2015 IEEE 29th International Parallel and Distributed Processing Symposium

Modeling Energy Consumption of Lock-Free Queue Implementations

Aras Atalar[†], Anders Gidenstam[†‡], Paul Renaud-Goud[†] and Philippas Tsigas[†]
[†]Chalmers University of Technology, 412 58 Göteborg; Email: name.surname@chalmers.se
[‡]University of Borås, 501 90 Borås; Email: name.surname@hb.se

***Abstract*—This paper considers the problem of modeling the energy behavior of lock-free concurrent queue data structures. Our main contribution is a way to model the energy behavior of lock-free queue implementations and parallel applications that use them. Focusing on steady state behavior we decompose energy behavior into throughput and power dissipation which can be modeled separately and later recombined into several useful metrics, such as energy per operation. Based on our models, instantiated from synthetic benchmark data, and using only a small amount of additional application specific information, energy and throughput predictions can be made for parallel applications that use the respective data structure implementation. To model throughput we propose a generic model for lock-free queue throughput behavior, based on a combination of the dequeuers' throughput and enqueuers' throughput. To model power dissipation we commonly split the contributions from the various computer components into static, activation and dynamic parts, where only the dynamic part depends on the actual instructions being executed. To instantiate the models a synthetic benchmark explores each queue implementation over the dimensions of processor frequency and number of threads. Finally, we show how to make predictions of application throughput and power dissipation for a parallel application using a lock-free queue requiring only a limited amount of information about the application work done between queue operations. Our case study on a Mandelbrot application shows convincing prediction results.[1]**

***Index Terms*—lock-free; analysis; modeling; energy; power; throughput; queue; concurrent data structures**

I. Introduction

Lock-free implementations of data structures is a scalable approach for designing concurrent data structures. Lock-free data structures offer high concurrency and immunity to deadlocks and convoying, in contrast to their blocking counterparts. Concurrent FIFO queue data structures are fundamental data structures that are key components in applications, algorithms, run-time and operating systems. The producer/consumer pattern, *e.g.*, is a common approach to parallelizing applications where threads act as either producers or consumers and synchronize and stream data items between them using a shared collection. A concurrent queue, *a.k.a.* shared "first-in, first-out" or FIFO buffer, is a shared collection of elements which supports at least the basic operations Enqueue (adds an element) and Dequeue (removes the oldest element). Dequeue returns the element removed or, if the queue is empty, NULL. A large number of lock-free (and wait-free) queue implementations have appeared in the literature, *e.g.* [1]–[6] being some of the most influential or most efficient results. Each implementation of a lock-free queue has obviously its strong and weak points so the impact on performance and energy when choosing one particular implementation for any given situation may not be obvious.

As the number of known implementations of lock-free concurrent queues is growing, it is of great interest to describe a framework within which the different implementations can be ranked, according to the parameters that characterize the situation. A brute force approach could achieve this by running the implementations on hand on the whole domain of study, gathering and comparing measurements. This would yield high accuracy, but at a tremendous cost, since the domain is likely to be large. Additionally, it would only bring a limited understanding on the phenomena that drive the behavior of the queue implementations. Therefore, we propose generic models for predicting the behavior of lock-free queues under steady state usage. The models are instantiated for the queue implementations and machine on hand using empirical data from a limited number of points in the domain.

The implementations can be ranked according to a plethora of metrics. Traditionally, performance in terms of throughput has been the main metric. Furthermore, the notion of energy efficiency has now extended into every nook and cranny of Information Technology, at any scale, from the Exascale machines that need huge improvements in terms of power dissipation to be feasible [7], to the small electronic devices where the battery lifetime is a critical issue.

We decompose the energy behavior of queues, and subsequently applications, into two components: (i) throughput and (ii) power dissipation. We model these components separately. The predicted throughput and power dissipation can be recombined into the energy-efficiency metric energy per queue operation, which is the ratio between power dissipation and queue throughput. When modeling an application, this metric can be extended to energy per unit of application work. Further, plotting energy per operation or unit of work according to throughput allows exploration of the Pareto-optimal frontier of the energy−performance bi-criteria optimization problem for the queues or the application.

Lock-free queue data structures generally offer disjoint-access parallelism: enqueuers and dequeuers modify only their respective ends of the queue, and compete mostly with

[1]The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2013-2016) under grant agreement 611183 (EXCESS Project, www.excess-project.eu).

1530-2075/15 $31.00 © 2015 IEEE
DOI 10.1109/IPDPS.2015.31

operations of the same kind. Nonetheless, when the queue is close to empty, both ends point to the same part of the queue, then enqueue and dequeue operations have to be synchronized, and every operation impacts the behavior of any other.

Concerning the queue as a whole, a successful event can be seen as the dequeue of a non-NULL item, since this event implies that the item has been enqueued and dequeued. Also, the throughput of the queue is naturally defined as the number of such events per unit of time, which is a meaningful performance criterion for queues.

