Energy-optimal Batching Periods for Asynchronous Multistage Data Processing on Sensor Nodes: Foundations and an mPlatform Case Study *

Qing Cao, Dong Wang, Tarek Abdelzaher
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA

Bodhi Priyantha, Jie Liu, Feng Zhao
Networked Embedded Computing
Microsoft Research
Redmond, WA 98052, USA

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.

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 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. 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

*The work presented in this paper is sponsored in part by Microsoft Research and in part by NSF grants CNS 06-26825, CNS 06-15301, and CNS 09-16028.

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 [20], 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 Section 5. Finally, the paper concludes with Section 6.

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 [21, 16, 1, 25]. Wakeup energy and time delay cost can be substantial in duty-cycled embedded devices. In [3], Benini et. al. 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 [15, 19]. In general, the problem of minimizing energy consumption of dependent tasks under hard real-time constraints is NP hard for heterogeneous multiprocessors. In [2], Baruah considers the task allocation problem on heterogeneous multiprocessor platforms without task precedence constraints nor hardware configurability. In [14], [22], and [24] the objectives are to maximize, respectively, the throughput, the minimal task slack, and task extensibility. An integer linear programming formalism has been proposed in [11] to compute the schedule.

Dataflow programming models have long been used in signal processing and control applications [17, 4]. Recently, static and dynamic dataflow models have been proposed to program sensor networks [9, 10, 23], since they match well with the data streaming abstraction of the application domain. Typical dataflow scheduling optimize for throughput [12, 8], dynamic memory usage [6], or code size [5]. Our task model is also inspired by time-triggered architectures and languages, such as Giotto [13]. 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 [18, 7]. 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.

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.

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, Figure 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.

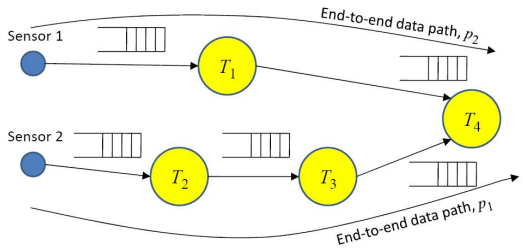


Figure 1. An Example of Two Paths

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:

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

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:

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

where c_i^k is a constant. In theory, one might be tempted to add other terms to Equation (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:

$$E_i^k = a_i^k + b_i^k P_i \quad (3)$$

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[†], the average power W_i^k consumed by processor k on executing task T_i is:

$$W_i^k = \frac{a_i^k}{P_i} + b_i^{k_i} \quad (4)$$

Given each task, T_i, $1 \leq i \leq n$, executing on processor k_i, and the paths p_l, $1 \leq l \leq m$, 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:

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

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) 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. 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:

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

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. First, we formulate the Lagrange function, L, to be minimized, which is defined as:

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

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:

$$P_i^* = \sqrt{\frac{a_i^{k_i}}{2\sum_{p:T_i \in p} \lambda_p}} \quad (8)$$

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

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

```
loop
   ∀i : P_i = sqrt( a_i^{k_i} / (2 Σ_{p:T_i∈p} λ_p) )
   ∀p : λ_p = λ_p + K(Σ_{i:T_i∈p} P_i − D_p/2)
end loop
```

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, ..., 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, Equation (8) reduces to:

$$P_i^* = \sqrt{\frac{a_i}{2\lambda_p}} \tag{10}$$

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

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

Finally, substituting from Equation (11) into Equation (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, ..., 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^*, ..., P_n^* partition $D/2$ proportionally to $\sqrt{a_1}$: ...: $\sqrt{a_n}$.

Proof: The proof follows trivially from Equation (12).

It is interesting to notice that a chain of n tasks, T_1, ..., 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 Equation (4), this means:

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

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

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

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

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, ..., 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 Equation (15) and Equation (16).

4.2 The Star Topology

Consider a scenario where outputs of tasks T_1, ..., 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:

$$P_0^* = \sqrt{\frac{a_0}{2\sum_{1\le j\le n}\lambda_j}} \quad (17)$$

$$P_i^* = \sqrt{\frac{a_i}{2\lambda_i}} \qquad 1 \le i \le n \quad (18)$$

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

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

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 $\sqrt{a_i/(2\lambda_i)}$ is constant or $\lambda_1/a_1 = ... = \lambda_n/a_n$, from which $\lambda_j = \lambda_i a_j/a_i$. Substituting in Equation (19) and solving for λ_i, we get:

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

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

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

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:

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

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 Section 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 Section 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 Equation (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, ..., 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^*, ..., 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 Equation (21) and Equation (22).

