$$P_i^* = \sqrt{\frac{a_i}{\lambda_p}} \tag{10}$$

Substituting for P_i^* in (9) and rearranging to solve for λ_p, we get:

$$\lambda_p = \frac{4(\sum_{i:T_i \in p} \sqrt{a_i})^2}{D_p^2} \tag{11}$$

Finally, substituting from (11) into (10), we get:

$$P_i^* = \frac{\sqrt{a_i}}{\sum_{i:T_i \in p} \sqrt{a_i}} \frac{D_p}{2} \tag{12}$$

The result is intuitive. First, note that the sum of the optimal batching periods of tasks on a given path p adds up to $D_p/2$, as expected. More interestingly, the periods of different tasks on the path split $D_p/2$ proportionally to the square root of their fixed energy cost a_i. This may be expected. Since the energy overhead a_i is spent every time the task runs (regardless of how much data it processes), tasks with a high a_i should run less often (i.e., have a higher batching period) than tasks with a small a_i. Note that, the data size dependent cost, b_i, does not affect period allocation. This might have been expected as well because, ultimately, the same amount of data are processed. Hence, the total energy spent on data processing does not depend on the batching period and does not affect the outcome of the optimization problem. In view of the above, we can state the following theorem:

Theorem 1 (Chain period allocation) *Given a set of n periodic tasks*, $T_1, \ldots, T_n$ *that form a single path*, *with an end-to-end delay constraint*, D, *where task* T_i *executes on processor* k_i, *the batching period of task* T_i *is* P_i, *and the energy expended by task* T_i *on processor* k_i *is* $a_i + b_i P_i$, *the optimal batching periods* $P_1^*, \ldots, P_n^*$ *partition* $D/2$ *proportionally to* $\sqrt{a_1} : \cdots : \sqrt{a_n}$.

Proof The proof follows trivially from (12). □

It is interesting to notice that a chain of n tasks, $T_1, \ldots, T_n$, described above, can be reduced to an equivalent single task, T_{eq}, in the sense that when T_{eq} is executed at its optimal batching period, P_{eq}^*, it consumes the same average power as the original chain of tasks, executing at their optimal batching periods. From (4), this means:

$$\frac{a_{eq}}{P_{eq}^*} + b_{eq} = \sum_{1 \le i \le n} \left(\frac{a_i}{P_i^*} + b_i \right) \tag{13}$$

Substituting for P_i^* from (12) and rearranging, we get:

$$\frac{a_{eq}}{P_{eq}^*} + b_{eq} = \frac{(\sum_{1 \le i \le n} \sqrt{a_i})^2}{D/2} + \sum_{1 \le i \le n} b_i \tag{14}$$

From (9), the optimal period, P_{eq}^*, is trivially $D/2$, for a single task. Substituting for $D/2$ with P_{eq}^* in (14) and matching the right hand side to the left hand side, we get:

$$a_{eq} = \left(\sum_{1 \le i \le n} \sqrt{a_i} \right)^2 \tag{15}$$

$$b_{eq} = \sum_{1 \le i \le n} b_i \tag{16}$$

This result is stated as the following theorem.

Theorem 2 (Chain reduction) *At their optimal batching periods, a set of n periodic tasks, $T_1, \dots, T_n$ that form a single path, is equivalent to a single task of $a_{eq} = (\sum_{1 \le i \le n} \sqrt{a_i})^2$ and $b_{eq} = \sum_{1 \le i \le n} b_i$.*

Proof The proof follows trivially from (15) and (16). □

4.2 The star topology

Consider a scenario where outputs of tasks $T_1, \dots, T_n$ are inputs to a single task T_0. Let us call the former, *leaf tasks* and the latter the *aggregator task*. Hence, there are n paths, where each path p is composed of task T_p and task T_0. We expect that T_0 fuses data that were collected around the same time. Hence, for meaningful aggregation, the end-to-end deadline, D_p, of each path p that merges into T_0 should usually be the same. Let us denote this common deadline by D. Equation (8) for the optimal period reduces to:

$$P_0^* = \sqrt{\frac{a_0}{\sum_{1 \le j \le n} \lambda_j}} \tag{17}$$

$$P_i^* = \sqrt{\frac{a_i}{\lambda_i}} \quad 1 \le i \le n \tag{18}$$

Substituting for P_0^* and P_i^* into (9), we get:

$$\sqrt{\frac{a_i}{\lambda_i}} + \sqrt{\frac{a_0}{\sum_{1 \le j \le n} \lambda_j}} = D/2 \tag{19}$$

Since both the right hand side and the second term of the left hand side are constants that do not depend on i, it follows that $\sqrt{a_i/(\lambda_i)}$ is constant or $\lambda_1/a_1 = \cdots = \lambda_n/a_n$, from which $\lambda_j = \lambda_i a_j/a_i$. Substituting in (19) and solving for λ_i, we get:

$$\lambda_i = \frac{4a_i}{D^2}\left(1 + \sqrt{\frac{a_0}{\sum_{1 \le j \le n} a_j}}\right)^2 \tag{20}$$

Finally, substituting for λ_i in (18), gives:

$$P_i^* = \frac{\sqrt{\sum_{1 \le j \le n} a_j}}{\sqrt{\sum_{1 \le j \le n} a_j} + \sqrt{a_0}} \frac{D}{2} \quad 1 \le i \le n \tag{21}$$

Observe that the equation states that the optimal period is the same for all leaf tasks. By subtracting from $D/2$, the optimal period of the aggregator task is:

$$P_0^* = \frac{\sqrt{a_0}}{\sqrt{\sum_{1 \le j \le n} a_j} + \sqrt{a_0}} \frac{D}{2} \tag{22}$$

In other words, in a star topology with an aggregator task T_0, the optimal periods of the aggregator task and the leaf tasks divide $D/2$ in proportion to $\sqrt{a_0}$ (for the aggregator) to $\sqrt{\sum_{1\le j\le n} a_j}$ (for each of the leaf tasks). This is consistent with the results of Sect. 4.1. Since the leaf tasks run in parallel at the same period, their energy overheads, a_i add up into one equivalent task of the combined fixed energy cost $\sum_{1\le j\le n} a_j$. That equivalent task is in a chain configuration with the aggregator task. From Sect. 4.1, we know that tasks in a chain split $D/2$ proportionally to the square root of their fixed energy costs, which leads to (22). The result is stated more formally as the following theorem.

Theorem 3 (Star period allocation) *Given a set of n periodic leaf tasks, $T_1, \dots, T_n$ in a star topology with an aggregator task T_0, and an end-to-end delay constraint, D, where the batching period of task T_i is P_i and the energy expended by task T_i on processor k is $a_i + b_i P_i$, the optimal batching periods $P_i^*, \dots, P_0^*$ on each path (T_i, T_0) partition $D/2$ proportionally to $\sqrt{\sum_{1\le j\le n} a_j}$, $\sqrt{a_0}$.*

Proof The proof follows trivially from (21) and (22). □

As in the case of the chain reduction theorem, it is now possible to prove the following.

