Energy-efficient Multiple Producer-Consumer

Ramy Medhat, Borzoo Bonakdarpour, and Sebastian Fischmeister

Abstract—

Hardware energy efficiency has been one of the prominent objectives of system design in the last two decades. However, with the recent explosion in mobile computing and the increasing demand for green data centers, software energy efficiency has also risen to be an equally important factor. The majority of classic concurrency control algorithms were designed in an era when energy efficiency was not an important dimension in algorithm design. Concurrency control algorithms are applied to solve a wide range of problems from kernel-level primitives in operating systems to networking devices and web services. These primitives and services are constantly and heavily invoked in any computing system and by a larger scale in networking devices and data centers. Thus, even a small change in their energy spectrum can make a huge impact on overall energy consumption for long periods of time.

This paper focuses on the classic *producer-consumer* problem. First, we study the energy profile of a set of existing producer-consumer algorithms. In particular, we present evidence that although these algorithms share the same functional goals, their behavior with respect to energy consumption are drastically different. Then, we present a dynamic algorithm for the multiple producer-consumer problem, where consumers in a *multicore* system use learning mechanisms to predict the rate of production, and effectively utilize this prediction to attempt to *latch onto* previously scheduled CPU wake-ups. Such group latching increases the idle time between consumer activations resulting in more CPU idle time and, hence, lower average CPU frequency. This in turn reduces energy consumption. We enable consumers to dynamically reserve more pre-allocated memory in cases where the production rate is too high. Consumers may compete for the extra space and dynamically release it when it is no longer needed. Our experiments show that our algorithm provides a 38% decrease in energy consumption compared to a mainstream semaphore-based producer-consumer implementation when running 10 parallel consumers. We validate the effectiveness of our algorithm with a set of thorough experiments on varying parameters of scalability. Finally, we present our recommendations on when our algorithm is most beneficial.

Index Terms—Concurrency control; Green computing; Power; Energy; Synchronization

1 Introduction

Designing low-power computing system architectures has been an active area of research in the recent years, partly due to the increasing cost of energy as well as the high demand on producing and manufacturing environment-friendly devices. While the former is an explicit financial cost-benefit issue, the latter is attributed to green computing. However, with the recent explosion in mobile computing and incredible popularity of smart-phones and tablet computers, energy efficiency in software products has become a prime concern in application design and development and, in fact, as important as energy-optimal hardware chips.

Another area where energy efficiency plays an important role is in large-scale data centers. In fact, energy and cooling are the largest cost of a data center. For example, a facility consisting of 30,000 square feet and consuming 10MW requires an accompanying cooling system that costs from \$2-\$5 million [1], and the yearly cost of running this cooling infrastructure can reach up to \$4-\$8 million [2]. These numbers simply mean that even a small improvement in energy usage leads to significant cost reduction. Hence, we are in pressing need to design and implement energy-efficient hardware and software artifacts to keep up with the increasingly high demand in mobile as well as big-data applications.

- *R. Medhat and S. Fischmeister are with the Department of Electrical and Computer Engineering, University of Waterloo, Canada. E-mail: rmedhat@uwaterloo.ca, sfischme@uwaterloo.ca*
- *B. Bonakdarpour is with the Department of Computer Science, Iowa State University, USA. E-mail: borzoo@iastate.edu*

- *A preliminary version of this article appeared at the IPDPS'14 conference.*

Classic algorithms in computer science are heavily used in virtually any computing system ranging from web services and networking devices to device drivers and operating systems kernels. These algorithms were designed in an era when energy efficiency was not an important dimension in algorithm design. For example, Dijkstra's shortest path algorithm fails in the context of energy-optimal routing problems, as simply evaluating edge costs as energy values does not work [3]. Thus, we argue that many of such classic algorithms need to be re-visited and re-designed, so that on top of their functional requirements, energy constraints are treated as a first-class objective as well. Some of these algorithms are applied in such a high capacity that even small improvements in their energy consumption behavior may have a huge impact in the energy profile of large-scale systems and mobile devices in long periods of time.

Producer-consumer is a classic problem in concurrent computing, where two processes, the *producer* and the *consumer* share a common bounded-size memory buffer as a queue. The multiple producer-consumer problem extends this setup by having multiple producers and consumers using a single buffer. In this paper, we study the more general case by supporting multiple buffers with the condition that any producer/consumer writes to/reads from a single buffer. This setup allows us to capture systems, where multiple services share energy resources yet receive data at different rates. Examples include:

- *Operating systems primitives.* Such primitives provide developers with high-level system calls to read and consume data received from I/O devices, e.g., in device drivers. In this setting, each device is a producer, providing data to its respective consumer, i.e. the user application.

- *Web servers.* HTTP requests produced by web browsers are stored in separate buffers per web application, that are consumed and processed by threads in each application's thread pool.
- *Runtime monitoring.* In runtime monitoring, different events produced by the environment or internal system processes are consumed and processed by multiple runtime monitors with separate buffers.
- *Networking.* In most networking devices (e.g., routers), data packets received from the network need to be removed and processed from internal buffers of the device and routed to different destinations. The internal buffers separate subnets, Virtual LANs, or QoS levels.

In this paper, we propose a novel energy-efficient algorithm for the multiple producer-consumer pair problem for multicore systems, where each consumer is associated with one and only one producer. To the best of our knowledge, this is the first instance of such an algorithm. To better understand the vital contributing factors to the energy consumption behavior of the problem, we first analyze the energy profile of existing popular implementations of the producer-consumer problem: a yield-based algorithm, two algorithms based on synchronization data structures (i.e., semaphores and mutexes), and two algorithms that employ batch processing. We observed that these implementations have drastically different behavior with respect to energy consumption. While the yield algorithm is the worst in energy efficiency due to high CPU utilization, the algorithms in which the consumer processes data items in batches are the most energy-efficient due to a lower average CPU frequency. In particular, batch processing results in an average of 75% reduction in energy compared to yield and 32% compared to the semaphore-based implementations. This is validated by a strong positive correlation between average CPU frequency and energy consumption. Such a dramatic shift in energy profile clearly motivates the need for designing an energy-aware solution for the producer-consumer problem.

In our setup, we refer to the group of producers/consumers writing to/reading from a single buffer as a producer consumer *island*. We abstract each island as a single producer-consumer pair associated with a single buffer. This abstraction helps simplifying our online analysis and reduction of energy consumption without loss of generality. Using this abstraction, we design an algorithm that exploits bounded-time dynamic batch processing. It interprets time as a track with periodic slots. To increase the idle time between wakeups, it dynamically constitutes track slots, so consumers can latch on and exploit a CPU wakeup in groups. Given a set of cores, since each core may host a set of consumers, a *core manager* component targets aligning consumers to the slots in that core's track. The core manager is responsible for managing the slot allocations on the track of its respective core. Consumers are designed so that they can dynamically predict production rate of data items to compute and request appropriate latching time. Furthermore, consumers may lend each other buffer space, so that a consumer dealing with a producer with high production rate can continue latching on other consumers and not cause new wakeups.

Our approach can adapt to systems that already employ dynamic voltage and frequency scaling (DVFS). In particular, our design:

1) can work with state-of-the-art dynamic voltage and frequency scaling (DVFS) schemes by leveraging their energy savings which focus on optimum choices of CPU frequency. Dynamic batch processing benefits from such frequency manipulations such that a batch consumption job consumes minimum energy.
2) reduces energy savings further by reducing the number of CPU wakeups. Thus, a lower number of wakeups coupled with efficient CPU frequency manipulations results in an improved energy consumption.

To demonstrate the effectiveness of our approach, we conduct our experiments over a wide range of varying parameters and explore the strengths and weaknesses of using dynamic batch processing. This allows us to provide a recommendation as to what systems are better suited to employ dynamic batch processing. We argue that our dynamic batch processing approach is particularly beneficial in two application areas:

1) According to a Google study [4], web servers are rarely completely idle and seldom operate near their maximum utilization, instead operating most of the time at between 10 and 50 percent of their maximum utilization levels. Moreover, the CPU contributes to more than 50% of Google server power consumption. In such servers, our approach results in longer CPU idle periods, which saves a great deal of energy. This energy saving comes at a cost in terms of response time, which is significantly increased when using batch processing. However, our approach provides controls with which a user can tune the energy saving versus response time.
2) Data-processing-intensive applications are becoming more widespread every day. In such applications, throughput is more important than individual item response time. Our experiments show that our approach results in higher throughput while saving a considerable amount of energy. This makes dynamic batch processing significantly beneficial for applications that can fit in a producer-consumer model.

We validate our claims by conducting thorough experiments using a synthetic queuing-theoretic arrival pattern, as well as a data set of a web server incoming HTTP requests log [5]. Our results show that our algorithm can reduce energy consumption by 38% as compared to a semaphore-based implementation of producer-consumer when running 10 parallel consumers. In comparison to a simple batch processing approach, our algorithm provides up to a 16% improvement in energy consumption. We experiment with varying buffer sizes, number of parallel consumers, as well as data item inter-arrival period. The objective of these experiments is to study the scalability of the proposed algorithm and identify its strengths and weaknesses. To that end, we present our results and recommendations on when it is most appropriate to use our dynamic batch processing approach. Our experimental results demonstrate the tradeoffs between response time and energy savings. The results suggest that the selection of consumer implementation should depend on the required maximum response time and the expected mean inter-arrival period.

Organization. The rest of the paper is organized as follows. In Section 2, we describe the background concepts on CPU power states. Section 3 presents our findings on energy profile of various implementations of the producer-consumer problem. We formally state the energy optimization objective for the multiple producer-consumer problem in Section 4. Our energy-efficient solution to multiple producer-consumer is described in Section 5, while Section 6 analyzes the results of experiments. Related work is discussed in Section 7. Finally, we make concluding remarks and discuss future work in Section 8.