In this work, we focus on queues that are in a steady state, *i.e.* such that the rate of each operation attempt is constant. Then, the throughput $\mathcal{T}$ of the queue is the minimum between the throughput of all dequeues $\mathcal{T}_d$, even those returning NULL, and throughput of enqueues $\mathcal{T}_e$. Indeed, if $\mathcal{T}_e > \mathcal{T}_d$, then the queue grows and the throughput is determined by the dequeuers, which cannot obtain any NULL items; and if $\mathcal{T}_e \leq \mathcal{T}_d$, then the queue is mostly empty and NULL items are dequeued, but the throughput is determined by the enqueuers.

Despite this decomposition, enqueuers' and dequeuers' throughput are still correlated when the queue is mostly empty. In addition, the interactions between them are rather asymmetric, as in broad terms, an enqueue can be delayed by any concurrent dequeue, while for a dequeue, concurrent enqueues will cease to disturb it as they move away from the dequeue end.

Based on these facts, we decorrelate the throughput into several uncorrelated and basic throughputs, and reconstitute the main throughput by combining them. Among the advantages of this process, we earn a better understanding of the performance (as the basic throughputs are meaningful), and we reduce the number of measurements needed to instantiate the model on the whole domain of study.

The domain of study that we envision here can be viewed as the Cartesian product of four sets: (i) number of threads accessing the queue, (ii) CPU frequencies, (iii) a range of dequeue access rates, (iv) a range of enqueue access rate. The cardinality of the first two sets is at most a few tens, while the last two are continuous sets that are not even bounded. In this paper, thanks to the removal of the dependencies between throughputs, we are able to instantiate the model with only a few data points, while the model covers the whole intervals.

Finally, this decomposition also eases the study of power dissipation, where we reuse the same ideas as in the throughput estimation part.

The rest of the paper is organized as follows. Section II discusses related work. Section III introduces our modeling framework for lock-free concurrent queues. Section IV describes how the throughput of lock-free concurrent queues is modeled, while Section V describes how the power dissipation is modeled. In Section VI we develop a method to model parallel applications using the queue models and apply it to an application for computing the Mandelbrot set. Finally, Section VII concludes this paper.

```
while ! done do
    el ← Parallel_Work(pw_e);
    Enqueue(el);
end
```
Procedure Enqueuer

```
while ! done do
    el ← Dequeue();
    Parallel_Work(pw_d);
end
```
Procedure Dequeuer

Fig. 1: Thread procedures

II. Related work

Hunt *et al.* [8] measured the performance and energy use of lock-free and lock-based implementations of FIFO queues, double-ended queues and sorted singly linked lists. The results from the lock-free and lock-based implementations are compared and also analyzed using captured hardware performance counters, *e.g.* instruction count, user/system time, L1 cache miss ratio and branch misprediction rate. Gautham *et al.* [9] compared the performance and energy use of locks and software transactional memory in benchmarks from the STAMP benchmark suite.

A variety of models have been proposed to estimate power dissipation, based on different approaches. PMC (Performance Monitoring Counters) based power models, build upon event selection and statistical correlation, draw considerable amount of attention. Using this approach, Contreras *et al.* [10] estimated CPU and memory power. Wang *et al.* [11] provided a two level power model for multiprocessors, which uses frequency and IPC (Instructions Per Cycle) as the only PMC event. Isci *et al.* [12] described a technique to estimate per-component power dissipation for CPU using PMCs and used this to determine phases of a program. Tiwari *et al.* [13] created an instruction level power model. They determined a base cost for each instruction type with micro-benchmarks and tried to clarify the inter-instruction impacts to estimate power dissipation of compositions. Ge and Cameron [14] provided a power-aware speedup model. They decompose the program into phases according to the degree of available parallelism and on/off-chip access ratios that is used to capture the impact of frequency scaling and process count. Choi *et al.* [15] introduced a roofline model which is parameterized with the maximum throughputs, operation energy and power cap values. They bound the throughput with the power cap, since energy consumption per unit of time depends on throughput, and extract the parameters' values using regression.

As seen above there exist some empirical studies on energy/power consumption of lock-free data structures and a huge variety of power models but we are not aware of any energy model targeting lock-free data-structures. In this study, we aim to begin filling this gap by providing a detailed analysis of power and performance of lock-free queues.