Theorem 4 (The star reduction) *At their optimal batching periods, a set of n periodic tasks, $T_1, \dots, T_n$ that form leaves of a star, is equivalent to a single task of $a_{eq} = \sum_{1\le i\le n} a_i$ and $b_{eq} = \sum_{1\le i\le n} b_i$.*

Proof The proof follows the derivation steps of the chain reduction theorem and hence will not be repeated. Intuitively, the theorem arises from observing that leaf tasks execute at the same period and hence can be lumped together into one task of their aggregate energy consumption. □

4.3 Period allocation in aggregation trees

The most common topology for data flows on a sensor node is that of an aggregation tree. Typically data are collected from multiple sensors, filtered, processed, and then fused. The results stated in Theorems 1 through 4 allow optimal batching periods to be analytically computed for arbitrary aggregation trees. This is best illustrated by an example.

Figure 3a shows a system of five tasks, $T_1, \dots, T_5$, forming an aggregation tree sinked in T_5. The consumed fixed energy overhead a_i for the respective tasks is 4, 4, 1, 4, and 9, as shown in figure. The end-to-end deadline is 48 seconds. It is desired to optimally allocate batching periods.

We first use Theorem 2 to reduce tasks T_1 and T_2, that form a chain, into one equivalent task, called T_{12}, with $a_{12} = (\sqrt{4} + \sqrt{4})^2 = 16$. Similarly, tasks T_3 and T_4, that also form a chain, can be reduced into an equivalent task, T_{34}, with $a_{34} = (\sqrt{1} + \sqrt{4})^2 = 9$. Next, tasks T_{12} and T_{34}, that form leaves of a star with T_5 as the

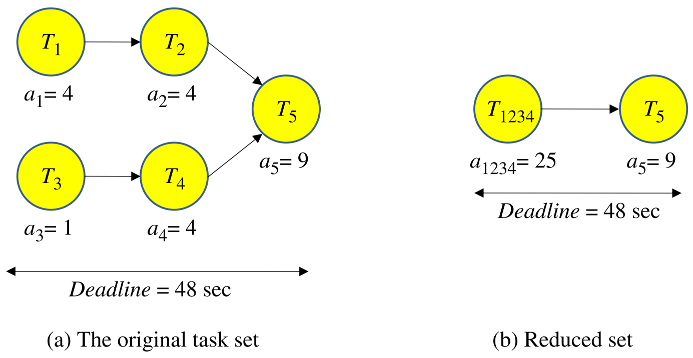


Fig. 3 Optimal period allocation

aggregator can be reduced by Theorem 4 into an equivalent task T_{1234} with $a_{1234} = 16 + 9 = 25$. This results in Fig. 3b. The figure shows a chain of two tasks. Theorem 1 says that their respective optimal batching periods split half the end-to-end deadline proportionally to $\sqrt{a_i}$ or in the ratio of 5:3. Hence, the optimal batching period of T_5 is $(3/8) * 24 = 9$ seconds. Each of the two chains must thus finish within $24 - 9 = 15$ seconds. By Theorem 1, the chain composed of tasks T_1 and T_2 split their 15 seconds equally, each getting a batching period of 7.5. Similarly, tasks T_3 and T_4 split their 15 seconds in the ratio 1:2. Hence, the optimal batching periods for tasks T_1 and T_2 are 5 and 10, respectively. Moreover, for data processing topology of cycles, it can basically be treated as a special chain topology where the output of the end task happens to be the input of the first task.

5 Task to processor assignment

The results presented in Sect. 4 determine the period as a function of parameters that depend on the allocation of tasks to processors. Hence, the period assignment problem is not entirely separable from the task-to-processor allocation. In general, the number of tasks on a sensor node is usually quite limited (e.g., 5–10). Given that each task has only two assignment options, the total number of possible assignments is tractable (30–1000). It is therefore entirely feasible to run the optimal period assignment for each possible task-to-processor allocation and to choose the allocation that results in the minimum energy solution to the optimal period assignment problem across all allocations. However, when the number of tasks becomes larger or the number of processors is more than two, such brute-force search of the optimal task-to-processor allocation can be time and resource consuming. For example, given the same number of tasks discussed above (i.e., 5–10), the total number of possible assignments for three processors increases significantly (200–60000). Therefore, it is also necessary to design a heuristic algorithm that has a high probability of finding the optimal solution directly and efficiently. In this section, we first describe the heuristic algorithm to find locally optimal task-to-processor assignment for a dual processor platform and then generalize it to a multiple processor platform.

Algorithm 1 Heuristic Task Allocation Algorithm

```
Start task allocation all on high-end processor
current_allocation = all_high;
new_allocation=NULL;
while $current\_allocation \neq new\_allocation$ do
  for $i = 1$:$i \leq task\_number$;$i++$ do
    if Inequality (23) not hold then
      Move task $T_i$ to lower-end processor
    end if
  end for
  Get new_allocation;
  Calculate the optimal batching periods for new task allocation
  if $iteration \geq iterate\_threshold$ then
    Break;
  end if
  $iteration++$;
end while
Find the Locally Optimal task assignment
```

Let the high-end processor be denoted by hi and the low-end processor by lo. If a_i^{hi} is large for a given task, T_i, it does not make sense to execute the task on that processor unless there is time to do enough batching to offset the overhead. From (4), for the energy consumed on executing a task, T_i, on the higher-end processor to be lower than that on the lower-end processor, the optimal batching period, P_i^*, must satisfy:

$$\frac{a_i^{hi}}{P_i^*} + b_i^{hi} < \frac{a_i^{lo}}{P_i^*} + b_i^{lo} \tag{23}$$

Hence, to find a locally optimal task-to-processor allocation, the following three-step process is conducted:

1. Run the optimal batching period assignment algorithm assuming that all tasks are allocated to the higher-end processor (i.e., using values of a_i^{hi}, not a_i^{lo}).
2. Test the resulting optimal batching periods for satisfaction of the period constraint stated by Inequality (23). If a task T_i fails the test, move it to the lower-end processor.
3. Repeat the optimal batching period assignment based on the new task to processor allocation. Check if the new task allocation is the same as the one before step 2. If they are different go back to step 2, otherwise the resulting periods and assignment are (locally) optimal.

The above steps are shown by the following pseudo code of Algorithm 1.

We can also extend the above algorithm of a dual processor platform to a generalized algorithm for a multiple processor platform (i.e., platform with more than two processors). Let the set of processors that task i can be allocated to on the platform be $K = (k_1^i, k_2^i, \ldots, k_M^i)$, where processors are sorted by the ascending order of their data-independent cost (i.e., $a_i^{k_i}$). Hence, processor k_1^i has the lowest a_i for task i while processor k_M^i has the highest. The optimal period constraint stated by (23) has been changed to:

$$\frac{a_i^{k_M^i}}{P_i^*} + b_i^{k_M^i} < \frac{a_i^{k_l^i}}{P_i^*} + b_i^{k_l^i} \quad l = 1, 2, \ldots, M - 1 \tag{24}$$

The generalized algorithm to find a locally optimal task-to-processor allocation for multiple processor platform is similar as the algorithm on dual processor platform, except the first two steps of the three-step process discussed above are changed to:

1. Run the optimal batching period assignment algorithm assuming that each task is allocated to the *highest-end* processor k_M^i respectively (i.e., using values of $a_i^{k_M^i}$).
2. Test the resulting optimal batching periods for satisfaction of the constraint stated by (24). If a task T_i fails the test, move it to the *lowest power-consuming* processor k_l^i for that task (i.e., processor that has the lowest power consumption of T_i).