2 Background

Power management technologies approach energy/power efficiency from different perspectives:

- *Static power management* (SPM) simplifies the power management problem by providing support for low-power modes at the hardware level. A system can statically transition to the low-power modes on demand. An example of this is a cell phone going into idle mode when it is locked, or a sensor periodically sleeping at a predefined period.
- *Dynamic power management* (DPM) employs dynamic techniques at run time that determine which power state the system should be in. DPM uses different techniques to infer whether or not a transition to a more efficient state is worthwhile, and, which efficient state to transition to.

In DPM, hardware with scalable power consumption is combined with management software to achieve improved efficiency. Hardware support comes in multiple flavors, e.g., Dynamic Voltage Scaling (DVS) and Dynamic Frequency Scaling (DFS). DVS scales the voltage at which the CPU operates, and thus controls its energy consumption. This is based on the basic Watt's law:

$$P = V \cdot I$$

DVS is becoming more prominent in disk drives, memory banks, and network cards [6]. DFS scales the frequency at which the CPU operates, such that when a high demand occurs, the frequency is raised to meet that demand (of course, at a higher energy cost). When the CPU utilization drops, so does the operating frequency, causing a decrease in energy/power consumption. This is because dynamic power is calculated by

$$P_d = C \cdot V^2 \cdot f$$

where C is the capacitance switched per cycle, V is the voltage, and f is the current CPU frequency. DVS and DFS are often combined into DVFS, where both techniques are used to scale CPU power consumption. CPUs generally support a predefined set of frequency/voltage combinations

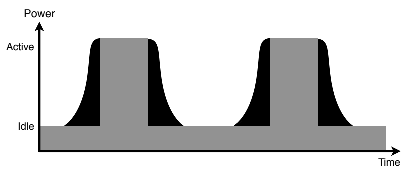


Fig. 1: Overhead due to waking up and idling the CPU. If both peaks are grouped, wakeup overhead becomes lower, and idle time becomes longer, increasing probability that the DPM has a chance to demote the CPU C-state.

performance states, known as P-states. These states define the performance of the CPU in terms of power and throughput.

A relatively different approach to power saving is utilizing CPU C-states. C-states are modes at which the CPU operates, differing mainly in their power consumption. This is achieved by turning off parts of the CPU that are needlessly consuming energy. This may include gradually turning off internal CPU clocks, cache, the bus interface, and even decreasing the CPU voltage (DVS). C-states generally start at C0 which indicates the CPU is fully active, and gradually increases the number (C1, C2, ...) until the idle state or in some cases the hibernate state.

Race-to-Idle is a well-known energy saving concept based on the premise that it is more energy efficient to execute the task at hand faster (a higher P-state) and then go to idle mode (i.e., a deeper C-state), versus running the task at a lower speed resulting in less idle time. Race-to-idle is based on the fact that idle power is significantly less than active power even at a low frequency. Furthermore, recent CPU chipsets such as the Intel Haswell are even more optimized to save a significant amount of power in the idle mode [7]. This indicates that hardware manufacturers are moving towards approaches that attempt to increase CPU residency in deeper C-states. This is especially useful in the context of *core parking*, where specific cores are put in a deep sleep state, reducing their energy consumption significantly.

Although race-to-idle is a valid approach, in the context of the producer-consumer problem, where items are arriving at a certain rate and not in bulk, it may not be the most appropriate strategy. If the items are arriving in such a way that denies the CPU any actual idle time, voltage scaling will detect that there is a consistent load on the CPU and, thus, increase its frequency. Effectively, the CPU attempts to race-to-idle but never gets a chance to become idle, and this results in huge energy waste, due to "racing" most of the time without any chance to "rest". This is in addition to the energy wasted due to idling and waking up frequently. In other words, a certain delay must occur in order for idle mode to be advantageous. Fig. 1 illustrates that more contiguous idle time is more efficient. Thus, a valid energy saving strategy is to increase the contiguousness of idle periods. This approach should be combined with race-to-idle to ensure that a power management strategy targets more idle time with minimum wasted power due to idle-active and active-idle transitions.

3 Producer-Consumer Energy Profile

In this section, we present experimental evidence showing that different implementations of a widely used concurrency control algorithm exhibit drastically different energy consumption profiles.

3.1 Producer-Consumer Implementations

The *producer-consumer* problem is a classic multi-process synchronization problem, where a *producer* process produces data items and places them in a memory buffer, and, a *consumer* process consumes the items from the same memory buffer. Since these processes work concurrently, they need to synchronize to prevent deadlocks and race conditions. Most of the implementations we study in this section rely on the use of a circular buffer. The advantage of a circular buffer is that reading from it and writing into it involves two different pointers, and, thus, alleviates the need to put a single counter in a critical section to avoid concurrency issues.

We study the following implementations:

- **Yield.** This implementation uses sched_yield within a spinlock to yield the CPU if the buffer is full or empty.
- **Mutexes and conditional variables (Mutex).** This implementation uses pthread mutex to ensure mutually exclusive concurrent access to a buffer. We use conditional variables to signal when data is available for the consumer and when space is available for the producer.
- **Semaphores (Sem).** This implementation uses two semaphores for synchronizing the emptiness and fullness of the buffer.
- **Batch processing (BP).** This implementation is similar to the semaphores implementation, except that the consumer waits until the buffer is full and then processes all items in the buffer in one batch.
- **Periodic batch processing (PBP).** This implementation is similar to the batch processing implementation, except that the consumer processes the batch within fixed time intervals (using the usleep() system call) instead of whenever the buffer gets filled. The period for this experiment is $1ms$.

3.2 Experimental Settings

We study the energy consumption of the different producer-consumer implementations using two methods: PowerTop[1] and RAPL (Running Average Power Limit)[2].

PowerTop is a popular Linux-based tool that uses CPU performance counters to estimate the power consumption of all running processes in the system. We use PowerTop to measure the number of wakeups per second that a process causes, and the percentage of CPU usage that the process consumes. The unit for CPU usage in PowerTop is microseconds per second, meaning the number of microseconds the process spends executing every second.

RAPL is an interface that provides monitoring and controlling power consumption of specific Intel CPUs. Starting from the second generation Core i7 processor (Sandy Bridge), RAPL can be used to monitor energy consumption by reading machine specific registers (MSRs). We use the Linux RAPL driver over PAPI to measure energy consumption of each experiment. Our test machine uses an Intel Core i7 Sandy Bridge processor.

Finally, each implementation of producer-consumer is tested using two datasets. First, we test using a synthetic dataset based on an $M/M/1/\mathbb{B}$ queue [8]. This denotes that the production and processing times are drawn from exponential distributions (M/M), there is 1 consumer (1 consumer for every producer), and the items are buffered in a buffer of size $\mathbb{B}$. Second, we test using a real-life dataset based on web server incoming HTTP requests log [5]. We use a portion of the log that covers web requests received over 40 days. Each experiment constitutes running 5 producers/consumers in parallel until the full dataset items are consumed. In the case of the web server data, the log is divided equally such that each producer simulates 8 days of logs. All producers run on one core, and all consumers run on another core. Since cores can be idled separately, this allows us to demonstrate the impact of consumers synchronization on energy consumption regardless of the behavior of producers, which is not in our control. We execute 50 replicates of each experiment to ensure 95% statistical confidence intervals. We measure five metrics in each experiment:

- **Energy (joules).** The number of joules consumed by the system when the respective implementation is executed. This refers to the energy consumption of the CPU package.
- **Wakeups/s.** The number of CPU wakeups per second due to the respective implementations.
- **Usage (ms/s).** The number of milliseconds out of every second that the CPU spends executing the respective implementations.
- **Core idle percentage.** The percentage of idle time that each CPU core spends during the respective implementations.
- **Core frequency percentage.** The percentage of time the CPU spends operating at each possible frequency while executing the respective implementations.

3.3 Experimental Results

3.3.1 Sanity Checks

We perform the following set of sanity checks to ensure our experimental setup is valid:

- We execute a test with a busy-waiting multithreaded program running on two cores of the processor, and we ensure that no experiment reaches the energy consumption found in that implementation.
- We execute a test where no background processes are running except kernel tasks, and we measure energy. We ensure that the energy consumed in this experiment is less than any other experiment we run.
- We measure the statistical confidence interval to ensure that our conclusions are not based on outliers.

3.3.2 Energy consumption of producer-consumer implementations

Figure 2 shows the energy consumption of each of the five producer-consumer implementations. The energy results are

1. https://01.org/powertop/
2. https://01.org/blogs/tlcounts/2014/running-average-power-limit--rapl

highly consistent, as apparent in the small error bars. The Yield implementation consumes the most energy since the CPU is mostly active during its execution. On average, both CPU cores are idle only 1% of the time when executing Yield. Mutex and Sem behave similarly, using 50% of the energy consumed by Yield. BP reduces energy consumption further down to 35% of Yield, and 68% of Sem. Finally, PBP is consistently the most energy efficient implementation consuming 32% of Yield and 62% of Sem. This indicates that implementations based on batch-processing improve upon the most widely used implementation today significantly.

Note that the results in Fig. 2 are based on five pairs of producer-consumers. Each pair of a producer-consumer is an abstraction of the multiple producer consumer problem, which we refer to as a producer-consumer island. To verify our abstraction, we ran the same experiments, where each island has one producer and two consumers sharing the same buffer. This results in a total of five producers and 10 consumers. Fig. 3 illustrates the energy consumption of Mutex, Sem, BP, and PBP. The same trend in Fig. 2 is repeated, albeit with higher energy consumption due to the higher contention of 10 consumers versus only five in Fig. 2. This confirms our intuition that the abstraction only hides elevated contention on a buffer, yet the interactions of independent producer-consumer islands is still affected by the chosen implementation. Note that the number of producers should generally have little impact on prediction if the items coming into the buffer follow the arrival distribution of an $M/M/1$ process.

