2014 IEEE 28th International Parallel & Distributed Processing Symposium

# Power-efficient Multiple Producer-Consumer

Ramy Medhat
Dept. of Elec. and Comp. Eng.
University of Waterloo, Canada
Email: rmedhat@uwaterloo.ca

Borzoo Bonakdarpour
School of Computer Science
University of Waterloo, Canada
Email: borzoo@cs.uwaterloo.ca

Sebastian Fischmeister
Dept. of Elec. and Comp. Eng.
University of Waterloo, Canada
Email: sfischme@uwaterloo.ca

***Abstract*—Power efficiency has been one of the main objectives of hardware design in the last two decades. However, with the recent explosion of mobile computing and the increasing demand for green data centers, software power efficiency has also risen to be an equally important factor. We argue that most classic concurrency control algorithms were designed in an era when power efficiency was not an important dimension in algorithm design. Such 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 computer system and by larger scale in networking devices and data centers. Thus, even a small change in their power spectrum can make a huge impact on overall power consumption in long periods of time.**

**This paper focuses on the classic *producer-consumer* problem. First, we study the power efficiency of different existing implementations of the producer-consumer problem. In particular, we present evidence that these implementations behave drastically differently with respect to power consumption. Secondly, 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 results in minimizing the overall number of CPU wakeups and in effect, power consumption. We enable consumers to dynamically reserve more preallocated 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 up to $40\%$ decrease in the number of CPU wakeups, and $30\%$ decrease in power consumption. We validate the scalability of our algorithm with an increasing number of consumers.**

***Keywords*-Concurrency control; Green computing; Power; Energy; Synchronization**

## I. Introduction

Designing low-power computing system architectures has been an active area of research in the recent years, partly due to increasing cost of energy as well as the high demands 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, power 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 power efficiency plays an important role is in large-scale data centers. In fact, power and cooling are the largest cost of a data center. For example, a facility consisting of $30,000$ square feet and consuming 10MW, for instance, requires an accompanying cooling system that costs from \$2-\$5 million [10], and the yearly cost of running this cooling infrastructure can reach up to \$4-\$8 million [11]. These numbers simply mean that even a small improvement in power usage leads to significant cost reduction. Hence, it is quite evident that we are undoubtedly in pressing need to design and implement power-efficient hardware and software artifacts in order to keep up with the increasingly high demand in mobile as well as big-data applications.

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. However, these algorithms were designed in an era when power 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 [12]. Thus, we argue that many of such classic algorithms need to be re-visited and re-designed, so that power constraints are treated as a first-class citizen. Some of these algorithms are applied in such a high capacity that even small improvements in their power consumption behavior may have a huge impact in the power 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 producer process generates data items and puts them into the buffer and starts again. Concurrently, the consumer process is consuming the data by removing them from the buffer, one item at a time. Since the two processes work concurrently, an algorithm has to synchronize them, so that (1) the producer does not attempt to add data into the buffer if it is full, (2) the consumer will not try to remove data from an empty buffer, (3) access to a buffer location is mutually exclusive, and (4) deadlock scenarios do not occur. The producer-consumer problem is applicable to a multitude of real-world scenarios in many systems around us. 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

1530-2075/14 \$31.00 © 2014 IEEE
DOI 10.1109/IPDPS.2014.75

drivers.

- *Web servers.* HTTP requests produced by web browsers are stored in buffers that are consumed and processed by multiple threads in a web server.
- *Runtime monitoring.* In runtime monitoring, events produced by the environment or internal system processes are consumed and processed by a runtime monitor.
- *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.

In this paper, we propose a novel power-efficient algorithm for the multiple producer-consumer 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 power consumption behavior of the problem, we first conducted a study to analyze the power profile of existing popular implementations of the single producer-consumer problem. The implementations in our analysis consist of a busy-waiting algorithm, two algorithms based on synchronization data structures (i.e., semaphores and mutexes), and three algorithms that employ batch processing. We observed that these implementations behave drastically differently with respect to power consumption. While the busy-waiting algorithm is the worst in power efficiency due to high CPU utilization, the algorithms, where the consumer processes data items in batches are the most power-efficient due to lowest number of CPU wakeups. In particular, batch processing results in up to $80\%$ reduction in power as compared to busy-waiting and up to $33\%$ as compared to the semaphore-based implementations. This is validated by a strong positive correlation between wakeups and power consumption. Such a dramatic shift in power profile clearly motivates the need for designing a power-aware solution for the producer-consumer problem.

Roughly speaking, our proposed algorithm exploits bounded-time dynamic batch processing. It interprets time as a track with periodic slots. To minimize core 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.

We argue that our dynamic batch processing approach is in particular highly beneficial in web services. According to a Google study [5], 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, CPU contributes to more than $50\%$ of Google server power consumption. In such servers, our approach results in periods of high CPU utilization and periods of complete idleness, which saves a great deal of energy. We validate this claim by conducting thorough experiments on a data set from a web server's incoming HTTP requests log [4]. Our results show that our algorithm can lower power consumption by up to $30\%$ compared to a mutex implementation. In fact, it provides up to $13\%$ improvement over the most power efficient implementation in our study. We experiment with increasing the number of consumers and the results validate the effectiveness of our approach and its scalability.