III. Framework

A. Synthetic Benchmark

1) Skeleton: We run the synthetic benchmark composed of the two functions described in Figure 1, starting with an empty queue. Half of the threads are assigned to be enqueuers while the remaining ones are dequeuers. We disable logical

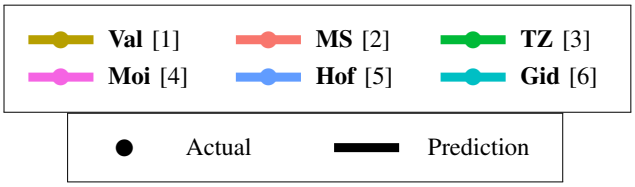

Fig. 2: Key legend of the graphs

cores (hyper-threading) and map different threads into different cores, also the number of threads never exceeds the number of cores. In addition, the mapping is done in the following way: when adding an enqueuer/dequeuer pair, they are both mapped on the most filled but non-full socket.

The parallel sections (Parallel_Work) shall be seen as a processing activity, pre-processing for the enqueuers before they enqueue an item, and post-processing on an item from the queue for the dequeuers. We assume that memory accesses in the parallel sections are negligible, and represent the parallel sections as sequences of bunches of *pause* instructions in the benchmark; we note pw_e (resp. pw_d) the number of bunches of 90 *pause*s (which corresponds to 1000 cycles) that compose the parallel work in the enqueuer (resp. dequeuer).

From a high-level perspective, Enqueue and Dequeue operations follow a retry loop pattern: a thread reads an access point to the data structure, works locally with this view of the data structure, possibly performs memory management actions and prepares the new desired value as an access point of the data structure. Finally, it atomically tries to perform the change through a call to the *Compare-and-Swap* primitive. If it succeeds, *i.e.* if the access point has not been changed by another thread between the first read and the *Compare-and-Swap*, then it goes to the next parallel section, otherwise it repeats the process.

2) Queue Implementations: We study some of the most well-known and studied lock-free and linearizable queues in the literature, as implemented in NOBLE [16]. The legend depicted in Figure 2 will be used throughout the paper. The aim of this work is still to predict the behavior of any lock-free queue algorithm and not only the ones mentioned above. These algorithms are used to validate the model that we present in the following sections.

B. General Power Model

The power is split into three elements: the *static* part is the cost of turning the machine on, the *activation* part incorporates a fixed cost for each socket and each core in use, and the *dynamic* part is a supplementary cost that depends on the running application.

In accordance with the RAPL energy counters [17]–[19], we further decompose each part per-component, for memory, CPU, and *uncore* (denoted by a superscript M, C and U, respectively):

$$P = \sum_{X \in \{M,C,U\}} \left(P^{(stat,X)} + P^{(active,X)} + P^{(dyn,X)} \right).$$

We assume that we already know the platform characteristics, *i.e.* all static and active powers (they can be obtained as explained for instance in the companion research report [20]), and we try to find the application-specific dynamic powers. In order to keep the formulas readable, in the following, we denote by $P^{(X)}$ the dynamic power $P^{(dyn,X)}$.

C. Notations and Setting

We denote by n the number of running threads that call the same operation, and by f the clock frequency of the cores (we only consider the case where all cores share the same clock frequency).

We recall that pw_e (resp. pw_d) is the amount of work in the parallel section of an enqueuer (resp. dequeuer), as the number of bunches of 90 *pause*s. For a given queue implementation, we denote by cw_e (resp. cw_d) the amount of work in one try of the retry loop of the Enqueue (resp. Dequeue) operation. Associated with these amounts of work, we define, for $o \in \{d, e\}$, the average execution time of the parallel section (resp. the retry loop and a single try of the retry loop) related to operation o as $\mathrm{t}(PS_o)$ (resp. $\mathrm{t}(RL_o)$ and $\mathrm{t}(SL_o)$).

In the same way, for $o \in \{d, e\}$, we denote by $P_o^{(C)}$ (resp. $P_{o,PS}^{(C)}$ and $P_{o,RL}^{(C)}$) the dynamic CPU power dissipated by component X in (resp. the parallel section related to and the retry loop related to) operation o.

Finally, for $o \in \{d, e\}$, we denote by r_o the ratio of the time that a thread spends in the retry loop, while it is associated with operation o.

In Sections IV and V, in order to keep expressions as simple as possible, we define one unit of time as λ sec, where λ is the execution time of $90 \times f$ *pause*s (as the *pause* instructions are perfectly scalable with clock frequency, λ is constant). Throughput is expressed in number of operations per unit of time, *i.e.* per λ secs. Finally, we derive the power in Watts.

All experiments and their underlying predictions are done on a platform composed of a dual-socket Intel® Xeon® processor, with eight cores per socket. The sizes of L3, L2 and L1 caches are 25 MB, 256 kB and 32 kB, respectively.

We run the implementations at the two extreme frequencies 1.2 GHz and 3.4 GHz, for all possible even total numbers of threads, from 2 to 16, *i.e.* for $n \in \{1, \ldots, 8\}$.

IV. Throughput Estimation

A. Throughput Decomposition Principles

We recall that the throughput of the queue is defined as:

$$\mathcal{T} = \min\left(\mathcal{T}_e, \mathcal{T}_d\right),$$

where $\mathcal{T}_e$ and $\mathcal{T}_d$ are the enqueuers' and dequeuers' throughput, respectively.