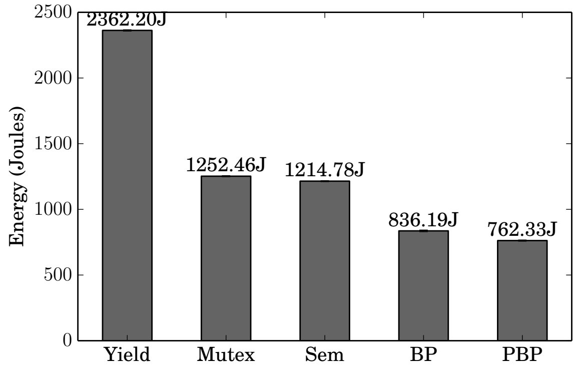


Fig. 2: Energy consumption for all five implementations when running the $M/M/1/\mathbb{B}$ dataset.

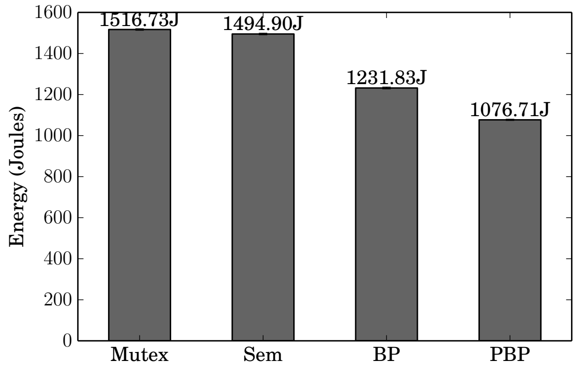


Fig. 3: Energy consumption for all implementations excluding yield when running the $M/M/1/\mathbb{B}$ dataset using 5 sets of a producer and two consumers.

3.3.3 *Understanding root causes of energy profile*

Next, we study the CPU usage and the number of wakeups in each implementation (see Fig. 4). Yield has the lowest number of wakeups which is expected since the CPU is active most of the time. However, the CPU usage of Yield is significantly less than all other implementations. This is counter-intuitive given the high energy consumption of Yield. Upon studying the percentage of time spent in each available CPU frequency, Yield spends on average 99% of the time at the highest CPU frequency of 3.4GHz. Intel SpeedStep dynamic voltage scaling recognizes that the CPU is active most of the time and upgrades its frequency to the maximum.

The number of wakeups of Mutex and Sem is similar, as is their energy consumption. BP has a significantly high number of wakeups, which can be explained by the consumers waiting for the buffer to be filled, resulting in a longer idle period which is an opportunity the DPM uses to put the processor to sleep. PBP reaches a compromise by using a periodic activation of consumers, which reduces the number of wakeups by approximately 50%, since there are shorter idle periods and hence less opportunity to put the processor to sleep. This decrease in the number of wakeups results in an increase in usage since the CPU has no chance to go to sleep too often.

The key to understanding the energy consumption of the different implementations is the distribution of time spent in different CPU frequencies. As mentioned earlier, Yield spends 99% of its time at the maximum CPU frequency. Fig. 5 shows the percentage of the time spent in different CPU frequencies of the remaining four implementations. We omitted Yield from Fig. 5 to make it more readable for the remaining implementations. Notice that Mutex and Sem have low percentages when residing in lower frequencies, and a spike at 3GHz. This explains their relatively high energy consumption compared to the batch-based implementations. BP on the other hand spends most of the time at the lowest CPU frequency of 800MHz (approx. 18%). PBP spends even less time at frequencies higher than 800MHz.

These results are interesting since the initial assumption is that a batch-based implementation will have a lower energy consumption due to the reduced number of wakeups. However, results show that batch-based behavior actually leverages DVFS effectively to increase the percentage of time the CPU operates at lower frequencies. A simple DPM scheme demotes the CPU C-state gradually by checking CPU activity periodically. Implementations such as Mutex and Sem activate the CPU too frequently and, thus, the DPM has no chance to demote the C-state. On the other hand, Intel SpeedStep increases CPU frequency gradually to match CPU activity, which causes it to operate mostly at high frequencies in these implementations.

Batch-based implementations such as BP and PBP give DPM time to demote the CPU C-states. This idling of the CPU prevents Intel SpeedStep from promoting the CPU frequency, and thus most of the time is spent at the lowest

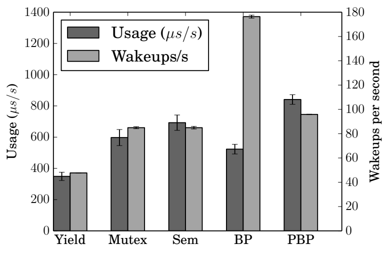


Fig. 4: Wakeups/s versus usage $\mu s/s$ for all five implementations when running the $M/M/1/\mathbb{B}$ dataset.

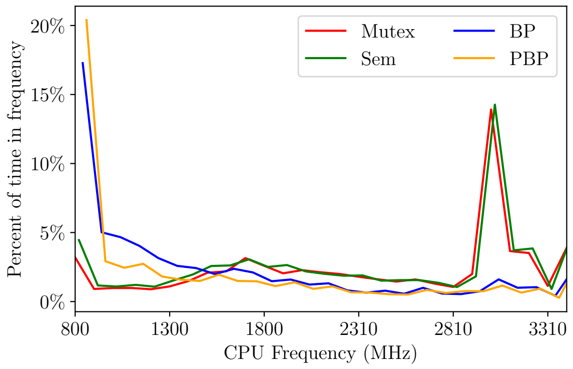


Fig. 5: Percentage of CPU time spent operating at different frequencies for Mutex, Sem, BP, and PBP when running the $M/M/1/\mathbb{B}$ dataset.

CPU frequency. PBP has a lower energy consumption versus BP due to the following reasons:

- CPU spends less time in higher frequencies compared to BP (see Fig. 5). This is because BP only wakes up to a completely full buffer, which increases the processing load resulting in the CPU climbing up to higher frequencies. However, the periodicity of PBP prevents this from happening, since it frequently activates the consumers to consume a small number of items, and the consumers are then shortly idle again. This prevents the CPU from climbing up to higher frequencies.
- The number of wakeups of PBP is less than BP.
- Due to periodically consuming items, PBP finishes processing the dataset slightly faster.

These conclusions are further supported by analyzing the correlation between the average CPU frequency and energy/power consumption. This correlation is calculated across all 250 experiments, and checked for Pearson correlation assumptions. The result is a 0.70 correlation between average CPU frequency (weighted average) and both energy and power and a -0.71 correlation between the number of wakeups per second and energy/power. We explore this further by studying the correlation between energy consumption and the percentage of the time spent at frequencies above 3GHz, which has a higher value of 0.86, indicating the significance of residency at higher frequencies on energy/power. Between batch processing implementations, the significant factor is the number of wakeups, which is 0.65 correlated to energy/power. Thus, the large difference in energy consumption between batch implementations and all others is due to the lower average frequency which is caused by the higher number of wakeups, while the small difference between both batch implementations is mainly due to the difference in the number of wakeups, since these implementations don't differ significantly in CPU frequency distribution.

Batch processing has its drawbacks, mainly the latency in response time. Mutex and Sem implementations have much lower latency. However, when energy efficiency is a main concern, a batch-based implementation with a bounded latency can provide an energy-efficient solution. We study this in further details in the upcoming sections.

In summary, the lesson learned from this simple study of a fundamental problem is the following:

The reduction level in energy consumption observed in our experiments justifies re-visiting the design and implementation of classic algorithms to make them energy efficient.

4 Formal Problem Description

In this section we formally present the optimization problem for reducing energy consumption in multiple producer-consumer pairs.

4.1 Overall Framework

As explained in Section 3, the number of wakeups and the average CPU frequency have a significant effect on energy consumption. Batch processing implementations leverage longer periods of idle time in increasing the number of CPU wakeups. This results in the CPU waking up to lower frequencies instead of promoting its frequency due to continuous load and lack of idle time. Thus, a logical objective is to force the CPU to sleep longer between wakeups. However, such an approach requires incorporating the DPM and DVFS mechanisms in the underlying system to determine the best period to sleep in order to force the CPU to go into a deeper idle state and consequently reduce average CPU frequency.

In this paper, we attempt to formulate the problem in a way that is independent of the underlying DPM and DVFS mechanism implemented in the system. This means that the DPM and DVFS are not in our direct control, and we can only indirectly manipulate them from the application level. Thus, we have no knowledge of when the CPU physically goes to sleep (C1 and above) and when it wakes up (C0). We can only control when the consumers start doing work on the CPU, which we refer to as *activating* the CPU. Thus, we assume a simple power model where there are only two CPU states: active and idle. In our abstraction, a CPU is *activated* when it transitions from idle to active. The CPU immediately goes to idle when a sleep or lock is called. For instance, a producer will sleep after producing an item, and activate the CPU to produce the next item. A semaphore based consumer will activate the CPU whenever a ticket is available to consume, and will sleep while there are no tickets available in the semaphore. Thus, the objective becomes *reducing* the number of *activations*. This implies longer idle periods between activations. If we reapply this back to our actual system, longer idle periods results in a higher chance that the CPU physically gets demoted to idle, as opposed to shorter idle periods where the CPU has no chance to go to sleep. This results in a higher number of physical wakeups/sec and a lower average CPU frequency.

4.2 System Assumptions

We begin by stating the assumptions on which the problem is based:

- *Multicore system.* The system we attempt to optimize for energy is a multicore system, which supports core parking.
- *Abstraction of DVFS and DPM.* For generality, we abstract the operations of DVFS and DPM in managing CPU frequency and idle state respectively. That is, we have no direct way to control DVFS and DPM.
- *Multiple producer-consumers.* The system hosts a set of producer-consumer pairs, where each consumer has its own buffer.
- *Independent producer rates.* Each producer produces data elements at its own non-linear and non-constant rate, independent of other producers.
- *Maximum response time.* Each consumer defines the maximum time allowed for a data item to be buffered and not processed. Any data item must be processed before or at the maximum response time.
- *Consumer isolation.* Consumers are isolated on dedicated cores. This assumption is to isolate the effect of background processes that can potentially wakeup the core on which the consumer is executing. We also assume producers are either processes on separate cores or external events, such that they do not interfere with the consumers and their dedicated cores.