*Organization:* The rest of the paper is organized as follows. In Section II, we describe the background concepts on CPU power states. Section III presents our findings on power profile of various implementations of the producer-consumer problem. We formally state the power optimization objective for the multiple producer-consumer problem in Section IV. Our power-efficient solution to multiple producer-consumer is described in Section V, while Section VI analyzes the results of experiments. Related work is discussed in Section VII. Finally, we make concluding remarks and discuss future work in Section VIII.

## II. Background

Power management technologies approach 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 runtime that determine which power state the system should be in. DPM uses different techniques to infer whether a transition to a more efficient state is worthwhile or not, 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 flavores, e.g., Dynamic Voltage Scaling (DVS) and Dynamic Frequency Scaling (DFS). DVS scales the voltage at which the CPU operates, and, thus, controls its power 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 [7]. DFS scales the frequency at which the CPU operates, such that when a high demand occurs, the frequency is raised to meet that demand, at a higher energy cost. When the CPU utilization drops, so does the

operating frequency, causing a decrease in 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 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 power saving concept based on the premise that it is more power efficient to execute the task at hand faster (a higher P-state) and then go to idle mode (i.e., a deeper C-state). Race-to-idle is based on the fact that idle power is significantly lower 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 idle mode. This indicates that hardware manufacturers are moving towards approaches that attempt to increase CPU residency in deeper C-states.

Although race-to-idle is a valid approach, it cannot be used as a standalone strategy. The reason for this is the cost of processor *wakeup*; i.e., the energy needed for reactivating the CPU. Thus, even though fast transition to idle implies saving power during the idle period, the wakeup cost may not be worth it. In other words, a certain delay must occur in order for idle mode to be advantageous. Figure 1 illustrates that more contiguous idle time is more efficient. Thus, a valid power saving strategy is to minimize the number of wakeups that a CPU undergoes. 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.

## III. Producer-Consumer Power Profile

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

### A. 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 circular buffers.

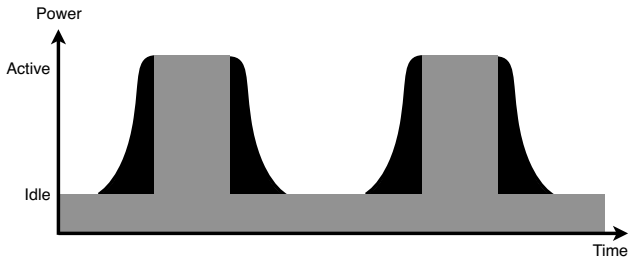



Fig. 1. Overhead due to waking up and idling the CPU. If both peaks are grouped, wakeup overhead becomes lower.

We study the following implementations:

- **Busy-waiting (BW).** This is the most trivial implementation, where the consumer simply busy-waits until the tail is not equal to the head of the buffer; i.e., meaning at least one item has been inserted.
- **Yield.** This is similar to the busy-waiting implementation, except the consumer yields the CPU voluntarily.
- **Mutexes and conditional variables (Mutex).** This implementation uses a mutex to ensure mutually exclusive concurrent access to a non-circular buffer. Thus, reading and writing from it requires atomicity to be able to track the number of items inside. 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 a circular buffer and two semaphores used for synchronizing emptiness and fullness of the buffer.
- **Batch processing (BP).** This implementation is similar to the semaphore-based 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 nanosleep() system call) instead of whenever the buffer gets filled. The period for this experiment is $100\mu s$.
- **Signal-based periodic batch processing (SPBP).** This implementation is identical to periodic batch processing except it uses UNIX signals instead of nanosleep().

### B. Experimental Settings

We study the power consumption of the different producer-consumer implementations using two methods: (1) PowerTop, and (2) measuring voltage drop across a resistor. The results of these two methods are combined to provide insight into the power consumption trends of the different implementations.

PowerTop[1] is a popular Linux tool that uses the ACPI subsystem and CPU performance counters to estimate the power consumption of every 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 time that the process consumes. The unit for CPU usage in PowerTop is milliseconds per second, meaning the number of milliseconds the process spends executing every second. For a process that is executing non-stop, this would be 1000 ms/s.

Our power measurement setup uses a resistor in series on the live power feed of the system. The resistor is chosen to support sufficient power as per the requirements of the system, and low enough resistance to not prevent the system from booting (see Figure 2). We measure the voltage drop across the resistor and use it to determine the amount of power consumed using the following formula:

$$P = \frac{V^2}{R}$$

where $R$ is the known resistance of our resistor, $V$ is the voltage drop across the resistor, and $P$ is the power in watts.

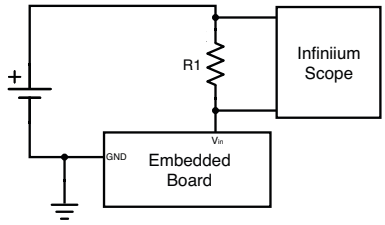



Fig. 2. Schematic of the power measurement circuit.