6 Evaluation

In this section, we evaluate the performance of the proposed optimization on mPlatform. This mote platform represents the next generation of sensor nodes, that exploits heterogeneity as opposed to relying on low-end microcontrollers alone. This section is organized as follows. First, we profile the energy properties of different mPlatform processor boards. Then, we compare the batching period optimization approach to several baselines and evaluate the performance of heterogeneous allocation (with optimal batching periods) compared to running the task set on one of the processors of mPlatform alone. Moreover, we also evaluate the performance of the heuristic task allocation algorithm on heterogeneous boards against the brute-force optimal allocation scheme, and investigate the effect of task granularity on the energy consumption. Finally, we evaluate the energy cost inherent in implementing processing stages as independent, asynchronously executed tasks.

6.1 Energy profiling

As we mentioned earlier, the low-end and high-end processors have their unique but different power characteristics and types of instructions that they are more energy-efficient at. We first carry out experiments to profile the energy properties of the two types of processor boards on mPlatform. The low-end processor board is equipped with an MSP430F2618 processor while the high-end processor board is equipped with an ARM LPC2138 processor.

In our experiments, we monitor, through an oscilloscope, the total real-time current of the entire processor board while running tasks. To ensure that the oscilloscope is synchronized with task execution, we use a pulse, toggled in software, on a GPIO pin of each processor. The pulse on this pin is used to trigger the horizontal scan of the oscilloscope, essentially causing it to display the waveform of the current consumed

Table 1 Energy profiling comparison of MSP board and ARM board. Board supply voltage is 4.5 V

Parameter	MSP	ARM
Frequency	16 MHz	60 MHz
Active current	8.61 mA	75 mA
Active power	38.745 mW	337.5 mW
Sleep current	0.017 mA	0.15 mA
Sleep power	0.0765 mW	0.675 mW
Wakeup time	0.7 ms	3 ms
Wakeup energy	7.43 μJ	217.4 μJ
Flash access energy	0.826 μJ/byte	1.422 μJ/byte
Inter-board transfer time	2 μs/byte	
Inter-board transfer energy	0.65 μJ/byte	
Sensing energy	1.64 μJ/byte	

by the task, which is measured from the voltage drop across a small resistor (7.1 Ω) that is in series with the mPlatform node. The integral of the current readings over the execution time of the task (multiplied by processor board voltage) yields the total energy the task consumes. The measured energy profiles of two types of processor boards in basic states is summarized in Table 1. By comparing the ARM board with the MSP board, we observe that the ARM board has higher active power, sleep power, wakeup and flash access cost than the MSP board. Moreover, as the sensor resides on the MSP board, data to be processed on the ARM board need to be transferred from the MSP board, inter-board transfer overhead is given in the table.

The ARM board can only be more energy efficient than the MSP board when b_i^{ARM} is smaller than MSP b_i^{MSP}. Table 1 compares the basic energy characteristics of the two processors. To compare energy expended on computation, one also needs to understand how efficient each processor is at processing different instructions and data types. Table 2 compares the times and energy spent in performing some basic operations by the ARM and MSP processor boards on different data types. Please note that these numbers are for the entire board and hence include energy consumption by all circuitry involved. Observe that different operations and data types have different energy efficiency on different boards. To be more specific, according to the table, the ARM board is more energy efficient at multiplication and division for most data types than the MSP board. This is most obvious for the uint_32 (long integer) data type. In contrast, the MSP board is better at other operations like addition, subtraction, bit operations, relations and logic. This is likely because the ARM processor is a 32-bit architecture which is good at handling long data types and complex operators while the MSP processor is a 16-bit architecture which is good at handling short data types and simple operators. Numbers in bold in Table 2 highlight which board is more energy efficient when. The results from this experiment give us an idea of what kinds of tasks will be more energy efficient on each processor board.

6.2 Task set generation

In order to evaluate energy efficiency of representative data processing, we select some representative routines in wireless sensor networks and digital signal processing

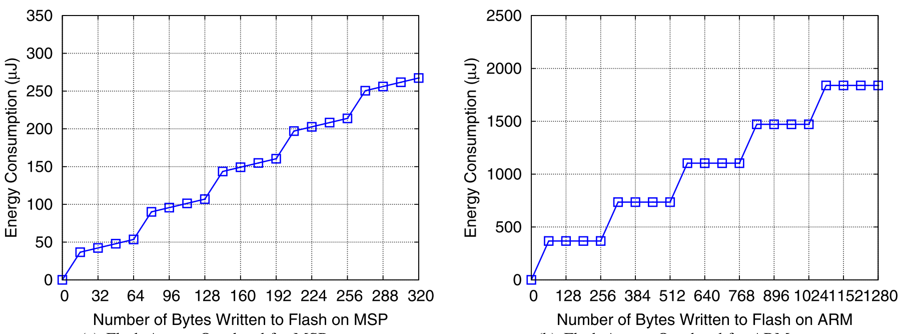
Table 2 Comparison of basic operations on two processor boards across different data types

OPERATION			ARM		MSP	
		Data Type	Time (μs)	Energy (μJ)	Time (μs)	Energy (μJ)
ARITHMETIC	Multiply	uint_32	0.66	**0.22275**	16.2	0.62767
		uint_16	0.66	**0.22275**	9.8	0.37970
		float	1.21	**0.40838**	20.6	0.79815
		double	1.9	**0.64125**	20.9	0.80977
	Divide	uint_32	1.12	**0.378**	26.5	1.02674
		uint_16	1.12	**0.378**	10.1	0.39132
		float	2.45	**0.82688**	26.2	1.01512
		double	8.32	2.808	26.2	**1.01512**
	Add	uint_32	0.61	0.20588	2.2	**0.08524**
		uint_16	0.66	0.22275	1.4	**0.05424**
		float	1.5	0.50625	10.1	**0.39132**
		double	2.1	0.70875	10.2	**0.3952**
	Subtract	uint_32	0.61	0.20588	2.2	**0.08524**
		uint_16	0.66	0.22275	1.4	**0.05424**
		float	1.5	0.50625	10.1	**0.39132**
		double	2.2	0.7425	10.2	**0.3952**
BIT OPERATION	AND	uint_32	0.48	0.162	1.6	**0.06199**
		uint_16	0.48	0.162	1.2	**0.04649**
	OR	uint_32	0.48	0.162	1.68	**0.06509**
		uint_16	0.49	0.16538	1.2	**0.04649**
	XOR	uint_32	0.49	0.16538	1.6	**0.06199**
		uint_16	0.49	0.16538	1.2	**0.04649**
	SHIFT	uint_32	0.46	0.15525	3.7	**0.14336**
		uint_16	0.5	0.16875	3.4	**0.13173**
RELATION	$\leq \geq$ $\equiv \neq$	uint_32	0.64	0.216	2.4	**0.09299**
		uint_16	0.68	0.2295	1.7	**0.06587**
		float	1.18	0.39825	3.6	**0.13948**
		double	1.35	0.45563	3.6	**0.13948**
LOGIC	AND OR NOT	All	0.31	0.10463	0.7	**0.02712**

to construct our task sets. The routines we implement and use in our experiments are: Digital Filter, Fast Fourier Transform (FFT), Statistics (mean, standard derivation, correlation), Cyclic Redundancy Check (CRC), Checksum, Encryption and Decryption. The type and parameters used for task routine generation are shown in Table 3. By pipelining these basic routines, we create several task templates that represent typical data processing and aggregation flows in sensor networks, including both chain and star topologies discussed in Sect. 4. For example, a chain template might

Table 3 Task routine type and parameters

Task routine	Type/Parameters
Digital filter	Type: FIR; Order: 2-8
FFT	Number of Points: 8
Statistics	Type: Mean, Standard Deviation, Correlation
CRC	Length: 16 bits
Checksum	Type: Parity Word; Length: 16 bits
Encryption/Decryption	Type: XOR Cipher



(a) Flash Access Overhead for MSP (b) Flash Access Overhead for ARM

Fig. 4 Flash access overhead for MSP and ARM

be given by the regular expression: $(Filter)(Statistics)(Filter)(FFT)(CRC)(Encrypt)$. Each data flow is an instantiation of one such template.