As we are in steady state, one operation o is performed every $\mathrm{t}(PS_o) + \mathrm{t}(RL_o)$ unit of time by each thread, and n threads attempt to concurrently execute o, hence the general expression of the throughput $\mathcal{T}_o$:

$$\mathcal{T}_o = \frac{n}{\mathrm{t}(PS_o) + \mathrm{t}(RL_o)}.$$

We have seen that the parallel sections of the benchmark are full of *pause*s, thus the time $\mathrm{t}(PS_o)$ spent in a given parallel section is straightforwardly given by $\mathrm{t}(PS_o) = pw_o/f$. The execution time of dequeue and enqueue operations is more problematic, for two main reasons. *Primo*, because of the lock-free nature of the implementations. As the number of retries is unknown, the time spent in the function call is not trivially computable. *Secundo*, when the activity on the queue is high, the threads compete for accessing a shared data, and they stall before actually being able to access the data. We name this as the *expansion*, as it leads to an increase in the execution time of a single try of the retry loop.

The contention on the queue is twofold. At any time, and even if it could be negligible, threads that perform the same operation disturb each other, since they try to access the same shared data. In addition, when the queue is mostly empty, enqueuers and dequeuers try to access the same data, then interference occurs; enqueuers make dequeuers stall and *vice versa*. We call the former case *intra-contention*, and the latter one *inter-contention*.

As expected, we have noticed a marked difference between the execution time of a dequeue operation returning NULL and one that returns a queue item, *i.e.* whether the queue was empty or contained at least one item. That is why we decompose $\mathcal{T}_d$ into throughput of dequeue on empty queue $\mathcal{T}_d^{(+)}$ (that returns a NULL item), and dequeue on non-empty queue $\mathcal{T}_d^{(-)}$ (that does not return NULL).

Further, the impact of inter-contention on dequeue operations is negligible compared to the impact of the queue being empty; therefore we ignore inter-contention for dequeues.

In contrast, the queue being empty does not notably change the execution time of the enqueue operation, while dequeue operations can impact the behavior of concurrent enqueue operations greatly when the queue is close to empty. Hence, we split $\mathcal{T}_e$ into the enqueue throughput $\mathcal{T}_e^{(+)}$ when the queue is not inter-contented, and the enqueue throughput $\mathcal{T}_e^{(-)}$ when the queue experiences the maximum possible inter-contention.

These basic throughputs fulfill the two following inequalities: $\mathcal{T}_d^{(+)} \geq \mathcal{T}_d^{(-)}$ and $\mathcal{T}_e^{(+)} \geq \mathcal{T}_e^{(-)}$.

Thanks to this separation into the four basic throughput cases $\mathcal{T}_d^{(+)}$, $\mathcal{T}_d^{(-)}$, $\mathcal{T}_e^{(+)}$ and $\mathcal{T}_e^{(-)}$, we earn a better understanding of the factors that influence the general throughput, and we deinterlace their dependencies, which dramatically decreases the number of points in the parallel section sizes set where we need to take measurements for our modeling. More precisely, by construction, $\mathcal{T}_d^{(+)}$ and $\mathcal{T}_d^{(-)}$ do not indeed depend on pw_e, while $\mathcal{T}_e^{(+)}$ and $\mathcal{T}_e^{(-)}$ do not depend on pw_d. Nonetheless $\mathcal{T}_d$ (resp. $\mathcal{T}_e$) is defined as a barycenter between $\mathcal{T}_d^{(+)}$ and $\mathcal{T}_d^{(-)}$ (resp. $\mathcal{T}_e^{(-)}$ and $\mathcal{T}_e^{(+)}$), whose weights depend on both pw_d and pw_e.

In Section IV-B, we describe the basic throughputs, we combine them in Section IV-C, then we explain how to instantiate the parameters of the model in Section IV-D, and finally exhibit results in Section IV-E.

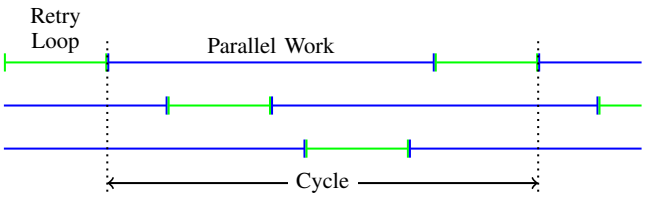

Fig. 3: Cyclic execution under low intra-contention

B. Basic Throughputs

We aim in this section at estimating the throughput $\mathcal{T}_o^{(b)}$ of one of the basic operations described in the previous subsection, where $o \in \{e, d\}$ and $b \in \{+, -\}$. We assume that $\mathcal{T}_o^{(b)}$ depends only on pw_o, in addition to the tacit dependencies on the clock frequency, number of threads and queue implementation. We denote by $cw_o^{(b)}$ the amount of work in a single try of the retry loop related to operation o in case b when the queue is not intra-contented.