We use an Agilent Infiniium 54853A scope to measure the voltage drop across the resistor. The Infiniium is a powerful machine that supports sampling up to 20GS/s. The system under inspection is an Arndale Samsung Exynos 5 board that hosts a dual-core ARM A-15 Cortex M3 chip. The board runs the Linaro operating system, which is essentially Ubuntu for ARM. The reason for choosing Linaro is because it includes a powerful dynamic power manager, with extensive optimization including and not limited to efficient use of the Wait For Interrupts (WFI) command. This board is frequently used to build prototype Android devices and comes preloaded with Android.

Finally, each implementation of producer-consumer is tested using a non-linear dataset of a web server request logs [4]. This dataset exhibits sporadic changes in the rate of production of items. Each experiment executes for 50 seconds. We execute 3 replicates of each experiment for statistical confidence. $95\%$ confidence intervals are calculated for all measurements. We measure three metrics in each experiment:

- **Power (watts).** The number of extra watts consumed by the system when the respective implementation is executed.
- **Wakeups/s.** The number of CPU wakeups per second due to the respective implementation.
- **Usage (ms/s).** The number of milliseconds out of every second that the CPU spends executing the respective implementation.

[1]https://01.org/powertop/

### C. Experimental Results

*1) Sanity Checks:* We perform the following set of sanity checks to ensure our experimental setup is valid:

- Ensuring that measured voltages are reasonable considering the voltage rating of the embedded board and the size of the resistor.
- We execute a test with a busy waiting multithreaded program running on both cores of the processor, and we ensure that no experiment reaches the power consumption found in that implementation.
- We execute a test where no background processes are running except kernel tasks, and we measure the power. We ensure that the power 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.

*2) Effect of CPU Usage on Power Consumption:* CPU usage demonstrates a significant effect on power consumption across all seven implementations (see Figures 3 and 4). This is strongly apparent in BW and Yield implementations. For BW, the CPU spends $99.5\%$ of its time executing the consumer process. The number of wakeups is significantly lower than every other implementation. However, a highly utilized CPU is bound to consume more significant amount of energy. The Yield implementation uses slightly less power and that is attributed to DVFS setting the CPU frequency to a smaller value due to the yield instructions.

Upon excluding BW and Yield from the comparison, the results seem dramatically different. CPU usage has a weak positive correlation with the power consumed ($12\%$), and in fact, it exhibits a significant amount of noise as seen in the larger error bars of the latter five implementations. Since these five implementations are similar in the sense that they are fundamentally based on idling the CPU one way or another, it is more reasonable to compare them collectively.

*3) Effect of wakeups on power:* Upon comparing *all* seven implementations in Figures 3 and 4, the number of wakeups per second has a strong negative correlation of $-79.6\%$ with power. However, this result is biased by the huge CPU usage that BW and Yield impose. In fact, the latter five implementations show a strong positive correlation of $74\%$ between wakeups and power consumption. Since the usage is similar for these implementations, wakeups/s is the stronger deciding factor affecting power. SPBP exhibits the least number of wakeups/s and in effect offers a $33\%$ reduction in power consumption over the more popular Mutex implementation. To validate our hypothesis, we run the following hypothesis test: $H_0$ : *Wakeups have a significant effect on power*. We manage to accept the hypothesis with $99\%$ confidence.

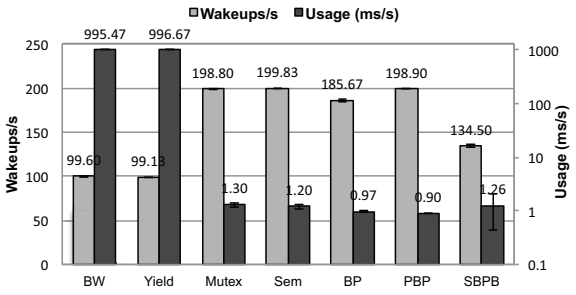



Fig. 3. A plot of wakeups/s versus usage ms/s for all seven implementations.

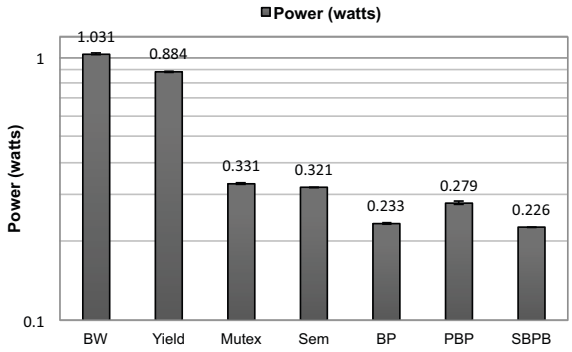



Fig. 4. A plot of the power consumption in watts for all seven implementations (log scale).

BP has the second lowest number of wakeups and the second lowest power consumption. In fact, all three batch-based implementations are the most power efficient. This is particularly remarkable since the more popular implementations of producer-consumer (i.e., Mutex and Sem) are the least power-efficient among the five latter implementations. On the contrary, batch-based implementations require more complex synchronization, and in case of periodic batching, it requires logic to handle the overflow of the buffer before a period expires, which makes the implementation more complicated. However, a 33% reduction in power is a highly significant gain, specially when applied to a fundamental and widely used problem such as producer-consumer.