Moreover, each of the above routines is parameterized to save and restore a different amount of state in Flash memory when the processor goes to sleep. For example, the digital filter needs to save a different amount of state depending on the order of the filter. Such flash access overhead is data-independent and encountered once every batching period (because state can be stored in RAM until the batch is finished). The measured flash access energy profiles for both MSP and ARM boards are shown in Fig. 4. Observe that the energy consumed on flash access is a step-like function of the number of bytes written. Because expensive flash operations happen at block granularity, there are jumps at block boundaries (and different processors have a different block size). Thus, for a task i, the a_i value is calculated as $a_i = a_i^{wakeup} + a_i^{state}$, where a_i^{wakeup} is the processor wakeup cost and a_i^{state} is the overhead of saving and restoring state into Flash memory. The b_i value is given by $b_i = b_i^{proc} + b_i^{comm}$, where b_i^{proc} is the cost to process the data that can be computed by having each routine process an increasing number of data units and computing the slope of the energy curve with data size, and b_i^{comm} is the read and communication overhead to send data to the appropriate processor board.

6.3 Experiments with batching periods

In this section, we evaluate the performance of the optimal batching period assignment on a single processor, comparing it with several baseline approaches. The heterogeneous processor assignment is evaluated in Sect. 6.4. We use the task model presented in Sect. 4, and carry out experiments for both the chain and star topology on MSP and ARM boards respectively. We incorporate all the overheads mentioned in Sect. 6.1 in the evaluation. The power and energy numbers used are from the measurements listed in Table 1.

We first evaluate the batching period assignment on the MSP board. For each topology, we generated 20 workflows, each of them is selected from the task templates we discussed in the previous section. We adopt an end-to-end deadline of 48 seconds. The input data rate is set as 300 Bytes/s. The optimal batching period assignment is compared to a uniform period assignment (all periods are the same) and a random assignment. For fairness, all assignments satisfy the constraint that the sum of the periods adds up to half the path deadline. Each data point on a graph is repeated 1000 times and the average power consumption is computed. For the random assignment, we also show the maximum and minimum power consumption across the 1000 experiments.

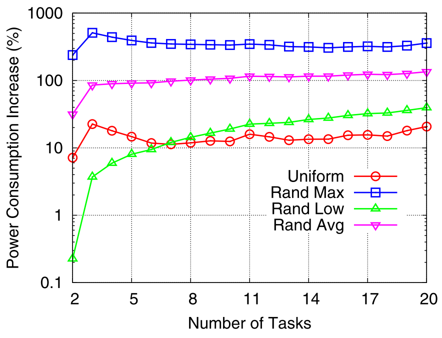
Figure 5 demonstrates the results for the chain topology on an MSP board. The X-axis is the number of tasks. The Y-axis is the *increase* (in percentage) of power consumption compared to the optimal case. Since the optimal case is used as an implicit baseline, it is not plotted. For the random case, the maximum, minimum, and average increase are plotted. Observe that the optimal period assignment always achieves lower power consumption compared to other baselines.

Figure 6 demonstrates the results of the experiment repeated for the star topology on the MSP board. Consistent with the previous example, we compared the random and uniform period assignment to the optimal period assignment. Again, we observe that the optimal period assignment achieves a lower power consumption compared to the other approaches.

We repeat the same experiments for the two topologies on the ARM boards as well. Results are shown in Figs. 7 and 8. We observe that the optimal batching period assignment is better than other approaches in terms of average power consumption.

The above experiments are carried out under the constant input data rate. However, in some sensor network applications (e.g., event-driven applications), the input data

Fig. 5 Comparison for chain topology on MSP versus task number



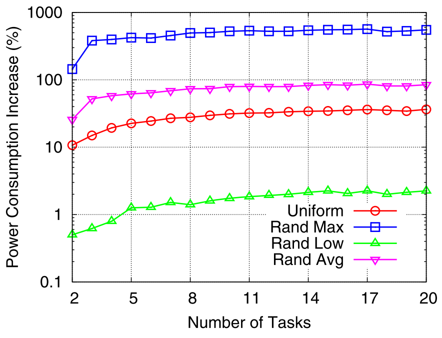


Fig. 6 Comparison for star topology on MSP versus task number

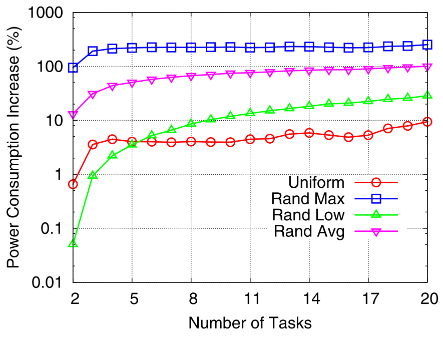


Fig. 7 Comparison for chain topology on ARM versus task number

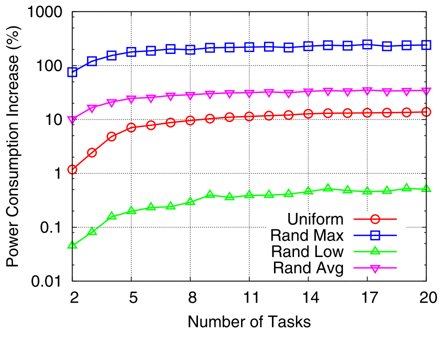


Fig. 8 Comparison for star topology on ARM versus task number

rate may not always be constant. Hence, it is also interesting to investigate how the variation of the input data rate (and hence task computation time) affects the power consumption savings of the optimal period assignment compared to other baselines. We repeated the above experiments to show such effect by varying the variation of the input data rate. The task number is fixed at 10 and the average input data rate is set as 300 Bytes/s. We vary the standard deviation of the input data rate from 10 to 50 Bytes/s. Each data point on the graph is repeated 1000 times and the average power consumption is computed.