1) Low Intra-Contention: We study in this section the low intra-contention case, *i.e.* when (i) the threads do not suffer from expansion due to threads that perform the same operation, and (ii) a success is obtained with a single try of the retry loop. As it appears in Figure 3, we have a cyclic execution, and the length of the shortest cycle is $\mathrm{t}(PS_o)+\mathrm{t}\left(SL_o^{(b)}\right)$. Within each cycle, every thread performs exactly one successful operation, thus the throughput is easy to compute:

$$\mathcal{T}_o^{(b)} = \frac{n}{\mathrm{t}(PS_o) + \mathrm{t}\left(SL_o^{(b)}\right)} = \frac{nf}{pw_o + cw_o^{(b)}}. \qquad (1)$$

2) High Intra-Contention: As explained in Section IV-A, in this case, the direct evaluation of the execution time of a retry loop is more complex, but we have experimentally observed that the throughput is approximately linear with the expected number of threads that are in the retry loop at a given time. In addition, this expected number is almost proportional to the amount of work in the parallel section. As a result, a good approximation of the throughput, in high intra-contention cases, is a function that is linear with the amount of work in the pw_o.

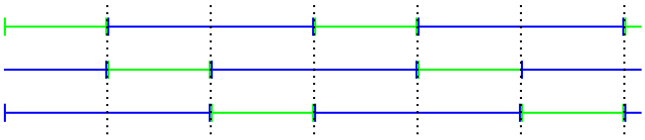

Fig. 4: Intra-contention frontier

3) Frontier: We now have to estimate whether the queue is highly intra-contended.

There exists a simple lower bound of the amount of work in the parallel section, such that there exists an execution where the threads are never failing in their retry loop. We plot in Figure 4 an ideal execution with $n = 3$ threads and

$\mathrm{t}(PS_o) = (n-1) \times \mathrm{t}\left(SL_o^{(b)}\right)$. In this execution, all threads always succeed at their first try in the retry loop. Nevertheless, if we shorten the parallel section, then there is not enough parallel potential any more, and the threads will start to fail: the queue leaves the low intra-contention state.

In practice, this lower bound ($\mathrm{t}(PS_o) = (n-1) \times \mathrm{t}\left(SL_o^{(b)}\right)$) is actually a good approximation for the critical point where the queue switches its state.

C. Combining Basic Throughputs

We are given parallel sections sizes, and show how to link the throughput of the four basic operations, with the dequeuers' and enqueuers' throughput. There are two possible states for the queue: either it is mostly empty (*i.e.* some NULL items are dequeued), or it gets larger and larger.

In the first case, some of the dequeues will occur on an empty queue. In 1 unit of time, $\mathcal{T}_e$ items are enqueued. These items are dequeued in $\mathcal{T}_e/\mathcal{T}_d^{(-)}$ units of time (the queue is non-empty while they are dequeued), which leads to a slack of $1-\mathcal{T}_e/\mathcal{T}_d^{(-)}$, where dequeues of NULL items can take place at a rate $\mathcal{T}_d^{(+)}$, hence the following throughput formula:

$$\mathcal{T}_d = \frac{\mathcal{T}_e}{\mathcal{T}_d^{(-)}} \times \mathcal{T}_d^{(-)} + \left(1 - \frac{\mathcal{T}_e}{\mathcal{T}_d^{(-)}}\right) \times \mathcal{T}_d^{(+)}. \tag{2}$$

Concerning the enqueuers, we use the same assumption on inter-contention as used on intra-contention in Section IV-B2, saying that the throughput is linear with the expected number of threads inside the retry loop. Here, the expected number of threads inside the dequeue operation is proportional to the ratio r_d of the time spent by one dequeuer in its dequeue operation. We do not know $\mathrm{t}(RL_d)$, but we know that in average, to complete a successful operation, a thread needs $\mathrm{t}(PS_d) + \mathrm{t}(RL_d)$ units of time, and among this time it will spend $\mathrm{t}(PS_d)$ in the parallel section. Therefore

$$r_d = 1 - \mathrm{t}(PS_d)/(\mathrm{t}(PS_d) + \mathrm{t}(RL_d)) = 1 - \frac{\mathcal{T}_d \times pw_d}{n \times f}.$$

The minimum intra-contention is reached when this ratio is 0, while the maximum is obtained when it is 1, thus:

$$\mathcal{T}_e = \frac{\mathcal{T}_d \times pw_d}{n \times f} \times \mathcal{T}_e^{(+)} + \left(1 - \frac{\mathcal{T}_d \times pw_d}{n \times f}\right) \times \mathcal{T}_e^{(-)}. \tag{3}$$

In the second case, enqueuers and dequeuers do not access to the same part of the queue, thus inter-contention does not take place, then $\mathcal{T}_e = \mathcal{T}_e^{(+)}$, and all dequeues return a non-NULL item, hence $\mathcal{T}_d = \mathcal{T}_d^{(-)}$.