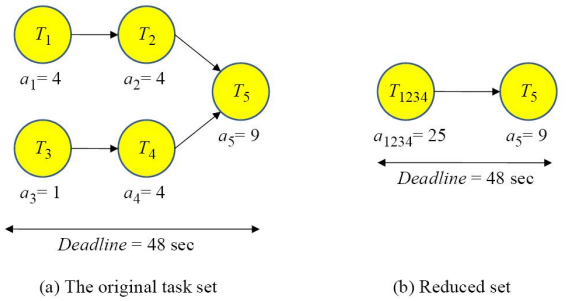
As in the case of the chain reduction theorem, it is now possible to prove the following.

Theorem 4: The Star Reduction: At their optimal batching periods, a set of n periodic tasks, T_1, ..., 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.



(a) The original task set (b) Reduced set

Figure 2. Optimal Period Allocation

Figure 2-a shows a system of five tasks, T_1, ..., 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 aggregator can be reduced by Theorem 4 into an equivalent task T_{1234} with $a_{1234} = 16 + 9 = 25$. This results in Figure 2-b. 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.

4.4 Task to Processor Assignment

The results presented earlier 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 task-to-processor allocation. In general, the number of tasks on a sensor node is usually quite limited (say, 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 choose the allocation that results in the minimum energy solution to the optimal period assignment problem across all allocations. Greedy heuristics can also be trivially constructed, for example, by computing the minimum period for each task at which batching benefits outweigh wakeup overhead, then allocating to the higher-end processor only those tasks whose optimal batching period on that processor is larger than the aforementioned minimum period. As mentioned in the introduction, we do not claim task to processor assignment as a contribution of this paper.

5 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. Finally, we evaluate the energy cost inherent in implementing processing stages as independent, asynchronously executed tasks.

5.1 Energy Profiling

Parameter	MSP	ARM
Frequency	16MHz	60MHz
Active Current (mA)	8.61	75
Active Power (mW)	38.745	337.5
Sleep Current (μA)	17	150
Sleep Power (μW)	76.5	675
Wakeup time (ms)	0.7	3
Wakeup energy (μJ)	7.43	217.4
Flash access energy $(\mu J/byte)$	0.826	1.422
Inter-board Transfer time $(\mu s/byte)$	2	
Inter-board Transfer energy $(\mu J/byte)$	0.65	
Sensing Energy $(\mu J/byte)$	1.64	

Table 1. Energy Profiling Comparison of MSP Board and ARM Board. Board Supply Voltage is 4.5V.

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 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.

OPERATION		Data Type	ARM Time(μs)	ARM Energy (μJ)	MSP Time(μs)	MSP 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**

Table 2. Comparison of Basic Operations on Two Processor Boards Across Different Data Types

5.2 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 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. 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 Section 4. For example, a chain template might 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 Figure 3. 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.

5.3 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 Section 5.4. We use the task model presented in Section 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 Section 5.1 in the evaluation. The power and energy numbers used are from the measurements listed in Table 1.

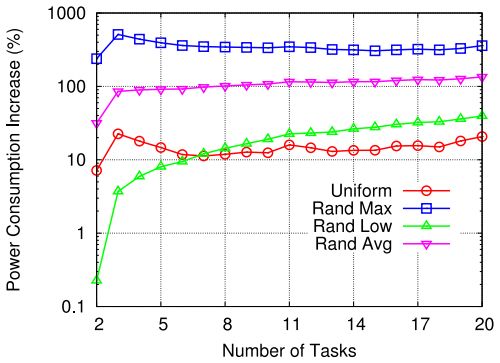


Figure 4. Comparison for Chain Topology on MSP

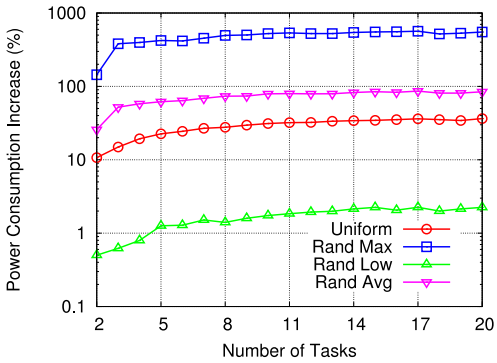
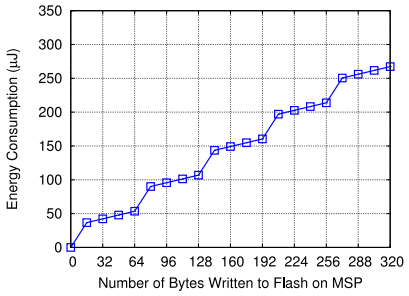
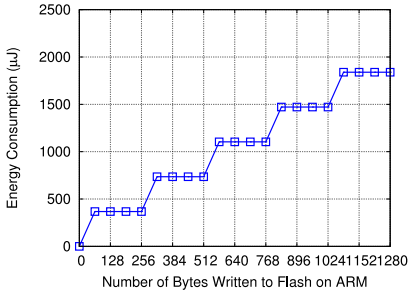