Figure 9 demonstrates the result of chain topology on MSP board. Observe that the optimal period assignment continues to achieve lower power consumption compared to other baselines and the variation of input data rate does not affect the power con-

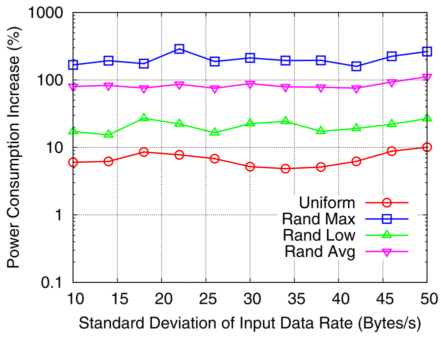


Fig. 9 Comparison for chain topology on MSP versus input data rate variation

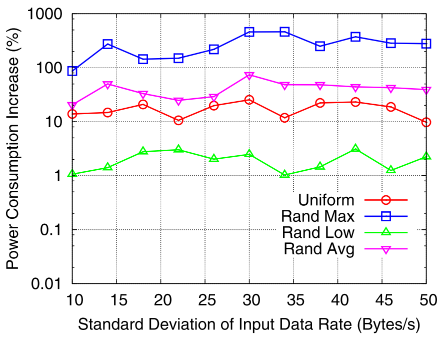


Fig. 10 Comparison for star topology on MSP versus input data rate variation

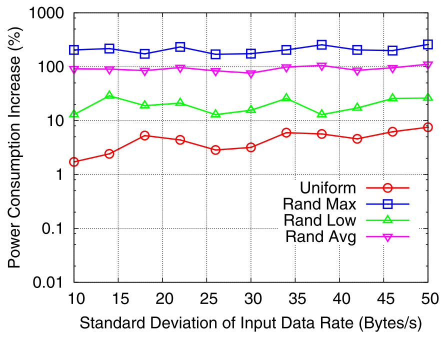


Fig. 11 Comparison for chain topology on ARM versus input data rate variation

sumption savings of the optimal period assignment significantly. The reason is that the data rate is incorporated into the data dependent average power consumption parameter (i.e., b_i^k) defined in (3), which is proven to be independent of the period allocation of the optimal assignment scheme. Figure 10 shows the result of star topology on MSP board. We also observe that the optimal period assignment scheme achieves the lowest power consumption and the variation of input data rate has limited effect on the power consumption savings achieved. We repeat the same experiments on the ARM processor. Results are shown in Figs. 11 and 12. Similar results on the MSP board are observed on the ARM board as well.

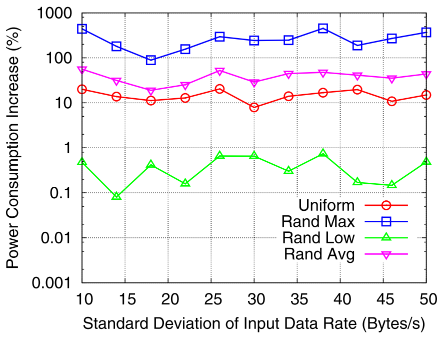


Fig. 12 Comparison for star topology on ARM versus input data rate variation

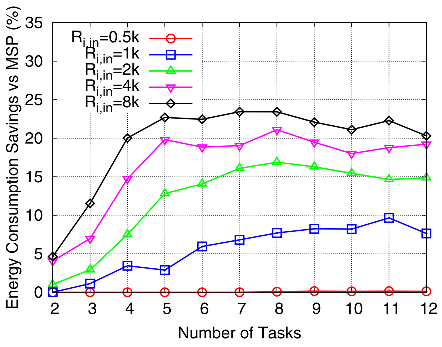


Fig. 13 Heterogeneous assignment versus MSP

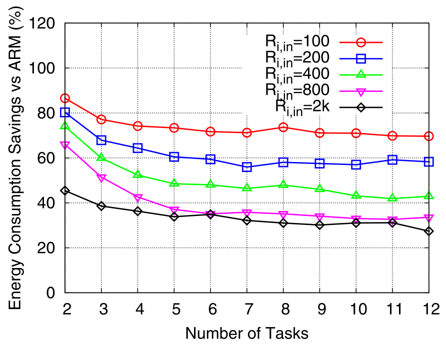


Fig. 14 Heterogeneous assignment versus ARM

6.4 Experiments with optimal task assignment

In this section, we compare the performance of assigning tasks to only one of processor boards vs the optimal heterogeneous processor board assignment. Reported results are averaged over 50 experiments. Figures 13 and 14 compare the energy consumed by the heterogeneous assignment to assignment on the MSP only and the ARM only, respectively. Observe that utilizing both processors saves a considerable amount of energy compared to using MSP alone (nearly 25% savings) or ARM alone (around 80% savings). The figures show energy savings for a different number of tasks under varying $R_{i,in}$ values. Interestingly, compared to the MSP processor, heterogeneous assignment saves more energy when the $R_{i,in}$ is larger. In contrast, for

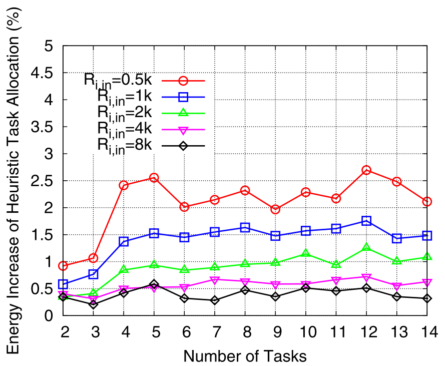


Fig. 15 Energy increase of heuristic task allocation versus input rate on mPlatform

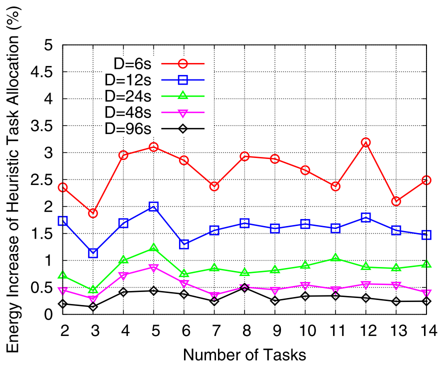


Fig. 16 Energy increase of heuristic task allocation versus deadline on mPlatform

the ARM, the algorithm saves more when the $R_{i,in}$ is smaller. The reason is that tasks that have a smaller b_i on the ARM board than on the MSP board need to process a certain amount data in each batching period to get enough energy savings to overcome the fixed overheads. Therefore, having a higher data rate for such tasks makes the ARM more efficient (while a lower rate favors the MSP). Results from above experiments validate our claim that we can achieve better energy efficiency by exploiting the processor heterogeneity with an optimal batching period allocation in sensing applications.

6.5 Experiments with heuristic task allocation and task granularity

In this section, we first evaluate the performance of the heuristic task allocation algorithm we proposed in Sect. 5 as compared to the brute-force optimal task allocation scheme on mPlatform. Reported results are averaged over 100 experiments. The heuristic algorithm is much faster and more efficient to implement than the brute-force one considering the limited energy and resource on motes. Figures 15 and 16 show the energy penalty for the heuristic task allocation algorithm compared to the brute-force optimal scheme across heterogeneous processor boards. Observe that the energy increase of the heuristic task allocation compared to the brute-force optimal scheme is reasonably small (less than 4%) under various input rates and deadlines. In other words, the proposed heuristic task allocation algorithm is able to find the global

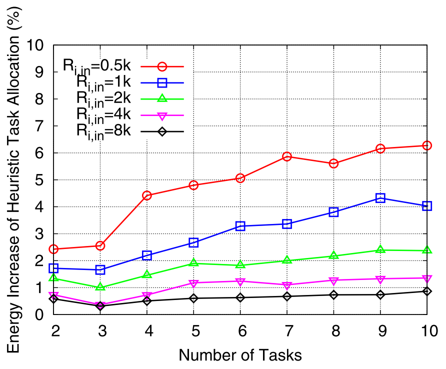


Fig. 17 Energy increase of heuristic task allocation versus input rate on an emulated 3 processor platform

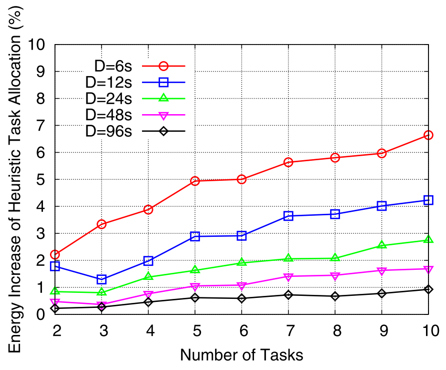


Fig. 18 Energy increase of heuristic task allocation versus deadline on an emulated 3 processor platform

optimal task allocation with a high probability. Interestingly, the energy increase is more obvious when the input rate $R_{i,in}$ and deadline D are smaller. The reason is that the energy consumption on MSP and ARM boards becomes similar with small $R_{i,in}$ and D values. The similarity provides more opportunities for the global optimal allocation found by the brute-force scheme to be different from the local optimal allocation found by the heuristic algorithm.