The discrimination of these two cases is trivial when enqueuers' and dequeuers' throughput are given: the queue is in the first state (mostly empty) if and only if $\mathcal{T}_e \leq \mathcal{T}_d$.

Reversely, if we know the four basic throughputs, and aim at reconstituting the dequeuers' and enqueuers' throughput, several solutions could be consistent. We show in the companion research report [20] that, given $\left(\mathcal{T}_d^{(+)}, \mathcal{T}_d^{(-)}, \mathcal{T}_e^{(+)}, \mathcal{T}_e^{(-)}\right)$,

- there exists a solution $(\mathcal{T}_d, \mathcal{T}_e)$ with a growing queue if and only if $\mathcal{T}_e^{(+)} > \mathcal{T}_d^{(-)}$. In addition, such a solution with a growing queue is such that $\mathcal{T}_e = \mathcal{T}_e^{(+)}$ and $\mathcal{T}_d = \mathcal{T}_d^{(-)}$.
- there exists a solution $(\mathcal{T}_d, \mathcal{T}_e)$ with a mostly empty queue if and only if

$$\frac{\mathcal{T}_e^{(-)}}{\mathcal{T}_d^{(-)}} \leq 1 - \frac{pw_d}{n \times f}\left(\mathcal{T}_e^{(+)} - \mathcal{T}_e^{(-)}\right). \tag{4}$$

  In addition, such a solution with a mostly empty queue is determined in a unique way by Equations 3 and 2.
- there exists at least one solution $(\mathcal{T}_d, \mathcal{T}_e)$.

One can notice that if $\mathcal{T}_e^{(+)} > \mathcal{T}_d^{(-)}$ and Inequality 4 are fulfilled and the queue could be either mostly empty or growing. In this case, we choose, for each operation, the mean of the two solutions, in order to minimize the discontinuities.

D. Instantiating the Throughput Model

We recall that, for all o and b, $\mathcal{T}_o^{(b)}$ depends only on pw_o, while $\mathcal{T}_e$ and $\mathcal{T}_d$ depend on both pw_d and pw_e. We denote now by $\mathcal{T}_d(pw_d, pw_e)$ (resp. $\mathcal{T}_e(pw_d, pw_e)$) the dequeuers' (resp. enqueuers') throughput as the amount of work in the parallel section of the dequeuers is pw_d and enqueuers' one is pw_e. The estimate of a value is denoted by a hat on top, while the measured value does not wear the hat.

Let $p_s = 1$, $p_m = 20$ and $p_b = 1000$ be three distinctive amounts of work, that corresponds to different states of the execution. If $pw_o = p_b$, we can neglect the impact of operation o on the queue, $pw_o = p_m$ is a low intra-contention case since the non-expanded critical sections are experimentally less than 2 units of time, and $pw_o = p_s$ corresponds to a highly inter- or intra-contention case. We note the we cannot use a 0 size as amount of work since it leads to undesirable results due to the back-to-back effect (a thread does not allow other threads to access the queue for several consecutive iterations).

1) Low Intra-Contention: The basic throughputs that are not intra-contented can be spawned from $cw_o^{(b)}$, which we try to estimate here. We pick four points where the basic throughputs are easy to approximate. We have $\mathcal{T}_d(p_m, p_s) < \mathcal{T}_e(p_m, p_s)$, as the order of magnitude of the amounts of work in the retry loops is less than a few units. For the same reason, at this point, we are in low intra-contention from the dequeuers' point of view. Altogether,

$$\mathcal{T}_d(p_m, p_s) = \mathcal{T}_d^{(-)}(p_m) = \frac{n \times f}{p_m + cw_d^{(-)}}, \text{ hence}$$

$$\widehat{cw_d^{(-)}} = \frac{n \times f}{\mathcal{T}_d(p_m, p_s)} - p_m.$$

Then, according to Equation 2, we have

$$\frac{nf}{p_m + \widehat{cw_d^{(+)}}} = \mathcal{T}_d^{(+)}(p_m)$$

$$\frac{nf}{p_m + \widehat{cw_d^{(+)}}} = \frac{\mathcal{T}_d(p_m, p_b) - \mathcal{T}_e(p_m, p_b)}{1 - \frac{\left(p_m + \widehat{cw_d^{(-)}}\right) \times \mathcal{T}_e(p_m, p_b)}{n \times f}},$$

from which we can extract $\widehat{cw_d^{(+)}}$ since we know already $\widehat{cw_d^{(-)}}$.