Figure 5. Comparison for Star Topology on MSP



(a) Flash Access Overhead for MSP



(b) Flash Access Overhead for ARM

Figure 3. Flash Access Overhead for MSP and ARM

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 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 4 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 5 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 Figure 6 and Figure 7. We observe that the optimal batching period assignment is better than other approaches in terms of average power consumption.

5.4 Experiments with Optimal Task Assignment

Finally, we compare the performance of assigning tasks to only one of processor boards vs the optimal heterogeneous processor board assignment. Figure 8 and Figure 9 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 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

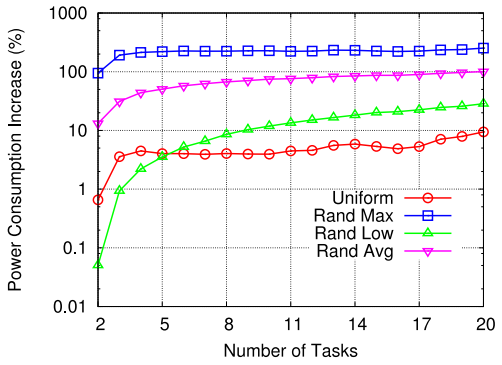


Figure 6. Comparison for Chain Topology on ARM

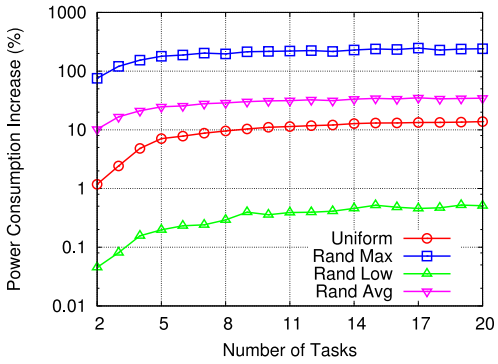


Figure 7. Comparison for Star Topology on ARM

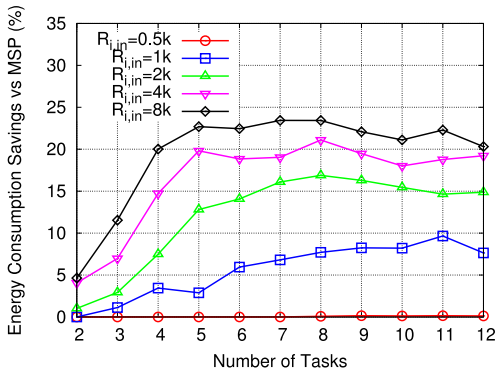


Figure 8. Heterogeneous Assignment versus MSP

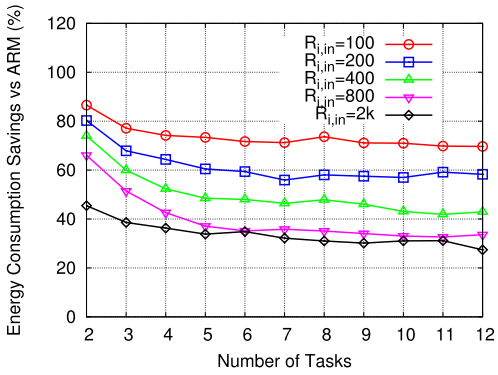


Figure 9. Heterogeneous Assignment versus ARM

batching period to get enough energy savings to overcome the fixed overheads. Therefore, having a higher data rate for such tasks makes the 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.

5.5 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 synchnization 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 optimal period assignment to the case where all processing stages are lumped in one that executes at a period equal to the end-to-end data processing deadline. It is clear that the latter case should use less energy as it allows for more batching.

We run same experiments as in the previous sections for the two approaches. Figure 10 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 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 batch-

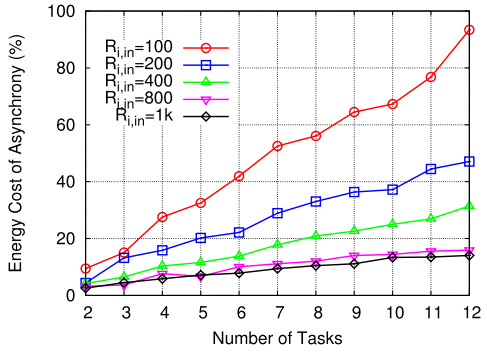


Figure 10. The Cost of Asynchrony

ing 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 higher-end processor can save energy compared to running all on the lower-end processor. This approach is useful for saving energy in sensor applications where sensor data pipelines are manipulated by independently executed periodic stages.