4.3 The Optimization Problem

We now formalize the multiple producer-consumer energy efficiency problem for a multicore system. Since we assume cores support parking, and can be idled and frequency scaled separately, we will focus the rest of the problem on one core and extend trivially to all cores of the system.

A set of producers $P = \{p_1, p_2, \ldots, p_M\}$ produce data items at their independent varying rates. Each producer p_i, $1 \leq i \leq M$, produces the data items $\mathcal{D}_i = d_{i,1}, d_{i,2}, \ldots, d_{i,N_i}$, where N_i is the total number of items produced by producer p_i. Each data item is a tuple defined as follows:

$$d_{i,j} = \langle \upsilon_{i,j}, \pi_{i,j} \rangle$$

where $\upsilon_{i,j}$ is the time stamp at which producer p_i produces data item $d_{i,j}$, and $\pi_{i,j}$ is the computational load of consuming $d_{i,j}$ in terms of CPU cycles.

We assume pairs of producers and consumers. Let there be a set $C = \{c_1, c_2, \ldots, c_M\}$ of consumers running on a single core. In case the system has multiple cores, we assume that each core serves a fixed set of consumers (i.e, consumers do not change core). Each consumer c_i, $1 \leq i \leq M$, consumes items produced by producer p_i. For each consumer c_i, let the activation times of the consumer be the total ordered set $\mathcal{T}_i = \{t_{i,1}, t_{i,2}, \ldots, t_{i,k_i}\}$, where k_i is the number of activations of the consumer. The value of k_i depends on the time of the activations and the number of data items buffered in between. If the consumer does not employ batch processing (such as Sem), then $k_i = N_i$, meaning the consumer will be activated every time it receives a data item. If it uses simple batch processing and has a buffer of size 10, then $k_i = \lceil N_i/10 \rceil$.

We now define four functions that help define the problem:

- **Activation times.** Let $\delta_i(t)$ be a function that returns the last activation of consumer i before time t:

$$\delta_i(t) = \max\{t_{i,j} \mid t_{i,j} < t\}$$

- **Buffered items.** We define function $\gamma_i(t)$ to denote the buffered data items at time t (items that have not been consumed yet):

$$\gamma_i(t) = \{d_{i,j} \mid \delta_i(t) < \upsilon_{i,j} \leq t\}$$

  Thus, if the buffer size is $\mathbb{B}$, then the following condition must hold to prevent buffer overruns:

$$\forall i : \forall j : |\gamma_i(t_{i,j})| \leq \mathbb{B} \quad (1)$$

- **Processed items.** Let $\beta(d_{i,j})$ be a function that returns the time at which data item $d_{i,j}$ is processed by consumer i:

$$\beta(d_{i,j}) = \min\{t_{i,j} \mid t_{i,j} \geq \upsilon_{i,j}\}$$

  Thus, the following condition must hold to maintain a maximum response time $\mathbb{M}$:

$$\forall i : \forall j : \beta(d_{i,j}) - \upsilon_{i,j} \leq \mathbb{M} \quad (2)$$

  Let A be the set of all activation times $t_{i,j}$ for every consumer c_i running on a single core.

$$A = \bigcup_{i \in [1,M]} \mathcal{T}_i$$

  We will refer to A as the set of *core activations*. Thus, this set is of the form $A = \{a_1, a_2, \ldots, a_L\}$, where each a_l is a core activation time. Let $\lambda(a_l)$ be a function that returns all the data items that are buffered by all consumers at core activation a_l:

$$\lambda(a_l) = \bigcup_{i \in [1,M]} \gamma_i(a_l)$$

Next, we define functions that help calculate the energy consumption given an abstract DPM and DVFS:

- **Wakeup energy.** Let us define

$$\epsilon_\omega(t_p, f_s) \rightarrow \langle E, f_e \rangle$$

  as a function that calculates the energy consumed by the CPU when no items are being processed for a duration of t_p, given that the CPU is initially operating at frequency f_s. If the DPM decides to idle the CPU core during time t_p, the following costs will apply:
  - The energy cost associated with the power-down of the CPU core (which is initially running at frequency f_s).
  - The energy cost of an idle CPU for a portion of t_p.

– The wakeup energy of the CPU at the end of period t_p.

The function will return the total energy consumed (denoted $\epsilon_\omega(t_p, f_s) \cdot E$) and the frequency of the CPU at the end of t_p (denoted $\epsilon_\omega(t_p, f_s) \cdot f_e$).
If the period is too short to go to idle, and the DPM decides to keep the CPU active, then there is no energy consumed due to idling in period t_p, and the returned energy is that of an active CPU for the duration of t_p.

- **Consumption energy.** Let us define the following function:
$$\epsilon_\vartheta(D, f_s) \rightarrow \langle E, t_c, f_e \rangle$$
where D is a set of data items and f_e is the initial CPU frequency. The function returns a tuple of the energy E consumed to process these items, the time t_c to consume them, and the CPU frequency f_e at the end of consumption. This function represents how DVFS would manage the energy consumption and completion time of a set of data items. We assume the CPU core does not go to idle while consuming items in D, since all items are already buffered and ready to get processed. This assumption isolates DPM from DVFS, such that $\epsilon_\vartheta(D)$ is solely responsible for the energy consumption of the CPU in stretches of time where it is active, without DPM intervention.

Now, we define the functions that calculate the energy consumption between every two consecutive core activations in A. Let us start with the very first activation, i.e., a_1. The CPU is assumed to be idle before a_1, and, hence, we need to calculate the first wakeup energy:

$$\omega(a_1) = \epsilon_\omega(a_1, 0)$$

where $\omega(a_1)$ denotes the tuple of energy consumed to wake the CPU up for the first time (E) and the CPU frequency at a_1 (i.e., f_e). Hence, the first wakeup energy is $\omega(a_1) \cdot E$.

Next, we need to calculate the energy of consuming items buffered at a_1. Let $\vartheta(a_l)$ be defined as follows:

$$\vartheta(a_l) = \epsilon_\vartheta(\lambda(a_l), \omega(a_l) \cdot f_e) \quad (3)$$

where $\vartheta(a_l)$ denotes the tuple of (1) energy consumed to process buffered items at a_l (i.e., E in the case of a_1), (2) the final frequency at the end of consumption (i.e, f_e), and (3) the consumption time (i.e, t_c). Hence, the energy of consuming items buffered at a_1 is $\vartheta(a_1) \cdot E$. For all the subsequent activations, wakeup energy is calculated recursively as follows:

$$\omega(a_l) = \epsilon_\omega(a_l - a_{l-1} - \vartheta(a_{l-1}) \cdot t_c, \vartheta(a_{l-1}) \cdot f_e) \quad (4)$$

Thus, the activation wakeup energy ω of core activation a_l is the energy consumed within the inactive period between a_{l-1} and a_l. This period is calculated as the time between a_{l-1} and a_l minus the time spent consuming the items that were buffered at a_{l-1}.

Finally, our ultimate objective is to minimize the energy consumption of the CPU core given all possible activations A. Our optimization function is defined as follows:

$$\min_A \left\{ \sum_{l=1}^{L} \Big(\vartheta(a_l) \cdot E + \omega(a_l) \cdot E \Big) \right\} \quad (5)$$

where $L = |A|$ was number of activations. Thus, the optimization objective to construct a wakeup pattern for all consumers running in the CPU core such that the set of core activation times A minimizes the total energy consumption.

5 Energy-aware Multiple Producer-Consumer Algorithm

This section presents the design of our algorithm to solve the multiple producer-consumer problem presented in Section 4. The optimization Objective 5 will yield an optimal energy consumption given retroactive knowledge of all arriving items. To design an online algorithm, we incorporate the insight gained in Section 3 with the formulation in Section 4. Roughly speaking, our algorithm interprets time as a *track* with periodic *slots*. Consumers latch onto core activations scheduled at slot boundaries to process their respective buffered items. This results in an increase of the length of idle time between consecutive core activations. Hence, the design attempts to minimize energy by targeting the two components of the objective in Formula 5:

- overall wakeup energy $\sum_{l=1}^{L} \omega(a_l) \cdot E$ is reduced by extending the period of idle time between core activations.
- consumption energy $\sum_{l=1}^{L} \vartheta(a_l) \cdot E$ is reduced due to longer idle periods which result in lower initial frequencies $\omega(a_l) \cdot f_e$.

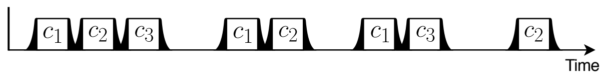
To that end, we introduce a *core manager* component that targets aligning consumers to the slots in that core's track. The core manager is responsible for managing the slot allocations on the track of its respective core. Consumers are designed so that they can predict the production rate of data items to compute an appropriate latching time.

In the rest of this section, we describe our technique and design choices for core managers and consumers in Subsections 5.1, 5.3, and 5.2, respectively.