In the same way, we can compute $\widehat{cw_e^{(+)}}$ then $\widehat{cw_e^{(-)}}$, by using (p_b, p_m) and (p_s, p_m).

2) High Intra-Contention: We aim here at estimating $\mathcal{T}_o^{(b)}$ on a high intra-contention point. $p_s = 1$ and $p_m = 20$ are such that $\mathcal{T}_d(p_s, p_m) \geq \mathcal{T}_e(p_s, p_m)$. According to Equation 2, we have

$$\mathcal{T}_d(p_s, p_m) = \mathcal{T}_e(p_s, p_m) + \left(1 - \frac{\mathcal{T}_e(p_s, p_m)}{\widehat{\mathcal{T}_d^{(-)}}(p_s)}\right) \times \widehat{\mathcal{T}_d^{(+)}}(p_s).$$

In addition, if $\mathcal{T}_d(p_s, p_s) \geq \mathcal{T}_e(p_s, p_s)$, then

$$\mathcal{T}_d(p_s, p_s) = \mathcal{T}_e(p_s, p_s) + \left(1 - \frac{\mathcal{T}_e(p_s, p_s)}{\widehat{\mathcal{T}_d^{(-)}}(p_s)}\right) \times \widehat{\mathcal{T}_d^{(+)}}(p_s),$$

otherwise, $\mathcal{T}_d(p_s, p_s) = \widehat{\mathcal{T}_d^{(-)}}(p_s)$. In both cases, we can find the two unknowns $\widehat{\mathcal{T}_d^{(-)}}(p_s)$ and $\widehat{\mathcal{T}_d^{(+)}}(p_s)$ thanks to the two equations.

This last point is also used in the same way for enqueuers: if $\mathcal{T}_d(p_s, p_s) \geq \mathcal{T}_e(p_s, p_s)$, then

$$\mathcal{T}_e(p_s, p_s) = \frac{\mathcal{T}_d(p_s, p_s) \times p_s}{n \times f} \times \widehat{\mathcal{T}_e^{(+)}}(p_s) + \left(1 - \frac{\mathcal{T}_d(p_s, p_s) \times p_s}{n \times f}\right) \times \widehat{\mathcal{T}_e^{(-)}}(p_s),$$

otherwise, $\mathcal{T}_e(p_s, p_s) = \widehat{\mathcal{T}_e^{(+)}}(p_s)$.

Like previously, we have $\mathcal{T}_d(p_m, p_s) < \mathcal{T}_e(p_m, p_s)$, hence $\widehat{\mathcal{T}_e^{(+)}}(p_s) = \mathcal{T}_e(p_m, p_s)$. This implies that in any cases we can compute $\widehat{\mathcal{T}_e^{(+)}}(p_s)$, but we do not have access to $\widehat{\mathcal{T}_e^{(-)}}(p_s)$ if $\mathcal{T}_d(p_s, p_s) < \mathcal{T}_e(p_s, p_s)$. In this case, the bottleneck of the queue is likely to be the dequeuers, hence we set the value $\widehat{\mathcal{T}_e^{(-)}}(p_s) = \widehat{\mathcal{T}_e^{(+)}}(p_s)$ by default.

All $\widehat{\mathcal{T}_o^{(b)}}$ are then obtained by joining $\widehat{\mathcal{T}_o^{(b)}}(p_s)$ to the leftmost point of the low intra-contention part:

$$\widehat{\mathcal{T}_o^{(b)}}(pw_o) = \begin{cases} \frac{\frac{f}{\widehat{cw_o^{(b)}}} - \widehat{\mathcal{T}_o^{(b)}}(p_s)}{(n-1)\widehat{cw_o^{(b)}} - p_s} \times (pw_o - p_s) + \widehat{\mathcal{T}_o^{(b)}}(p_s) \\ \qquad \text{if } pw_o \leq (n-1)\widehat{cw_o^{(b)}} \\ \frac{n \times f}{pw_o + \widehat{cw_o^{(b)}}} \quad \text{otherwise.} \end{cases}$$

Finally, dequeuers' and enqueuers' throughput are reconstituted as explained in Section IV-C: if Equation 4 is fullfilled, then they are computed through Equations 2 and 3 that can be rewritten as:

$$\begin{cases} \widehat{\mathcal{T}_d}(pw_d, pw_e) = \frac{\widehat{\mathcal{T}_d^{(+)}}(pw_d) + \widehat{\mathcal{T}_e^{(-)}}(pw_e)\left(1 - \frac{\widehat{\mathcal{T}_d^{(+)}}(pw_d)}{\widehat{\mathcal{T}_d^{(-)}}(pw_d)}\right)}{1 - \frac{pw_d}{nf}\left(\widehat{\mathcal{T}_e^{(+)}}(pw_e) - \widehat{\mathcal{T}_e^{(-)}}(pw_e)\right)\left(1 - \frac{\widehat{\mathcal{T}_d^{(+)}}(pw_d)}{\widehat{\mathcal{T}_d^{(-)}}(pw_d)}\right)} \\ \widehat{\mathcal{T}_e}(pw_d, pw_e) = \frac{\widehat{\mathcal{T}_d}(pw_d, pw_e) \times pw_d}{n \times f} \times \widehat{\mathcal{T}_e^{(+)}}(pw_e) + \left(1 - \frac{\widehat{\mathcal{T}_d}(pw_d, pw_e) \times pw_d}{n \times f}\right) \times \widehat{\mathcal{T}_e^{(-)}}(pw_e). \end{cases}$$