Another important observation is the decrease in the number of wakeups from periodic batching to periodic batching with signals. We believe that this improvement is due to the accuracy of SIGALRM signals compared to the sleep() system call. The jitter associated with sleep() causes more buffer overflows and thus, more wakeups.

Thus, with the exception of BW and Yield, the results indicate the significance of the number of wakeups in power consumption. Despite the fact that all implementations consume the same number of data items, batch processing provides a reduction in power up to 33%. Batch processing has its drawbacks, mainly of which is the latency in responding to items. 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 a power-efficient and acceptable solution.

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

*We believe that the power profile and reduction level in power consumption observed in our experiments strongly justify the pressing need to re-visit the design and implementation of classic algorithms to make them power efficient.*

## IV. Formal Problem Description

### A. System Assumptions

As observed in Section III, minimizing the number of wakeups in a CPU may lead to decreased power consumption. We begin by stating the assumptions on which the problem is based:

- *Multicore system.* The system we attempt to optimize for power is a multicore system, which supports core parking.
- *Simplified power model.* For simplicity, the system does not support frequency scaling and operates at two states: idle and active.
- *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 latencies.* 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 latency.
- *Consumer isolation.* All consumers are isolated from background processes by being locked to a set of cores on which no background process is allowed to execute. This assumption is to isolate the effect of background processes that can potentially wakeup a core on which a consumer is executing. We also assume producers are either processes on separate cores or external events, such that they do not interfere with consumers.

### B. The Optimization Problem

We formalize the multiple producer-consumer power efficiency problem by defining a multicore system that consists of a set of CPU cores $\alpha = \{\alpha_1, \alpha_2, \ldots, \alpha_A\}$.

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 $d_1, d_2, \ldots, d_{N_i}$, where $N_i$ is the total number of items produced by producer $p_i$. For producer $p_i$, the time at which it produces data item $d_j$ is

$\upsilon_{i,j}$. Note that $N_i$ should not conflict with infinitely running producers. Yet we simplify this by requiring minimum power consumption during a finite period of time.

The consumers in the set $C = \{c_1, c_2, \ldots, c_M\}$ consume data in batches. That is, they are idle for a period of time until they process all buffered data items in one batch, after which they become idle again. For each consumer $c_i$, $1 \leq i \leq M$, let the invocation times of the consumer be $\tau_{i,1}, \tau_{i,2}, \ldots, \tau_{i,k_i}$, where $k_i$ is the number of invocations of the consumer. $k_i$ depends on the time of the invocations and the number of events in between. To clarify this, let us define the function $\gamma$ as follows:

$$\gamma_i\left(\tau_{i,m-1}, \tau_{i,m}\right) = \{d_j \mid \tau_{i,m-1} \leq \upsilon_{i,j} < \tau_{i,m}\} \quad (1)$$

Thus, for a consumer $c_i$, $\gamma_i$ is the set of data items produced by the producer $p_i$ between times $\tau_{i,m-1}$ and $\tau_{i,m}$. Using $\gamma$, we can define $k_i$ as the index of the consumer invocation that processes the last data item produced by the producer. In other words, $k_i$ is the invocation such that

$$d_{N_i} \in \gamma_i\left(\tau_{i,k_i-1}, \tau_{i,k_i}\right) \quad (2)$$

Every core $\alpha_l$ hosts a set of consumers $C_{\alpha_l} \subset C$, where $C_{\alpha_l} \cap C_{\alpha_{l'}} = \emptyset$, for all distinct $l$ and $l'$. We define the $f : C \rightarrow \alpha$ as a function that maps every consumer to a core. The function $s : \alpha \times \mathbb{N} \rightarrow \{\mathsf{idle}, \mathsf{active}\}$ returns the (idle or active) state of a core in $\alpha$ at a point of time in $\mathbb{N}$.

Based on our assumptions, a core $f(c_i)$ is idle unless the consumer $c_i$ is activated and is processing data. If a core is idle, and an invocation $\tau_{i,j}$ of $c_i$ occurs, then the core is activated with a cost $\omega$, which is the wakeup cost (in terms of power) of that core. This leads to defining the function $w$ as follows:

$$w(\tau_{i,j}) = \begin{cases} \omega & s(f(c_i), \tau_{i,j}) = \mathsf{idle} \\ 0 & s(f(c_i), \tau_{i,j}) = \mathsf{active} \end{cases} \quad (3)$$

Thus, the optimization objective to build a power-efficient multiple producer-consumer is the following (to minimize the number of wakeups):

$$\min \left\{ \sum_{i=1}^{M} \sum_{j=1}^{k_i} w\left(\tau_{i,j}\right) \right\} \quad (4)$$

## V. Power-aware Multiple Producer-Consumer Algorithm

This section presents the design of our algorithm to solve the multiple producer-consumer problem presented in Section IV. Figure 5 illustrates the building blocks of our solution for a multicore system. Roughly speaking, our algorithm interprets time as a *track* with periodic *slots*. To minimize core wakeups, it dynamically constitutes track slots, so consumers can latch on and exploit a CPU wakeup. Given a set $\alpha$ of cores, since each core in $\alpha$ hosts a set of consumers $C_{\alpha_l}$, we introduce a *core manager* component that targets aligning consumers to the slots in that core's track ($S_l$). The core manager is responsible for managing the slot allocations on the track of its respective core. Consumers are designed so that they can predicate production rate of data items to compute an appropriate latching time.