Furthermore, we discussed about the generalization of the proposed heuristic scheme to a multiple processor platform in Sect. 5. Since mPlatform only has two type of processors (i.e., MSP and ARM) on boards, for evaluation purpose, we emulate a 3-processor platform with the third type of processor chosen as ATmega128L (i.e., the processor used by MicaZ mote). The power parameters of the ATmega128L processor emulated are taken from its datasheet.[2] The inter-board communication cost of the emulated platform is assumed to be the same as mPlatform. We run simulations of the above experiments on the emulated 3-processor platform and show the performance of the generalized heuristics for multiple processor platform. Figures 17 and 18 show the energy penalty for the generalized heuristic task allocation algorithm for multiple processors compared to the brute-force optimal scheme across the emulated 3-processor platform. Observe that the energy increase of the generalized

[2]See http://www.datasheetarchive.com/ATMEGA128L-datasheet.html.

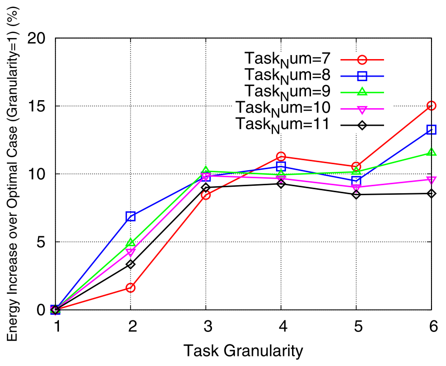


Fig. 19 Energy cost versus task granularity

heuristic task allocation scheme compared to the brute-force optimal scheme is larger than the dual processor case on mPlatform. However, it still remains to be reasonably small under various input rates and deadlines. The reason for such energy increase is that the larger the number of different types of processors, the more chances that the heuristic allocation scheme can deviate from the global optimal one. Results verify the effectiveness of the proposed generalized heuristics to find the optimal task to processor allocation across multiple processor platform.

Additionally, we also evaluate the effect of task granularity on the energy consumption of the platform. The task granularity here is defined as the number of task routines that are tied up together (i.e. written, compiled and programmed) to run on a single processor board. Figure 19 shows the result of energy consumption increase of the platform under varying task granularity compared to the optimal case (i.e. $granularity = 1$) with different size of task set. Observe that the energy cost increases as the task granularity increases. This is intuitive: as more tasks are bound together on a single board, the task allocation scheme will have less flexibility in manipulating tasks across different boards. This trade off can give helpful hints to programmers in implementing and compiling their task routines over heterogeneous platforms.

6.6 The cost of asynchrony

The reader is reminded that the paper starts with an assumption on application structure. Namely, we consider applications where the processing of each data flow is structured as a set of independent periodic tasks, each executing on the flow independently, without synchronization with other tasks. Buffers between stages make such independence possible. The approach is motivated by advantages of simplicity, separation of concerns, and possibly increased reliability as a result of fewer bugs, compared to designs where synchronization primitives are used to trigger stage-execution in a synchronized fashion. Next we examine the cost paid for asynchrony. To do so, we compare our asynchronous optimal period assignment to the synchronous case where all processing stages are lumped in one that executes at a period equal to the end-to-end data processing deadline. The energy consumption difference of two schemes is given by:

$$E_{asyn} - E_{syn} = \sum_i a_i \times \left(\left\lfloor \frac{D_p}{P_i^*} \right\rfloor - 1 \right) \tag{25}$$

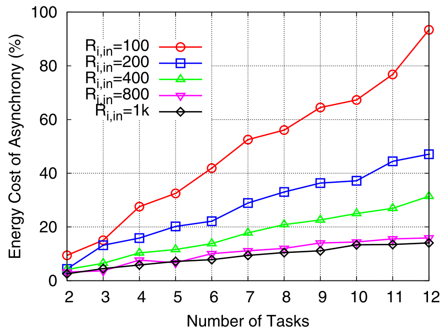


Fig. 20 The cost of asynchronous scheme compared to synchronous scheme

where, E_{asyn} and E_{syn} denote the energy consumption of the asynchronous and synchronous scheme respectively. Observe that the extra energy cost of asynchronous scheme comes from the data-independent cost (i.e., a_i) of processors. This is because processors may wake up multiple times under asynchronous scheme while they only wake up once under synchronous scheme. Meanwhile, the data-dependent costs of two schemes are the same (i.e., they both process the same amount of data).

The normalized cost of asynchrony is:

$$\begin{aligned} C_{asyn} &= \frac{E_{asyn} - E_{syn}}{E_{syn}} \\ &= \frac{\sum_i a_i \times \left(\lfloor \frac{D_p}{P_i^*} \rfloor - 1\right)}{\sum_i a_i + b_i D_p} \end{aligned} \tag{26}$$

Note that $P_i^* < D_p$ for all i and $\lfloor \frac{D_p}{P_i^*} \rfloor > 1$ for some i in the asynchronous scheme, hence $C_{asyn} > 0$. This indicates that the synchronous scheme should always use less energy than the asynchronous one as it allows for more batching and less wake up of processors.

We run same experiments as in the previous sections for the two approaches. Figure 20 shows the overhead paid in our optimal period assignment compared to the synchronized case. Observe that for a large number of tasks and small data rates, the cost of asynchrony is relatively high. This is because the average batching period becomes smaller in our approach as the number of tasks increases. When the data rate is small, the data-dependent energy component for both approaches shrinks, magnifying the effects of the data-size-independent cost encountered. We conclude that application designers should choose with care which approach to use. The choice may depend on many factors including component availability nature of interfaces, and of course energy implications. For a system that uses the asynchronous approach, our contribution lies in optimizing performance while maintaining independence among processing stages.

6.7 The trade-offs between energy savings and responsiveness

In some sensor network applications (e.g., health monitoring), the network responsiveness is the primary concern. The model proposed in Sect. 3 incorporates the end-

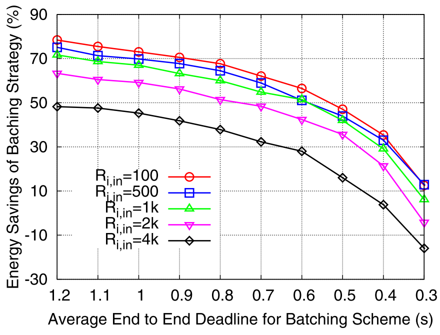


Fig. 21 Trade-offs between energy savings and responsiveness

to-end deadline of data processing on a sensor node, which can help impose the responsiveness guarantees on such applications. Therefore, it is interesting to show the trade-offs between energy savings achieved by the proposed batching scheme and the average end-to-end deadline of data processing on a sensor node. We run the same experiments as in the previous sections and show the energy savings of the proposed optimal batching scheme compared to the scheme where no batching is done (i.e., processors are always kept on) under different average data processing deadline and input data rate. We set the number of tasks to be 10 and vary the average end-to-end deadline from 1.2 to 0.3 s. Reported results are averaged over 100 experiments.