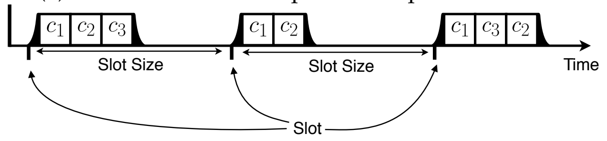
5.1 Contiguous Consumer Activations

As established in Section 3, longer time between consumer activations results in the CPU going into deeper sleep, and hence, running at a lower frequency at the next consumer activation. This results in a lower average CPU frequency, and in turn, lower energy consumption. Recall that in Section 3, we identified batch processing as a more energy-efficient implementation of the producer-consumer problem. Now, consider three consumers: c_1, c_2, and c_3 that are invoked when their respective buffer is full. Since the amount of time it takes to fill the buffer depends upon the rate of the data item production, a possible activation pattern of these three consumers could be as shown in Fig. 6a. The DPM can handle the consumer activations in the figure in one of two ways:

- The DPM can put the CPU to sleep after each consumer finishes consumption, resulting in an energy and time overhead.
- The DPM does not idle the CPU due to the shortness of the inactive period between consumer activations, and DVFS increases the CPU frequency assuming a continuous load, resulting in significantly higher energy consumption.



(a) Uncontrolled wakeups of multiple consumers.



(b) Aligned wakeups with consumer latching.

Fig. 6: Uncontrolled vs. aligned wakeups of 3 consumers A, B, and C.

Our algorithm attempts to group consumer activations dynamically. We begin with interpreting time as a *track* with periodic *slots*. In our case, this is denoted as the *slot size* Δ. The default slot size is equal to the minimum of all maximum acceptable response latencies defined by the producer-consumer pairs. Figure 6b presents this idea. Observe that upon grouping, the number of wakeups is reduced to three, and the idle time between slot activations is longer. This reduction maps to a reduced number of activations L (see Equation 5). This gives DPM a chance to idle the CPU, which in turn wakes up to a lower frequency and so on. This example illustrates the potential impact of grouping on energy consumption. Let the timestamps of the start of these slots be the set $S = \{s_1, s_2, s_3, \ldots \infty\}$, where $s_m - s_{m-1} = \Delta$. The initial objective of the algorithm is to ensure that all core activations are aligned to the slots:

$$\forall l : \; a_l \in S \tag{6}$$

While this constraint may result in a suboptimal core activation pattern, it simplifies the problem in the context of an online algorithm.

5.2 Consumer Design

The consumer is responsible for predicting the rate of items produced by the producer based on the recent past. We use a *moving average estimation* to determine the upcoming rate of items:

$$\hat{r}_{i,j+1} = \frac{\sum_{j'=j-h+1}^{j} r_{i,j'}}{h}$$

where i is the index of the consumer, j is the index of the consumer activation, $r_{i,j'}$ is the rate recorded at activation j', and h is the number of previously recorded rates used by the moving average to estimate the future rate $\hat{r}_{i,j+1}$. Any rate $r_{i,j}$ is calculated as follows for consumer c_i:

$$r_{i,j} = \frac{|\gamma_i\left(t_{i,j-1}, t_{i,j}\right)|}{t_{i,j} - t_{i,j-1}}$$

where γ_i is defined in Equation 1. The reason for selecting the moving average is the simplicity of its calculation, imposing very low overhead on the processing involved, which is a desirable characteristic when attempting to minimize energy consumption.

5.3 Core Manager Design

The fundamental part of the proposed solution is the design of the core manager. The core manager performs the following steps:

1) Upon a scheduled wakeup (a_l), it looks up the registered consumers for the current slot, and activates them. This is achieved by signalling a semaphore. The energy consumed due to this wakeup is $\omega(a_l) \cdot E$ (see Equation 4). The consumers update their predicted rate in their individual data structures and begin processing items normally. The energy consumed in processing of items ($\vartheta(a_l)$) is dependant on the number of buffered items and the DVS mechanism involved (see Equation 3).
2) After activating all consumers, the core manager calculates how to distribute the shared buffer space such that consumers experiencing high load receive extra space to accommodate for the increase. Resizing is performed once every q core manager invocations. This is detailed in Subsection 5.3.1.
3) Next, the core manager reads the predicted rate of each consumer, and according to the new buffer size, makes a reservation to minimize the number of CPU wakeups. This is detailed in Subsection 5.3.2.
4) Finally, the core manager determines the next slot to wake up. Note that this does not necessarily have to be at time $s_m + \Delta$, where s_m is the current slot. The core manager will schedule the next slot with *at least one reservation*, thus, ensuring that the CPU is not activated needlessly. This results in a longer idle period, and thus has a direct impact on reducing wakeup energy and an indirect impact on reducing consumption energy (Formula 5).

It is worth noting that the core manager does not use a significant amount of memory in storing the reservations, since it only needs to maintain the set of reservations in the near future. Past reservations are replaced and future reservations are limited to only the next activation of every consumer.

5.3.1 Dynamic Buffer Resizing

Consider the case where the predicted rate of items is too high to be accommodated within one slot. This implies that a buffer overflow may occur before the closest slot triggers. Such a scenario motivates our idea on implementing a dynamic buffer resizing solution.

We previously introduced $\mathbb{B}$ as the size of the buffer used by each producer-consumer pair. To allow dynamic buffer resizing, we divide the buffer into a guaranteed portion $\mathbb{B}_0$ and a dynamic portion $\mathbb{B}_d$, such that $\mathbb{B} = \mathbb{B}_0 + \mathbb{B}_d$. Thus, the total amount of dynamic buffer space is $\mathbb{B}_d \times M$, where M is the number of consumers.

A consumer can relinquish part of the dynamic portion of its buffer to other consumers that are experiencing a high load. Thus we define $B_{i,m}$ as the actual reserved dynamic buffer size of consumer i at slot m. Thus, at slot m, the total buffer size available for consumer i is $\mathbb{B}_0 + B_{i,m} \leq \mathbb{B}$.

A consumer can be forced to run before its reserved slot to avoid a buffer overflow. This would occur if the producer is attempting to write an element for which there is no space, and the consumer is still sleeping. The consumer is then immediately activated to empty the buffer and provide space for the producer. We refer to this forced wake up as *buffer trigger*.

Dynamic resizing works as follows: after the core manager wakes up the scheduled / forced consumers, it calculates the number of times each consumer was *buffer triggered* during the last q invocations. It then redistributes the dynamic buffer space proportionally according to the number of buffer triggers. Consumers with more buffer triggers receive more dynamic space. Note that the core manager only redistributes space used by the consumers it activated, so as to not interfere with sleeping consumers awaiting activation.

The shared buffer space is implemented as a linked list and requires synchronization since it is being used across consumers. This synchronization is similar to that used in the multiple producer consumer problem with a single shared buffer. While the synchronization impacts writing to the buffer, it only occurs intermittently when a producer generates unpredicted high load.

The core manager blocks each producer-consumer pair while the update is in progress using mutexes. There might be a case where a buffer is shrinking and a producer has already written to an item beyond the newly shrunk size. Consequently, another producer which now has a bigger buffer cannot fully utilize it since there is no space left. This problem will be remedied once the consumers starts pulling items out of the buffer, and will not arise again until the next q^{th} invocation of the core manager, at which new resizing will apply.

5.3.2 Reservation

The process of selecting a slot to reserve is based on minimizing the cost function ρ over the set of possible slots:

$$\rho_i\left(s_{m'}\right) = \frac{\omega\left(s_{m'}\right)\cdot E + \epsilon_{\vartheta}(\hat{r}_{i,j+1}\times\left(s_{m'}-s_m\right), \omega\left(s_{m'}\right)\cdot f_e)\cdot E}{\hat{r}_{i,j+1}\times\left(s_{m'}-s_m\right)} \tag{7}$$

where s'_m is the slot being evaluated and s_m is the current slot. The value of $\hat{r}_{i,j+1}\times\left(s_{m'}-s_m\right)$ is used to calculate the number of data items predicted to have been buffered in slot $s_{m'}$, given that $t_{i,j}=s_m$ (the last activation of the consumer was at slot time s_m). The cost function ρ is normalized to represent the cost per data item. This gives perspective on the tradeoff between latching on a slot with a low predicted number of items versus reserving a new slot with a high predicted number of items.

Recall that the size of the buffer that the consumer reads from is now $\mathbb{B}_0 + B_{i,m}$. Thus, given that the current time (slot) is s_m and the predicted rate is $\hat{r}_{i,j+1}$, the time expected to fill the buffer is

$$s_m + \frac{\mathbb{B}_0 + B_{i,m}}{\hat{r}_{i,j+1}}$$

To provide response time guarantees, the actual time we consider as the time the buffer is filled is the following:

$$t_f = s_m + \min\{\frac{\mathbb{B}_0 + B_{i,m}}{\hat{r}_{i,j+1}}, T_i\}$$

where T_i is the maximum response time allowed for consumer i.

The core manager starts evaluation at the latest slot to occur before t_f and backtracks until it is impossible to find a slot with lower ρ. If a slot has higher ρ than its predecessor, then it is safe to assume that no better slots can be found by further backtracking. Using a helper function in the core manager that backtracks to the next slot *with reservations*, the backtracking process only consumes one iteration and is, hence, a lightweight operation taking constant time and energy.

6 Experimental Results

This section presents the results of the experiments to compare our algorithm with other standard implementations of the multiple producer-consumer problem. The experimental settings are similar to the ones presented in Subsection 3.2, with the addition that the core manager runs on a separate core, resulting in a total of 3 cores used.