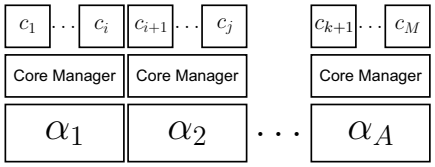



Fig. 5. The architecture of the proposed solution.

In the rest of this section, we describe our wakeup minimization technique and design choices for core managers and consumers in Subsections V-A, V-B, and V-C, respectively.

### A. Minimizing Wakeups

As established in Section III, the number of wakeups strongly affects power consumption. Recall that in Section III, we identified batch processing as a more power-efficient implementation of the producer-consumer problem. Now, consider three consumers: $A$, $B$, and $C$ 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 invocation pattern of these three consumers could be as shown in Figure 6(a). As can be seen, the CPU is activated 8 times, which can potentially impose significant overhead. Our idea is to group the invocations together, so that one CPU wakeup handles multiple consumers.

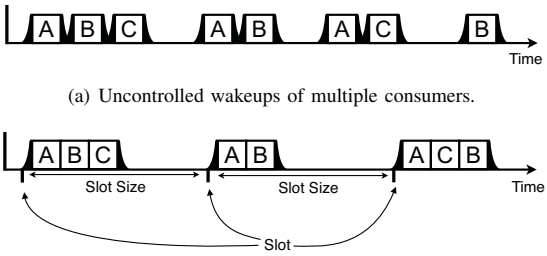
Our algorithm attempts to group consumer invocations dynamically. We begin with interpreting time as a *track* with periodic *slots*. This is based on the metaphor of a race track with markings every $X$ number of meters. In our case, this is denoted as the *slot size* $\Delta$. The default slot size is equal to the minimum of all maximum acceptable response latencies defined by the producer-consumer pairs. Figure 6(b) presents this idea. Observe that upon grouping, the number of wakeups is reduced to 3. This example illustrates the potential impact of grouping on the number of wakeups. Let the timestamps of the start of these slots be the set $S = \{s_1, s_2, s_3, \ldots \infty\}$. The initial objective of the algorithm is to ensure that all consumer invocations are aligned to the slots:

$$\forall i, j : \ \tau_{i,j} \in S \quad (5)$$

Although this objective is ideal, it is not realistic, since a buffer overflow can occur at any time. To reach a realistic objective, we first define the function $g : \mathbb{N} \rightarrow \mathbb{N}$ that returns the slot closest to a certain consumer invocation, with the condition that it is earlier than that invocation, since if it is later it implies that a buffer overflow has occurred.

$$g(\tau) = \inf\{s \in S \mid s \leq \tau\} \quad (6)$$

Thus, a realistic optimization objective is to minimize the difference between an invocation and its nearest slot across all consumer invocations on all cores.




(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.

$$\min \left\{ \sum_{i=1}^{M} \sum_{j=1}^{k_i} \left| \tau_{i,j} - g\left(\tau_{i,j}\right) \right| \right\} \tag{7}$$

Obviously, this minimum is equal to 0, if all invocations $\tau_{i,j}$ are aligned to slots.

Although achieving this objective for an appropriately sized $\Delta$ would result in a decrease in the number of wakeups, there is still more room for improvement. To this end, we consider the actual gain from realigning a consumer invocation to a certain slot instead of aligning blindly. We use this to determine the optimum slot to select. The basic principle of this optimality is based upon the fact that if the CPU is already awake at a specific point in time, then it is beneficial to schedule consumers to be invoked at that same time. In that sense, consumers are *latching on* a wakeup caused by another consumer. This is further explained in the consumer design in Subsection V-C.

A second optimization objective that arises from attempting to apply the idea of consumer latching is buffer utilization. Latching manages to decrease the number of wakeups across multiple consumers, yet decreasing the number of wakeups within one consumer is also an important objective. Recall that in Equation 2, $k_i$ is the number of invocations of consumer $c_i$. Decreasing this number has a direct effect on the number of wakeups. To minimize $k_i$, maximum utilization of the buffer should be achieved at every invocation since the value of $k_i$ depends on $\gamma_i$ (see Equation 1).

### B. Core Manager Design

The core manager accepts reservation requests for specific slots made by the consumers. It maintains a list of consumers to invoke at every slot, and supports deregistering if a consumer decides a slot is no longer appropriate. Figure 7 presents the sequence of operations performed at every scheduled CPU wakeup. The core manager performs the following steps:

- Upon a scheduled wakeup, the core manager looks up the registered consumers for the current slot, and activates them. This can be achieved by, for instance, signaling a semaphore.
- After all registered consumers finish executing, the core manager determines the next slot to wake up. Note that this does not necessarily have to be at time $s_i + \Delta$ where $s_i$ 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.