Figure 21 shows the trade-offs between the energy savings achieved by the proposed batching scheme and the average end-to-end deadline of data processing on a sensor node. Observe that the energy savings of the optimal batching scheme drops as the average deadline of data processing decreases. This is intuitive: the batching period decreases when the average deadline of data processing decreases, which directly shortens the time that processors stay in the sleep mode. After the deadline of data processing falls below certain values, the proposed batching scheme stops gaining energy savings. The reason is that the batching period is so small under those conditions that the processor wakeup cost overweights the energy savings obtained during the sleeping period. Also note that the energy savings is high when the input data rate is low. This is because lower data rate results in shorter task computation time, which allows the processor to sleep longer within a batching period.

6.8 The impact of batching on network communication

We also note that the proposed batching scheme has an impact on the network communication. In particular, the network traffic becomes more bursty when the batching is done more aggressively (i.e., with longer batching periods) on sensor nodes. We study the effect of our proposed batching scheme on the network traffic by simulation, where multiple sensor nodes have to share the radio channel to concurrently transmit data. The simulation is implemented in ns-2.34 simulator with IEEE 802.15.4 module developed by CUNNY.[3] In the simulation scenario, we run our experiments under a typical one-hop neighborhood of a sensor network where a node is exposed to the

[3] ns-2 network simulator: http://www.isi.edu/nsnam/ns/index.html.

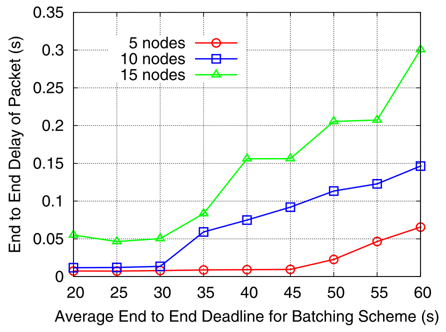


Fig. 22 The batching impact on end to end delay of packets

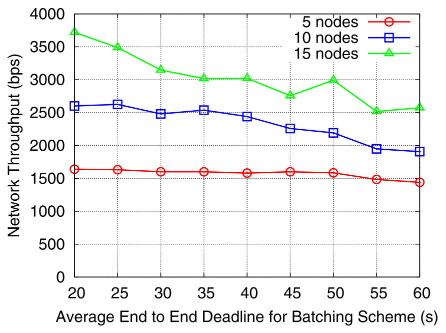


Fig. 23 The batching impact on network throughput

traffic from multiple neighboring nodes. Specifically, there are one coordinator in the center of the neighborhood and 5 to 15 sensor nodes with the same distance away from the coordinator. All nodes including the coordinator are in each other's transmission range. At each node, data transmission is invoked at the period that equals to the end-to-end deadline of data processing on that sensor node to transmit all the processed data accumulated over the period. Data packets are transmitted from sensor nodes to the coordinator. The packet size is set to 100 Bytes and packet interval is set to 0.05 s. The simulation time is 10000 seconds.

Figure 22 shows the effect of batching on the end-to-end delay of packets. The X axis is the average deadline of data processing of the proposed batching scheme on a sensor node and the Y axis is the end-to-end delay of packets. Observe that the end-to-end delay of packets increases as the average deadline of data processing increases. This is because the network traffic becomes more bursty as the average deadline of data processing grows. It is then more difficult for the MAC protocol to accommodate nodes that have conflicting transmission schedule to finish the data transmission within their periods. We also observe that the end-to-end delay is longer when the number of nodes in the neighborhood becomes larger. The reason is that the probability of two nodes starting to transmit at the same time increases as the network becomes denser, hence the back-off scheme of the MAC protocol will further delay the packet transmission.

Figure 23 shows the effect of batching on the network throughput. Observe that the network throughput degrades gracefully as the average deadline of data processing on a sensor node increases. This is attributed to the contention avoidance feature of the CSMA/CA protocol implemented on 802.15.4 nodes. Even though the traffic becomes more bursty as the average deadline of data processing becomes larger, the CSMA/CA still manages to deliver most of the packets when the network is not very dense. Moreover, we also observe that the network throughput degradation becomes more significant when the number of nodes in the network increases.

7 Conclusions

This paper describes how to optimally amortize energy overheads by batching sensor data processing, when sensor data flows are processed asynchronously by stages implemented as independent periodic tasks. An algorithm was developed for computing the optimal batching period for tasks involved in sensory data processing, with a special emphasis on aggregation trees. Experimental results, measured on mPlatform, show that the optimal batching period algorithm saves energy over other baselines for batching period assignment. Results also show that running some of the batched tasks on a heterogeneous processor platform (e.g., mPlatform with ARM and MSP) can save energy compared to running all the tasks on either the lower-end processor (e.g., MSP) alone or the higher-end processor (e.g., ARM) alone. This approach is useful for saving energy in sensor applications where sensor data pipelines are manipulated by independently executed periodic stages.

References

Acquaviva A, Benini L, Riccò B (2001) Processor frequency setting for energy minimization of streaming multimedia application. In: CODES, pp 249–253

Agarwal Y, Hodges S, Scott J, Chandra R, Bahl P, Gupta R (2009) Augmenting network interfaces to reduce pc energy usage. In: USENIX symposium on networked systems design and implementation (NSDI '09)

Baruah S (2004) Cost efficient synthesis of real-time systems upon heterogeneous multiprocessor platforms. In: Proc of 18th international parallel and distributed processing symposium (IPDPS'04), pp 120–128

Benini L, Bogliolo A, De Micheli G (2002) A survey of design techniques for system-level dynamic power management, pp 231–248

Benveniste A, Caspi P, Edwards S, Halbwachs N, Le Guernic P, de Simone R (2003) The synchronous languages 12 years later. Proc IEEE 91(1):64–83

Bhattacharyya SS, Murthy PK, Lee EA (1999) Synthesis of embedded software from synchronous dataflow specifications. J VLSI Signal Process Syst 21(2):151–166

Buck JT (1993) Scheduling dynamic dataflow graphs with bounded memory using the token flow model. PhD thesis

Cai L, Lu Y-H (2004) Dynamic power management using data buffers. In: DATE '04: Proceedings of the conference on design, automation and test in Europe, Washington, DC, USA. IEEE Computer Society, New York, pp 526–531

Cao Q, Wang D, Abdelzaher T, Priyantha B, Liu J, Zhao F (2010) Energy-optimal batching periods for asynchronous multistage data processing on sensor nodes: foundations and an mPlatform case study. In: IEEE RTAS, Stockholm, Sweden. IEEE Computer Society, New York

Chao L-F, Sha EH-M (1997) Scheduling data-flow graphs via retiming and unfolding. IEEE Trans Parallel Distrib Syst 8(12):1259–1267