6.1 Experimental Parameters

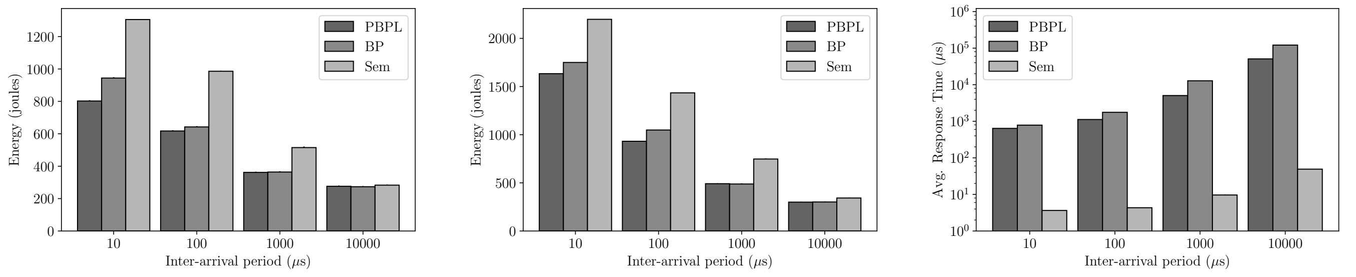
The experiments are based on executing producer-consumer pairs in parallel. We evaluate three implementations: Sem and BP discussed in Section 3, and our proposed algorithm in Section 5, periodic batch processing with latching (PBPL). We chose Sem because it is the most widely used implementation and also the most energy efficient out of common producer-consumer implementations. We chose BP because it is the simplest form of batch processing and is our frame of reference in terms of trivial batch processing implementations. To that end, we run two sets of experiments:

- *$M/M/1/\mathbb{B}$:* Each producer-consumer pair uses an $M/M/1/\mathbb{B}$ process to generate/service requests. We experiment with different arrival rates by varying the λ parameter of the exponential distributions in the $M/M/1/\mathbb{B}$ process in an increasing fashion, such that the mean inter-arrival period ranges from $10\mu s$ to $10ms$. We use a constant service time rate parameter and $\mathbb{B}$ indicates the buffer size we use.
- *Web server log data:* In these experiments the producers use the web server log data set mentioned in Section 3 with different phase shifts to create more variation among production rates of producers.

We experiment with the following parameters:

- *Number of consumers:* We experiment with 10, 20, and 30 producer-consumer pairs.
- *Buffer size:* For the batch processing based implementations (BP and PBPL), we experiment with three different buffer sizes: 25, 50, and 100. Due to space limitations, the next sections report results only from buffer size 25, and we refer to results of the other buffer sizes in Section 6.7.
- *Maximum response time:* For our implementation (PBPL), we experiment with different response time upper bounds, namely: 10μs, 100μs, 1ms, and 10ms.

We run each experiment for a duration of 100 seconds and measure the energy consumed during that period given an infinite supply of data items to consume.



(a) Energy / 10 consumers (b) Energy / 30 consumers (c) Avg. Response time / 10 consumers

Fig. 7: Plots of energy consumption and average response time at various mean inter-arrival periods of the $M/M/1/\mathbb{B}$ queue.

6.2 Experimental Metrics

The following is the set of metrics measured for the executed experiments:

- **Energy.** The energy consumption in joules of the CPU package (all cores).
- **Number of wakeups per second.** The number of wakeups per second measured by PowerTop.
- **Average response time.** The average response time of consuming items, measured from the time the item is produced till the time it is consumed.
- **Number of buffer triggers.** The number of times a consumer is activated because the buffer is full.
- **Throughput.** Consumed items per second.
- **CPU frequency percentage.** The percentage of time spent by the CPU at different frequencies.

6.3 Energy Consumption Results

6.3.1 $M/M/1/\mathbb{B}$ *Process*

Figure 7a shows the energy consumption of the three compared implementations at exponentially increasing mean inter-arrival periods. The results in this figure are based on 10 concurrent consumers running on the same core and sharing a single core manager. As can be seen in the figure, PBPL reduces energy consumption of the popular semaphore-based implementation (Sem) by 38%, and is 16% better than a basic buffered implementation. That is in the case of a short inter-arrival period of 10μs. The advantage decreases as the mean inter-arrival period increases, with PBPL and BP approaching each other at $1ms$ and approaching sem at $10ms$. This is expected since idle power dominates experiments with a large inter-arrival period.

Figure 7b show the energy consumption of the three implementations when 30 consumers are running in parallel. The reduction in energy drops to 26% when 30 consumers run in parallel, which is due to the high contention accompanied by running a larger number of consumers in parallel with a short inter-arrival period.

6.3.2 *Web Server Dataset Input*

We observe similar energy savings when running the producers using the web server dataset that we used for our study in Section 3 (see Fig. 8). PBPL reduces energy consumption by 33%, 42%, and 45% compared to Sem when 10, 20, and 30 consumers are running in parallel, respectively. It also improves upon BP by up to 7.5%.

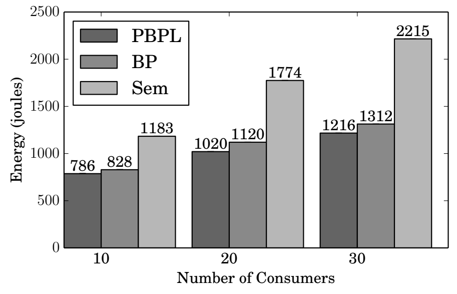


Fig. 8: A plot of energy consumption of all 3 implementations running 10, 20, and 30 concurrent producer-consumer pairs. These experiments are based on the web server dataset.

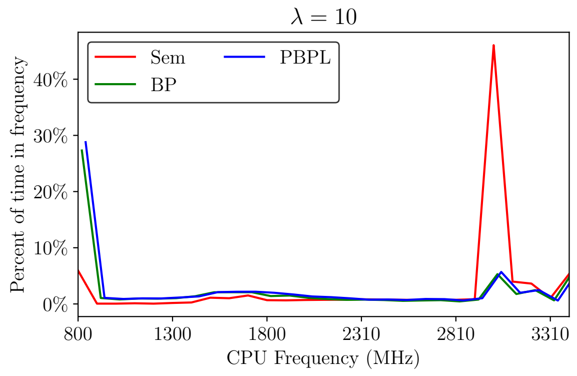


Fig. 9: Percent of CPU time spent at different frequencies for Sem, BP, and PBPL when running the $M/M/1/\mathbb{B}$ dataset.

Energy consumption results from both sets of experiments demonstrate a consistent advantage in using the PBPL implementation. PBPL reduces energy consumption significantly and consistently for systems that experience high loads, whether due to high concurrency or short inter-arrival period.

The explanation for the reduced energy consumption of PBPL and BP is similar to our findings in the study in Section 3. The batch-based invocation of consumers in both implementations results in a lower average CPU frequency, since the CPU has a chance to switch to idle. When the CPU wakes up, it starts operating at the lowest available frequency and gradually increases. This is in contrast to Sem which does not transition to idle that often and, hence, the CPU DVFS mechanism increases the frequency due to the

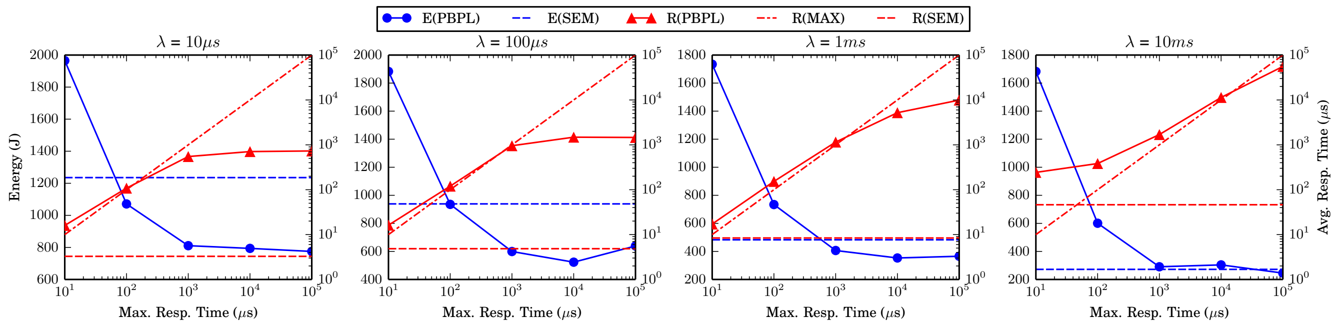


Fig. 10: A plot of the energy and average response time of the three implementations when running 10 concurrent producer-consumer pairs at different latency presets (Max. Resp. Time). The experiments are executed at various mean inter-arrival peri

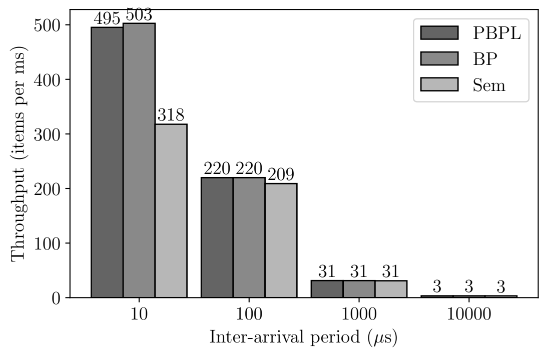


Fig. 11: A plot of the throughput of all 3 implementations running 10 concurrent producer-consumer pairs at various mean inter-arrival periods of the $M/M/1/\mathbb{B}$ queue.

continued arrival of items to process. The overhead of context switching for each element is magnified as processing overhead, resulting in an average frequency of 1.8GHz for Sem versus an average of 1.2GHz for BP.

Figure 9 shows the frequency distribution of all three implementations when running 10 parallel consumers. Sem spends approximately 45% of its time at 3GHz, while BP and PBPL spend approximately 30% of their time at 800MHz, the lowest available frequency. PBPL outperforms BP due to the periodicity of its wakeup pattern. This allows it to mitigate the high loads more uniformly, resulting in a lower average CPU frequency of 1.1GHz. This is visible in Fig. 9 where PBPL spends more time at 800MHz and less time at every higher frequency. Recall that longer residency in lower frequencies means a lower $\vartheta(a_l) \cdot f_e$ for every activation. This results in a lower energy consumption $\vartheta(a_l) \cdot E$, as well as shorter time to idle after consumption is complete, resulting in a lower $\omega(a_{l+1}) \cdot E$. Thus, the latching behavior of consumers allows the load to be more uniform, and reduces the number of wakeups per second by 30% on average versus Sem.