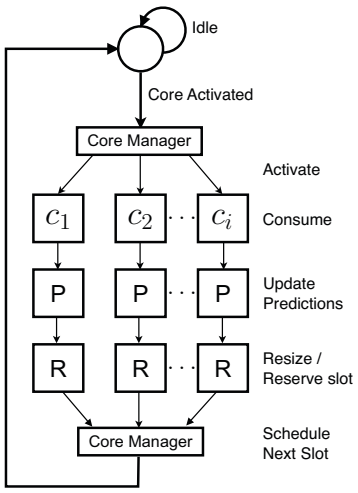



Fig. 7. The sequence of operations performed at every scheduled CPU wakeup.

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 invocation of every consumer.

### C. Consumer Design

The fundamental part of the proposed solution is the design of the consumer. On a principal level all consumers behave identically and are designed to be autonomous. The scheduling aspect of the consumer invocation should not be dictated by the system. This assists in maintaining flexibility and scalability in the design. Figure 7 illustrates the functions performed by each consumer after it is activated by a core manager. In short, a consumer attempts to (1) predict the rate of items produced, (2) reserve a slot for latching, and (3) dynamically resize the shared buffer, only if greater power saving can be achieved.

We now discuss the functions in detail:

- **Prediction.** The consumer attempts to predict 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.

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

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

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

where $\gamma_i$ 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 power consumption.

- **Reservation.** After predicting the upcoming rate of items, the consumer attempts to reserve a slot. The process of selecting a slot to reserve is based on minimizing the cost function $\rho$ over the set of possible slots:

$$\rho\left(s_j\right) = \frac{w\left(s_j\right) + e\left(\hat{r}_{i+1} \cdot \left(s_j - s_i\right)\right)}{\hat{r}_{i+1} \cdot \left(s_j - s_i\right)} \tag{8}$$

where $s_j$ is the slot being evaluated, $s_i$ is the current slot, $e\left(x\right)$ is the energy consumed by processing $x$ data items, and $w$ is defined in Equation 3. Naturally, the cost of a wakeup $\omega$ is much higher than the cost of processing one data item. $\hat{r}_{i+1} \cdot \left(s_j - s_i\right)$ is used to calculate the number of events predicted to have been buffered in slot $s_j$. The cost function $\rho$ is normalized to represent the cost per data item. This gives consumers 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.

Let the size of the buffer that the consumer reads from be $B$, thus, given that the current time (slot) is $s_i$ and the predicted rate is $\hat{r}_{i+1}$, the time expected to fill the buffer is $s_i + B/\hat{r}_{i+1}$. The consumer starts evaluation at the slot determined by $g(s_i + B/\hat{r}_{i+1})$ and backtracks until it is impossible to find a slot with lower $\rho$. If the $j^{th}$ slot being evaluated has higher $\rho$ than its predecessor $j-1$, then it is safe to assume that no better slots can be found by further backtracking. The reason for this is that if the $j^{th}$ slot has a higher cost than $j-1$ (since we are backtracking, $j-1$ occurs later than $j$), this implies that the added cost is due to a wakeup, since it is impossible for the cost per item to *increase* when we *decrease* the time period. 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.

- **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. Figure 6(b) demonstrates such a problem. The third 'A' and second 'C' invocations occur earlier than the closest slot. This implies that a buffer overflow will occur prior to the activation of the next slot. Our algorithm attempts to resolve this issue by dynamic buffer resizing solution.

Figure 8 shows how dynamic resizing works. Initially, each consumer is provided with a preallocated buffer space of size $B_0$. This buffer space is in fact part of what we call a *global buffer* ($B_g$): a preallocated buffer of size $B_g = B_0 \times M$, where $M$ is the number of consumers. When a consumer selects a slot to reserve, it calculates the predicted number of items to be found when that slot is triggered. This prediction is used to downsize the consumer's buffer such that it is only sufficient to accomodate the predicted items and not more. Thus, for consumer $c_i$, the downsizing of buffer size $B_i$ is as follows:

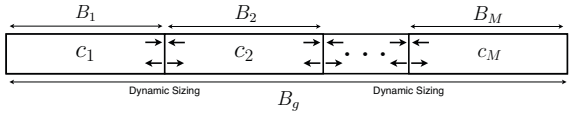



Fig. 8. Dynamic buffer resizing.

$$B_i = \hat{r}_{j+1} \cdot \left(\tau_{i,j+1} - \tau_{i,j}\right)$$

where $\hat{r}_{j+1}$ is the predicted rate, $\tau_{i,j}$ is the current time slot and $\tau_{i,j+1}$ is the next slot that has just been reserved. When another consumer fails to find a slot that can support its expected high rate of items, it requests to resize its buffer according to the space available. This upsizing is calculated as follows:

$$B_i = \min\left\{B_g - \sum_{q=1}^{M} B_q \;,\; \hat{r}_{j+1}\left(\tau_{i,j+1} - \tau_{i,j}\right)\right\}$$

This virtually causes the walls between the consumer buffers shown in Figure 8 to be elastic, providing more memory to the consumers in need. This is implemented using linked lists and is, hence, not actual contiguous resizing as shown in the figure.