Chu D, Popa L, Tavakoli A, Hellerstein JM, Levis P, Shenker S, Stoica I (2007) The design and implementation of a declarative sensor network system. In: SenSys '07: Proceedings of the 5th international conference on embedded networked sensor systems. ACM, New York, pp 175–188

Girod L, Jamieson K, Mei Y, Newton R, Rost S, Thiagarajan A, Balakrishnan H, Madden S (2006) Wavescope: a signal-oriented data stream management system. In: SenSys '06: Proceedings of the 4th international conference on embedded networked sensor systems. ACM, New York, pp 421–422

Goraczko M, Liu J, Lymberopoulos D, Matic S, Priyantha B, Zhao F (2008) Energy-optimal software partitioning in heterogeneous multiprocessor embedded systems. In: DAC, pp 191–196

Ha S, Lee EA (1997) Compile-time scheduling of dynamic constructs in dataflow program graphs. IEEE Trans Comput 46(7):768–778

Henzinger TA, Horowitz B, Kirsch CM (2001) Giotto: a time-triggered language for embedded programming. In: EMSOFT, pp 166–184

Jin Y, Satish N, Ravindran K, Keutzer K (2005) An automated exploration framework for FPGA-based soft multiprocessor systems. In: CODES+ISSS. ACM, New York, pp 273–278

Khemka A, Shyamasundar RK (1997) An optimal multiprocessor real-time scheduling algorithm. J Parallel Distrib Comput 43(1):37–45

Krishna CM, Lee Y-H (2000) Voltage-clock-scaling adaptive scheduling techniques for low power in hard real-time systems. In: RTAS, pp 156–165

Lee EA, Messerschmitt DG (1987) Static scheduling of synchronous data flow programs for digital signal processing. IEEE Trans Comput 36(1):24–35

Lu Y-H, Benini L, Micheli GD (2002) Dynamic frequency scaling with buffer insertion for mixed workloads. IEEE Trans CAD Integr Circuits Syst 21(11):1284–1305

Luo J, Jha, NK (2000) Power-conscious joint scheduling of periodic task graphs and aperiodic tasks in distributed real-time embedded systems. In: ICCAD. IEEE Press, New York, pp 357–364

Lymberopoulos D, Priyantha B, Zhao F (2007) mPlatform: a reconfigurable architecture and efficient data sharing mechanism for modular sensor nodes. In: IPSN '07

Lymberopoulos D, Priyantha NB, Goraczko M, Zhao F (2008) Towards energy efficient design of multiradio platforms for wireless sensor networks. In: IPSN '08

Ning X, Cassandras CG (2007) Message batching in wireless sensor networks—a perturbation analysis approach. In: Proceedings of the 46th IEEE conference on decision and control

Pavlovski C, Boyd C (199) Efficient batch signature generation using tree structures. Technical Report CrypTEC'99, City University of Hong Kong

Sengul C, Bakht M, Harris AF, Abdelzaher T, Kravets R (2008) Improving energy conservation using bulk transmission over high-power radios in sensor networks. In: ICDCS '08

Shin Y, Choi K, Sakurai T (2000) Power optimization of real-time embedded systems on variable speed processors. In: CAD, pp 365–368

Shih E, Bahl P, Sinclair MJ (2002) Wake on wireless: an event driven energy saving strategy for battery operated devices. In: MobiCom '02: Proceedings of the 8th annual international conference on mobile computing and networking

Sivanthi T, Killat U (2004) Global scheduling of periodic tasks in a decentralized real-time control system. In: IEEE IWFCS. IEEE Press, New York

Sorber J, Banerjee N, Corner MD, Rollins S (2005) Turducken: hierarchical power management for mobile devices. In: MobiSys '05: Proceedings of the 3rd international conference on mobile systems, applications, and services

Vapnyarskii I (2001) Lagrange multipliers. In: Hazewinkel M (ed) Encyclopaedia of mathematics. Springer, Berlin

Whitehouse K, Zhao F, Liu J (2006) Semantic streams: a framework for composable semantic interpretation of sensor data. In: EWSN, pp 5–20

Youn T-Y, Park Y-H, Kwon T, Kwon S, Lim JJ (2008) Efficient flexible batch signing techniques for imbalanced communication applications. IEICE Trans Inf Syst, pp 1481–1484, May 2008

Zheng W, Chong J, Pinello C, Kanajan S, Sangiovanni-Vincentelli AL (2005) Extensible and scalable time triggered scheduling. In: ACSD. IEEE Computer Society, New York, pp 132–141

Zhong L, Jha H (2004) Dynamic power optimization of interactive systems. In: 17th International conference on VLSI design, pp 1041–1047

Dong Wang received the BEng degree in communication and information systems from University of Electronic Science and Technology of China (UESTC), Chengdu, China, in 2004, and the MS degree in Electrical and Computer Engineering from Peking University, Beijing, China, in 2007. He is now a PhD candidate of Computer Science Department at the University of Illinois at Urbana Champaign. His research interests include Quality of Information quantification for social sensing, cost-aware prediction for data fusion systems and energy-aware real-time scheduling of heterogeneous sensor platforms.

Tarek Abdelzaher received the PhD degree from the University of Michigan, Ann Arbor, in 1999, under Professor Kang Shin. From August 1999 to 2005, he was an assistant professor at the University of Virginia. Then, he joined the University of Illinois at Urbana Champaign as an associate professor with tenure. His research interests include operating systems, networking, sensor networks, distributed systems, and embedded real-time systems. Especially, he is interested in developing theory, architectural support, and computing abstractions for predictability in software systems, motivated by the increasing software complexity and the growing sources of nondeterminism. Applications range from sensor networks to large-scale server farms and from avionics to homeland defense. He is a member of the IEEE.

Bodhi Priyantha received his BSc in Electronic Engineering from University of Moratuwa in 1996 and the PhD in Electrical Engineering & Computer Science from MIT in 2005. He is currently working in Networked Embedded Computing Group of Microsoft Research. His research interests include low-power systems design, wireless communication, and networked sensing.

Jie Liu received his my Bachelor (1993) and Master (1996) degrees from Department of Automation, Tsinghua University, Beijing, China and the PhD degree from EECS, UC Berkeley in 2001. He is currently a Principal Researcher and a Research Manager at Microsoft Research, leading the Sensing and Energy Research Group. Before joining MSR in May 2004, he was a researcher at Palo Alto Research Center (formerly Xerox PARC). His research interests include understanding and managing the physical properties of computing. Examples include timing, location, energy, and the awareness of the impact on the physical world. He worked on modeling, simulation, programming models, protocol designs, resource management & control, and novel applications. His recent projects range from sensor networks, mobile computing, and data centers.

Feng Zhao received his PhD in Electrical Engineering and Computer Science from MIT. He is now an Assistant Managing Director at Microsoft Research Asia. Prior to Microsoft Research Asia, he was a Principal Researcher at Microsoft Research Redmond, where he founded and managed the Networked Embedded Computing Group. He serves as the founding Editor-In-Chief of ACM Transactions on Sensor Networks, and has authored or co-authored over 100 technical papers and books. His research interest include networked embedded systems such as sensor networks, energy and resource management of distributed systems as in data centers and cloud computing, and mobile devices and mobility. He has also done work on parallel computing, fast N-body algorithms, machine recognition and diagnostics.