Upon comparing PBPL to BP, the impact on $\omega(a_{l+1}) \cdot E$ is affected by two opposing forces. On one hand, lower final frequency in PBPL ($\vartheta(a_l) \cdot f_e$) helps the CPU go to idle faster. However, BP has a longer idle period and a higher chance of going into deeper sleep. Our experiments confirm this, showing that PBPL increases the number of wakeups by 17% versus BP. This mixed impact on idle energy as well as the large advantage of PBPL in active energy results in a reduction of energy consumption by up to 7.5%.

6.4 Average Response Time Results

As expected, the average response time of batch-processing-based implementations is significantly higher than that of a standard semaphore implementation. Figure 7c shows the response times of experiments based on $M/M/1/\mathbb{B}$ arrival pattern with varying mean inter-arrival periods. The figure shows that both BP and PBPL are orders of magnitude slower than Sem in terms of average response time. Due to the prediction and latching mechanism of PBPL, it has a response time ranging from 82% to 42% of the response time of BP. This is a significant advantage of PBPL, since it can provide better energy savings than BP while reducing response time by half.

PBPL allows the user to control the maximum allowed latency, thus allowing the user to control the tradeoff between energy savings and average response time. To that end, we run a set of experiments that demonstrate the energy savings of PBPL versus Sem when different maximum latency settings are applied. Figure 10 shows the different average response times of PBPL when different latency settings are applied. As can be seen in the figure, a maximum response time setting of any value less than the mean inter-arrival period causes a spike in the energy consumption of PBPL, due to the complicated logic of prediction and latching, and the overhead of setting timers whose period is shorter than the mean inter-arrival rate. Energy savings begin to manifest when the maximum response time is set to the inter-arrival period or higher. This can be easily mitigated by selectively turning PBPL on or off based on the detected mean inter-arrival rate and configured maximum response time. This will result in a system of energy consumption equal to the minimum of PBPL and Sem.

6.5 Throughput Results

Next, we study throughput. Interestingly, while the average response time of batch-based implementations is orders of magnitude slower than that of the semaphore implementations, the throughput of batch-based implementations is consistently equal or higher. Figure 11 demonstrates the throughput recorded for the $M/M/1/\mathbb{B}$ experiments running 30 parallel consumers. As can be seen, both BP and PBPL outperform Sem at all inter-arrival periods. The improvement in throughput reaches 1.58x in case of an inter-arrival period of $10\mu s$. This improvement diminishes at larger inter-arrival periods. The cause for this behavior is that when the inter-arrival period is significantly short the parallel producers and consumers become contentious. This results in a significant increase in context switching overhead in Sem, which causes the producers to block too often waiting for an empty slot in the circular buffer. Frequent blocking results in thrashing between the producers and consumers threads. This thrashing is reduced when using BP and PBPL since they process items in bulk, resulting in less context switching, and higher throughput.

6.6 Buffer Triggers

A buffer trigger indicates the CPU was activated before its scheduled slot. This counteracts the benefits of PBPL by increasing the total number of CPU wakeups. We introduced dynamic buffer resizing to reduce the number of buffer triggers. According to our $M/M/1/\mathbb{B}$ experiments, dynamic buffer resizing reduces the number of buffer triggers by approximately 50%. This translates into an energy saving of 7%-18% compared to a run without dynamic buffer resizing when running 10-30 consumers at an inter-arrival period of $10\mu s$. Table 1 shows the results of the buffer resizing experiments. The energy benefits diminish when increasing the inter-arrival period as shown in Fig. 7a due to the decreased number of wakeups altogether.

TABLE 1: Impact of buffer resizing on energy consumption of $10/20/30$ consumers receiving data from $M/M/1/\mathbb{B}$ producers at 10μs inter-arrival period.

Cons.	Buffer Triggers		Energy	
	no resizing	w/ resizing	no resizing	w/ resizing
10	45K	28K	862J	802J
20	111K	60K	1145J	1066J
30	227K	99K	1580J	1295J

6.7 Discussion

We validated the effectiveness and efficiency of our algorithm by conducting a set of experiments. Our experiments demonstrate the following advantages of using the proposed approach:

- Our proposed approach (PBPL) can reduce energy consumption by up to 38% compared to a standard semaphore implementation of producer-consumer. It can reduce energy consumption by up to 16% compared to a simple batch-processing implementation.
- PBPL consistently has lower response time than BP, in some cases reducing response time by 58%.
- PBPL has a higher throughput than a semaphore-based implementation in most cases, reaching 1.58x in a highly contentious setting.

More importantly, the experiments help identify the circumstances where the proposed approach is beneficial, and where it should not be used.

- If the required maximum response time is too stringent (less than 1ms in our experiments), then PBPL will fail to provide meaningful energy savings. In fact, if transiently that is the case, the system should fall back to a semaphore-based implementation.
- Energy savings are diminished when the inter-arrival period is longer than 10ms.
- PBPL reduces energy consumption and increases throughput when the system is contentious, either due to a high number of consumers, a short inter-arrival period, or both.
- If the inter-arrival period is too short or the number of consumers is too high, PBPL fails to provide any advantages, since the system is almost fully utilized.
- The gap between PBPL and BP in terms of energy is diminished when the buffer size increases. We observed this at buffer sizes 50 and 100, where the CPU wakes up significantly less frequently than buffer size 25, and thus the wake up energy and the length of the idle period have no significant impact on energy consumption.

Another aspect that can influence energy savings is service time. In this paper, we have shown the impact of synchronization strategies – in isolation – on energy consumption. It is important to note that if service time is considered, the impact of PBPL will be affected. As service time increases, the CPU is active for a longer time while servicing requests. This results in an increase in CPU frequency, and consequently, energy consumption. Applications where consumers are CPU-bound (e.g., routers that perform deep packet inspection) are less likely to benefit from our concurrency-based energy savings, since energy consumption is dominated by consumers processing data items. Applications that perform limited computation on data items or serve static content can benefit from PBPL. Examples are web servers serving static content, runtime monitors updating state, or networking buffers passing on requests without any involved processing.

We have also experimented with larger buffer sizes (50 and 100). Similar to our conclusions regarding contentiousness of inter-arrival periods, smaller buffer sizes magnify energy savings. Large buffer sizes result in a relaxed system that does not transition from idle to active very frequently, and hence the energy savings are diminished.

The above findings help identify the system where PBPL is most suitable. If the system is contentious with strict constraints on throughput rather than response time, PBPL is a suitable choice to reduce energy consumption. The situations where PBPL increases energy consumption can be easily detected and conditioned to run an implementation such as Sem.

7 Related Work

The literature related to ideas in this paper can be divided into three categories: (1) DVS scheduling (Subsection 7.1), (2) DPM techniques (Subsection 7.2), and (3) Energy models for concurrency problems (Subsection 7.3).

7.1 DVS Scheduling

The work on DVS scheduling is diverse, targeting real-time systems, as well as workstation-like models where latency is not an issue, rather a quality of service metric. The work in [9]–[13] discusses interval-based approaches to voltage scaling. In these techniques, time is divided into intervals during which CPU utilization is studied and a decision is made on whether to change the CPU frequency.

Our problem differs from work on DVS scheduling in the following aspects:

- Our work does not tackle the problem from a scheduling perspective; specifically not real-time scheduling. Our approach runs at the application level and is independent of the OS scheduler, which is the main target for DVS scheduling literature. Pure DVS scheduling has been argued to lack adaptivity to variable load and varying system dynamics [14]. There is work in feedback control techniques for real-time to counter this deficiency [15].
- We focus on idle power as well as voltage scaling. The majority of work on DVS scheduling deals with the problem of selecting the frequency to assign to a task, with the exception of the line of work by Irani [16], which considers power-down strategies.

7.2 DPM Techniques

DPM techniques have been extensively studied by both the research community [17]–[21] and industry [22]–[24]. One major technique is using prediction schemes to anticipate device usage. Consequently, such schemes decide to change the device power state. When discussing DPM techniques, we focus on power states in which parts (or all) of the CPU is idle. The work in [25], [26] targets building OS level DPM techniques to replace the `ondemand` governor, arguing that all layers of the system should coordinate to adapt to usage needs while targeting reduced energy consumption.

Our work should integrate nicely with work on DPM to form a diverse basis of OS-level energy management. Most of the work on DPM is application agnostic. Our approach focuses on the internal mechanisms of applications that can be modelled using the producer-consumer problem. This allows for more insight into the behavior of the application and more opportunity for energy savings.

7.3 Energy Optimizations for Concurrency Primitives

The work in [27] uses performance counters to estimate the power consumption of a system. The paper proposes *power weights*, which map hardware performance counters to power consumption. The work in [28] proposes a method to estimate per-core power consumption using CPU frequency and IPC as the only PMC (Performance Monitoring Counter) used.

The work in [29] experimentally evaluates the performance and energy of both lock-based and lock-free FIFO queues using a set of contentious workloads. The work in [30] compares locks to Software Transactional Memory (STM) in terms of energy efficiency. The work in [31] introduces an analytical energy model for lock-free queues.

The work in [32] studies the energy consumption profile of different lock configurations, and proposes a new implementation of locks that sacrifices fairness for energy efficiency. Similar to our work, the paper concludes that energy savings come hand in hand with an improvement in throughput.

Our proposed problem can be compared to the work above in the following points:

- Energy modelling targets prediction of energy demand of a system and its feasibility by avoiding exhaustive empirical analysis. However, such models entail assumptions and extrapolation that limit the applicability of the model to a diverse range of systems. Active research in energy modelling should result in high fidelity models with reduced empirical dependence as shown in [31]. In that sense, currently we rely on online adaptation to construct *real-time models* used to make energy-aware decisions.
- Our objective is constructing online adaptive approaches. An inherent condition is that these approaches should have a low energy footprint. This objective is not stressed in related work on energy modelling, since their purpose is to construct a high-fidelity model and not a robust online mechanism.
- Our approach assumes the existence of a DPM strategy and a DVFS governor unlike the majority of work on energy modelling.