## VI. Implementation and 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 identical to the ones presented in Subsection III-B

### A. Experimental Parameters

The experiments are based on executing producer-consumer pairs in parallel. The producers use the web server log data set mentioned in section III with different phase shifts, namely, each consumer is shifted one $M^{th}$ further into the dataset, where $M$ is the number of consumers. The reason for this is to create more variation among the producers and their production rates.

We evaluate the multiple producer-consumer version of $4$ implementations discussed in Section III: Mutex, Sem, BP, and our proposed algorithm in Section V, periodic batch processing with latching (PBPL). We chose Mutex and Sem because they are popular implementations and BP because it showed the best performance in our study in Section III.

Furthermore, we experiment with $2$, $5$, and $10$ producer-consumer pairs. For the batch processing based tests (BP and PBPL), we experiment with three different buffer sizes: $25$, $50$, and $100$.

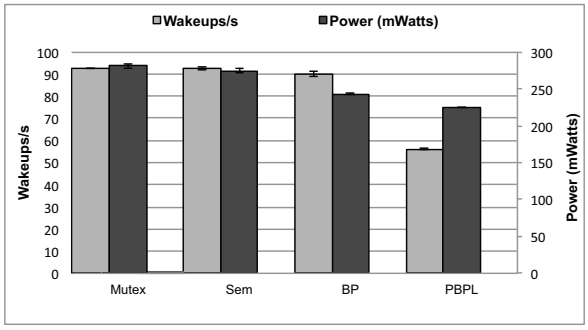



Fig. 9. A plot of wakeups/s versus power (mWatts) for all four implementations running 5 consumers.

### B. Experimental Metrics

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

- **Power.** The power consumption in milliwatts. This is not the total consumption of the system, it is, however, the increase in power consumption measured upon executing the experiment.
- **Wakeups/s.** The number of wakeups/sec measured by PowerTop.
- **Upper bound wakeups.** The number of wakeups we estimate internally in the batch processing based implementations.
- **Average buffer size.** This metric is the average buffer size in batch processing-based implementations. Recall that local buffers may change in PBPL when dynamic buffer resizing is operating.
- **Number of buffer overflows.** The number of buffer overflows in the batch processing based implementations.

### C. Experimental Results

Figure 9 shows the average power consumption as well as the average wakeups/s of all four implementations when the number of consumers is 5 and the buffer size is 25. As the graph demonstrates, wakeups/s is directly correlated with power consumption. PBPL offers the lowest power consumption of all four implementations, mostly due to its lower number of wakeups/s. Compared to Mutex, PBPL lowers the number of wakeups/s by 39.5%, and lowers power consumption by 20%. Compared to simple batch processing, it lowers the number of wakeups/s by 37.8% and lowers power consumption by 7.4%. When applied system-wide, this improvement becomes a major contributor to power savings.

An interesting observation is that although the reduction in wakeups is dramatic, it does not necessarily translate into the same order of reduction in power consumption. We attribute this to the background processes running on the system. There are multiple kernel processes executing including drivers, schedulers, timers, and other kernel daemons. Any processing caused by these processes affects power consumption. This shows that the power saving achieved from optimizing an application can always be potentially diminished by background

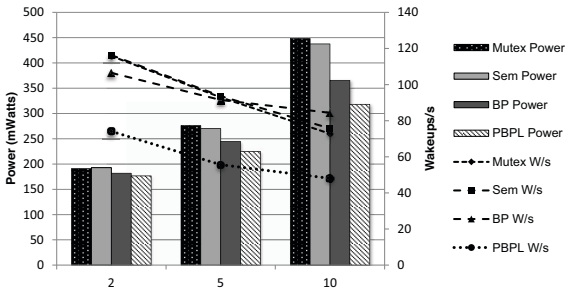



Fig. 10. A plot of wakeups/s versus power (mWatts) for all four implementations when the number of consumers is 2, 5, and 10.

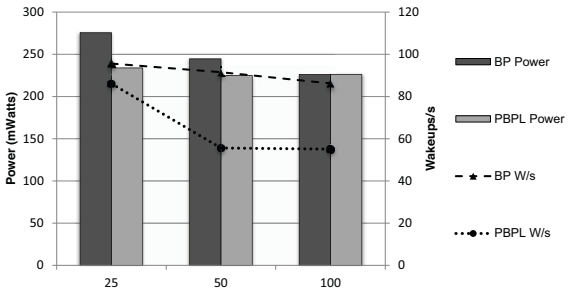



Fig. 11. A plot of wakeups/s versus power (mWatts) for BP and PBPL when the buffer size is 25, 50, and 100.

processes. However, even in these circumstances, we show that PBPL improves power consumption. Also, generalizing the approach in PBPL to be implemented system-wide would significantly impact total power consumption, since the producer-consumer problem is a fundamental problem that can be used to model many software processes.

Figure 10 shows the effect of changing the number of consumers on the power and wakeups/s of all four implementations. The buffer size for this set of experiments is 25. As can be seen, power consumption increases consistently with increasing the number of consumers, which is an expected result since the amount of work done is increased. However, the wakeups/s consistently decrease at the higher consumer count. This is due to the fact that CPU becomes more busy at a higher number of consumers, rendering it less idle, and, hence, less wakeups. Observe that in this figure as the number of consumers increase, the gap between the implementations increases too. For instance, the improvement of PBPL over Mutex is 7.5%, 20%, and 30% for consumer counts 2, 5, and 10, respectively. This shows that PBPL scales with a larger number of consumers, and in fact, performs better. Intuitively, this is a predictable result since PBPL is based on latching consumer invocations, and thus, it prospers when there are more consumers and more possibilities for latching.