References

[1] A. Acquaviva, L. Benini, and B. Riccò. Processor frequency setting for energy minimization of streaming multimedia application. In *CODES*, pages 249–253, 2001.

[2] S. Baruah. Cost efficient synthesis of real-time systems upon heterogeneous multiprocessor platforms. In *Proc. of 18th International Parallel and Distributed Processing Symposium (IPDPS'04)*, pages 120 – 128, 2004.

[3] L. Benini, A. Bogliolo, and G. De Micheli. A survey of design techniques for system-level dynamic power management. pages 231–248, 2002.

[4] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, Jan 2003.

[5] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. Synthesis of embedded software from synchronous dataflow specifications. *J. VLSI Signal Process. Syst.*, 21(2):151–166, 1999.

[6] J. T. Buck. *Scheduling dynamic dataflow graphs with bounded memory using the token flow model*. PhD thesis, 1993.

[7] L. Cai and Y.-H. Lu. Dynamic power management using data buffers. In *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, pages 526–531, Washington, DC, USA, 2004. IEEE Computer Society.

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

[9] D. Chu, L. Popa, A. Tavakoli, J. M. Hellerstein, P. Levis, S. Shenker, and I. Stoica. The design and implementation of a declarative sensor network system. In *SenSys '07: Proceedings of the 5th international conference on Embedded networked sensor systems*, pages 175–188, New York, NY, USA, 2007. ACM.

[10] L. Girod, K. Jamieson, Y. Mei, R. Newton, S. Rost, A. Thiagarajan, H. Balakrishnan, and S. Madden. Wavescope: a signal-oriented data stream management system. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 421–422, New York, NY, USA, 2006. ACM.

[11] M. Goraczko, J. Liu, D. Lymberopoulos, S. Matic, B. Priyantha, and F. Zhao. Energy-optimal software partitioning in heterogeneous multiprocessor embedded systems. In *DAC*, pages 191–196, 2008.

[12] S. Ha and E. A. Lee. Compile-time scheduling of dynamic constructs in dataflow program graphs. *IEEE Trans. Comput.*, 46(7):768–778, 1997.

[13] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: A time-triggered language for embedded programming. In *EMSOFT*, pages 166–184, 2001.

[14] Y. Jin, N. Satish, K. Ravindran, and K. Keutzer. An automated exploration framework for fpga-based soft multiprocessor systems. In *CODES+ISSS*, pages 273–278. ACM Press, 2005.

[15] A. Khemka and R. K. Shyamasundar. An optimal multiprocessor real-time scheduling algorithm. *Journal of Parallel and Distributed Computing*, 43(1):37–45, 1997.

[16] C. M. Krishna and Y.-H. Lee. Voltage-clock-scaling adaptive scheduling techniques for low power in hard real-time systems. In *RTAS*, pages 156–165, 2000.

[17] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Comput.*, 36(1):24–35, 1987.

[18] Y.-H. Lu, L. Benini, and G. D. Micheli. Dynamic frequency scaling with buffer insertion for mixed workloads. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 21(11):1284–1305, 2002.

[19] J. Luo and N. K. Jha. Power-conscious joint scheduling of periodic task graphs and aperiodic tasks in distributed real-time embedded systems. In *ICCAD*, pages 357–364. IEEE Press, 2000.

[20] D. Lymberopoulos, B. Priyantha, and F. Zhao. mplatform: A reconfigurable architecture and efficient data sharing mechanism for modular sensor nodes. In *IPSN '07*, 2007.

[21] Y. Shin, K. Choi, and T. Sakurai. Power optimization of Real-Time embedded systems on variable speed processors. In *CAD*, pages 365–368, 2000.

[22] T. Sivanthi and U. Killat. Global scheduling of periodic tasks in a decentralized real-time control system. In *IEEE IWFCS*. IEEE Press, 2004.

[23] K. Whitehouse, F. Zhao, and J. Liu. Semantic streams: A framework for composable semantic interpretation of sensor data. In *EWSN*, pages 5–20, 2006.

[24] W. Zheng, J. Chong, C. Pinello, S. Kanajan, and A. L. Sangiovanni-Vincentelli. Extensible and scalable time triggered scheduling. In *ACSD*, pages 132–141. IEEE Computer Society, 2005.

[25] L. Zhong and H. Jha. Dynamic power optimization of interactive systems. In *17th International Conference on VLSI Design*, pages 1041–1047, 2004.