8 Conclusion

In this paper, we proposed a novel energy-efficient algorithm for the multiple producer-consumer problem for multicore systems, where each consumer is associated with one and only one producer. To our knowledge, this is the first instance of such an algorithm. Our approach is based on dynamic periodic batch processing, such that consumers process a set of items and let the CPU switch to idle state, hence, saving energy. Consumers make prediction about the rate of incoming data items and group themselves together. This results in two energy reducing consequences: (1) the number of wakeups is significantly reduced versus trivial batch processing, and (2) high loads are mitigated and spread out uniformly preventing DVFS mechanisms from promoting the CPU frequency, resulting in the CPU spending most of its time at lower frequencies.

We observed that our algorithm can lower energy consumption by 38% compared to a mutex/semaphore implementation when running 10 parallel consumers. In fact, it provides up to 16% improvement over a simple batch processing implementation. We also observe that our algorithm excels with the increase in the number of consumers, making it scalable and robust.

For future work, we are considering other techniques such as using Kalman filter for estimating producer rate with better accuracy. We are planning to adapt and test our approach in other domains, such as operating system kernels and in runtime monitoring. Another interesting research direction is to design a generic *resource-aware* producer-consumer algorithms, where energy, memory, CPU overhead, timing constraints, etc need to be taken into account simultaneously.

References

[1] J. D. Moore, J. S. Chase, P. Ranganathan, and R. K. Sharma, "Making scheduling "cool": Temperature-aware workload placement in data centers," in *USENIX Annual Technical Conference, General Track*, 2005, pp. 61–75.

[2] P. Ranganathan, P. Leech, D. E. Irwin, and J. S. Chase, "Ensemble-level power management for dense blade servers," in *Proceedings of the 33rd International Symposium on Computer Architecture (ISCA)*, 2006, pp. 66–77.

[3] M. Sachenbacher, M. Leucker, A. Artmeier, and J. Haselmayr, "Efficient energy-optimal routing for electric vehicles," in *Proceedings of the 25th Conference on Artificial Intelligence, (AAAI)*, 2011.

[4] L. A. Barroso and U. Hölzle, "The case for energy-proportional computing," *IEEE Computers*, vol. 40, no. 12, pp. 33–37, 2007.

[5] M. Arlitt and T. Jin, “1998 world cup web site access logs,” 1998.

[6] R. Ge, X. Feng, and K. W. Cameron, “Improvement of power-performance efficiency for high-end computing,” in *Proceedings of the 19th International Parallel and Distributed Processing Symposium (IPDPS)*, 2005, p. 8.

[7] “The Haswell Review,” 2013, https://www.anandtech.com/show/7003/the-haswell-review-intel-core-i74770k-i54560k-tested/2.

[8] J. Sztrik, “Basic queueing theory,” *University of Debrecen, Faculty of Informatics*, vol. 193, 2012.

[9] M. Weiser, B. Welch, A. Demers, and S. Shenker, “Scheduling for reduced CPU energy,” in *Mobile Computing*. Springer, 1996, pp. 449–471.

[10] K. Govil, E. Chan, and H. Wasserman, “Comparing algorithm for dynamic speed-setting of a low-power CPU,” in *Proceedings of the 1st annual international conference on Mobile computing and networking*. ACM, 1995, pp. 13–25.

[11] F. Yao, A. Demers, and S. Shenker, “A scheduling model for reduced CPU energy,” in *Proceedings of 36th Annual Symposium on the Foundations of Computer Science*. IEEE, 1995, pp. 374–382.

[12] T. Pering, T. Burd, and R. Brodersen, “The simulation and evaluation of dynamic voltage scaling algorithms,” in *Proceedings of the 1998 international symposium on Low power electronics and design*. ACM, 1998, pp. 76–81.

[13] P. Pillai and K. G. Shin, “Real-time dynamic voltage scaling for low-power embedded operating systems,” in *ACM SIGOPS Operating Systems Review*, vol. 35. ACM, 2001, pp. 89–102.

[14] D. Shin, J. Kim, and S. Lee, “Intra-task voltage scheduling for low-energy, hard real-time applications,” *IEEE Design & Test of Computers*, pp. 20–30, 2001.

[15] Y. Zhu and F. Mueller, “Feedback edf scheduling exploiting dynamic voltage scaling,” in *Real-Time and Embedded Technology and Applications Symposium, 2004. Proceedings. RTAS 2004. 10th IEEE*. IEEE, 2004, pp. 84–93.

[16] S. Irani, S. Shukla, and R. Gupta, “Algorithms for power savings,” *ACM Transactions on Algorithms (TALG)*, vol. 3, no. 4, p. 41, 2007.

[17] C.-H. Hwang and A. C.-H. Wu, “A predictive system shutdown method for energy saving of event-driven computation,” *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 5, no. 2, pp. 226–241, 2000.

[18] M. B. Srivastava, A. P. Chandrakasan, and R. W. Brodersen, “Predictive system shutdown and other architectural techniques for energy efficient programmable computation,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 4, no. 1, pp. 42–55, 1996.

[19] S. K. Shukla and R. K. Gupta, “A model checking approach to evaluating system level dynamic power management policies for embedded systems,” in *Proceedings of the Sixth IEEE International High-Level Design Validation and Test Workshop*. IEEE, 2001, pp. 53–57.

[20] D. Ramanathan, S. Irani, and R. Gupta, “Latency effects of system level power management algorithms,” in *Proceedings of the 2000 IEEE/ACM international conference on Computer-aided design*. IEEE Press, 2000, pp. 350–356.

[21] S. Irani, R. Gupta, and S. Shukla, “Competitive analysis of dynamic power management strategies for systems with multiple power savings states,” in *Proceedings of the conference on Design, automation and test in Europe*. IEEE, 2002, p. 117.

[22] “Intel TurboBoost,” 2015, http://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html.

[23] C. C. Corporation and R. B, “Advanced configuration and power interface specification,” 2000. [Online]. Available: http://www.acpi.info/

[24] “AMD PowerNow!” 2015, http://support.amd.com/TechDocs/24404a.pdf.

[25] D. G. Sachs, W. Yuan, C. J. Hughes, A. F. Harris III, S. V. Adve, D. L. Jones, R. H. Kravets, and K. Nahrstedt, “Grace: A hierarchical adaptation framework for saving energy,” Tech. Rep., 2004.

[26] V. Vardhan, W. Yuan, A. F. Harris, S. V. Adve, R. Kravets, K. Nahrstedt, D. Sachs, and D. Jones, “Grace-2: integrating fine-grained application adaptation with global adaptation for saving energy,” *International Journal of Embedded Systems*, vol. 4, no. 2, pp. 152–169, 2009.

[27] G. Contreras and M. Martonosi, “Power prediction for intel xscale® processors using performance monitoring unit events,” in *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*. IEEE, 2005, pp. 221–226.

[28] S. Wang, H. Chen, and W. Shi, “Span: A software power analyzer for multicore computer systems,” *Sustainable Computing: Informatics and Systems*, vol. 1, no. 1, pp. 23–34, 2011.

[29] N. Hunt, P. S. Sandhu, and L. Ceze, “Characterizing the performance and energy efficiency of lock-free data structures,” in *15th Workshop on the Interaction between Compilers and Computer Architectures (INTERACT)*. IEEE, 2011, pp. 63–70.

[30] A. Gautham, K. Korgaonkar, P. Slpsk, S. Balachandran, and K. Veezhinathan, “The implications of shared data synchronization techniques on multi-core energy efficiency,” in *Proceedings of the 2012 USENIX conference on Power-Aware Computing and Systems*. USENIX Association, 2012, pp. 6–6.

[31] A. Atalar, A. Gidenstam, P. Renaud-Goud, and P. Tsigas, “Modeling energy consumption of lock-free queue implementations,” Chalmers University of Technology, Tech. Rep., 2015.

[32] B. Falsafi, R. Guerraoui, J. Picorel, and V. Trigonakis, “Unlocking energy.” in *USENIX Annual Technical Conference*, 2016, pp. 393–406.

Ramy Medhat has received his PhD from the Electrical and Computer Engineering department at the University of Waterloo. He received his B. Sc. and M. Sc. in Computer Science from Ain Shams University, Egypt in 2007 and 2011 respectively. His research focuses on algorithms for software energy efficiency in both the embedded and distributed domains. His research interests also include runtime verification and specification mining of embedded systems.

Borzoo Bonakdarpour is currently an assistant professor of Computer Science at Iowa State University, USA. He received his B. Sc. in Computer Engineering (Software) from the University of Esfahan, Iran, in 1999 and his M. Sc. and Ph.D. degrees in Computer Science from Michigan State University in 2004 and 2009. He has developed several efficient automated synthesis techniques that make synthesis of complex fault-tolerant distributed protocols possible. His Ph.D. dissertation on this subject was nominated for the ACM Doctoral Dissertation Award. He has received Best Paper awards from IEEE SRDS’17, SSS’14, and IEEE SIES’10 conferences. His current research interests include dependable distributed systems, runtime monitoring, multi-UAV autonomous path planning, and information-flow security.

Sebastian Fischmeister Sebastian Fischmeister received the Dipl.-Ing. degree in Computer Science at the Vienna University of Technology, Austria, in March 2000, and his Ph.D. degree in Computer Science at the University of Salzburg, Austria in December 2002. He continued working at the University of Salzburg as researcher and lecturer and was awarded the Austrian APART stipend in 2005. He subsequently worked at the University of Pennsylvania, USA, as Post Graduate Research Associate until 2008. Sebastian Fischmeister is currently Associate Professor at the Department of Electrical and Computer Engineering at the University of Waterloo, Canada.