Figure 11 shows the effect of changing the buffer size on power consumption and wakeups in BP and PBPL. As shown in the figure, increasing the buffer size causes a decrease in power consumption as well as wakeups, which is expected due to the ability of both implementations to buffer more items and

thus, wake up less. The gap between PBPL and BP decreases as the buffer size increases. This is due to the saturation of these implementations at a higher buffer size, rendering them more similar in their operation with only minor differences in the number of wakeups.

Our PBPL implementation counts internally a high level upper bound on the number of scheduled wakeups during its run, as well as the number of unscheduled wakeups (buffer overflows). For BP, every wakeup, or in other words every consumer invocation, is essentially a buffer overflow. On average, PBPL scores $5160$ scheduled wakeups, and $1626$ buffer overflows. In comparison, BP scores $9290$ buffer overflows. This amounts to a $25\%$ decrease in total wakeups, and an overflow conversion percentage of $82.5\%$.

The average buffer size in PBPL fluctuates depending on the rate of incoming data items. Although a buffer of size $50$ is allocated for each consumer, PBPL uses on average only $43$ buffer locations. This is achieved by dynamic resizing, causing the entire setup to use memory more efficiently. The unused space in the buffer is granted to consumers suffering from a high production rate, so that they can maintain their latching duties.

## VII. Related Work

Research in scheduling with energy efficiency as an objective demonstrates interesting results as in [1], [2], [6]. In these papers, the general approach is the analytical construction of an algorithm that provides an improved competitive ratio. This work is interesting in constructing a more effective dynamic power manager.

The work presented in [8], [13] is closer to the work presented in this paper in the sense that prediction is used to optimize the dynamic power manager. The concept of grouping events is presented in [3] among others. This paper approaches power efficiency from the perspective of improving the power profile of a fundamental concurrency problem.

The work in [9] presents PowerNap: an energy efficiency approach that targets minimizing idle power and transition time. The objectives for this paper are based on [9], however this paper attempts to contruct autonomous consumers capable of optimizing their behavior to decrease overall system power consumption.

## VIII. Conclusion

In this paper, we proposed a novel power-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 power. Consumers make prediction about the rate of incoming produced data items and group themselves together, so that the number of CPU wakeups is minimal.

We validated the effectiveness and efficiency of our algorithm by conducting a set of experiments. We observed that our algorithm can lower power consumption by up to $30\%$ compared to popular mutex and semaphore-based implementations. In fact, it provides up to $13\%$ improvement over the most power efficient implementation in our study. 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 currently working on 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 power, memory, CPU overhead, throughput, timing, constraints, etc., need to be taken into account simultaneously.

## IX. Acknowledgments

This research was supported in part by NSERC Discovery Grant 418396-2012, NSERC Strategic Grant 430575-2012, NSERC DG 357121-2008, ORF-RE03-045, ORF-RE04-036, ORF-RE04-039, CFI 20314, CMC, and the industrial partners associated with these projects.

## References

[1] Susanne Albers. Energy-efficient algorithms. *Communications of the ACM*, 53(5):86–96, 2010.

[2] Susanne Albers and Antonios Antoniadis. Race to idle: new algorithms for speed scaling with a sleep state. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1266–1285. SIAM, 2012.

[3] Hrishikesh Amur, Ripal Nathuji, Mrinmoy Ghosh, Karsten Schwan, and Hsien-Hsin S Lee. Idlepower: Application-aware management of processor idle states. *Proceedings of MMCS, in conjunction with HPDC*, 8, 2008.

[4] Martin Arlitt and Tai Jin. 1998 world cup web site access logs, 1998.

[5] L. A. Barroso and U. Hölzle. The case for energy-proportional computing. *IEEE Computers*, 40(12):33–37, 2007.

[6] Jessica Chang, Harold N Gabow, and Samir Khuller. A model for minimizing active processor time. In *Algorithms–ESA 2012*, pages 289–300. Springer, 2012.

[7] R. Ge, X. Feng, and K. W. Cameron. Improvement of power-performance efficiency for high-end computing. In *Proceedings. 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, page 8, 2005.

[8] Chi-Hong Hwang and Allen C-H Wu. A predictive system shutdown method for energy saving of event-driven computation. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 5(2):226–241, 2000.

[9] David Meisner, Brian T Gold, and Thomas F Wenisch. Powernap: eliminating server idle power. In *ACM Sigplan Notices*, volume 44, pages 205–216. ACM, 2009.

[10] 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*, pages 61–75, 2005.

[11] 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)*, pages 66–77, 2006.

[12] 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.

[13] Hao Shen, Ying Tan, Jun Lu, Qing Wu, and Qinru Qiu. Achieving autonomous power management using reinforcement learning. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 18(2):24, 2013.