Otherwise, $\widehat{\mathcal{T}_d}(pw_d, pw_e) = \widehat{\mathcal{T}_d^{(-)}}(pw_d)$ and $\widehat{\mathcal{T}_e}(pw_d, pw_e) = \widehat{\mathcal{T}_e^{(+)}}(pw_e)$.

E. Results

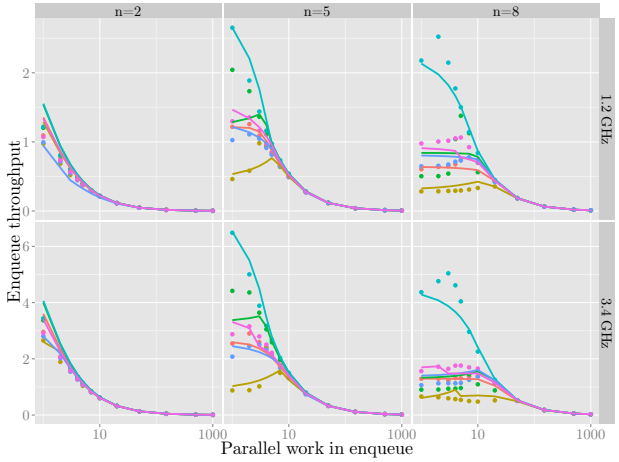

Fig. 5: Enqueue throughput with $pw_d = 7$

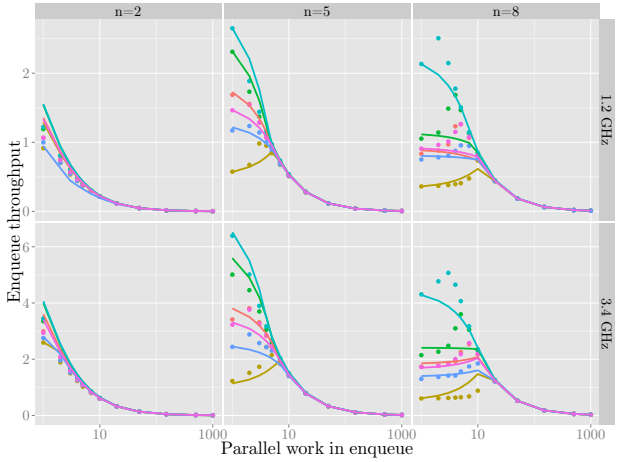

Fig. 6: Enqueue throughput with $pw_d = 50$

The throughput predictions are plotted in Figures 5 and 6 for the enqueuers, and in Figure 7 for the dequeuers (the legend is in Figure 2). Points are measurements, while lines

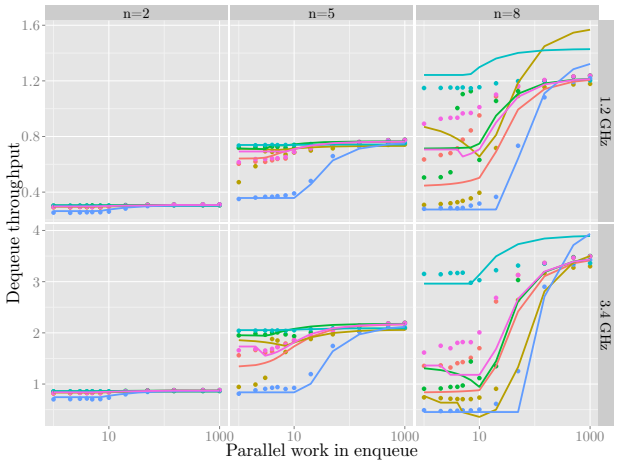

Fig. 7: Dequeue throughput with $pw_d = 7$

are predictions. We will follow this rule for all comparisons between prediction and measurement. In the actual execution, the queue goes through a transient state when the amount of work in the parallel section is near the critical point, but the prediction is not so far from the actual measurements, as illustrated in Figure 5 and 6. Under intra-contention, some of the curves get flat, since only one thread can be succeeding at the same time, according to the definition of the retry loop. Some curves even decrease because the successful one is stalled by other failing ones due to serialization of the atomic primitives, namely expansion. The slope presumably indicates the density of atomic primitives in retry loops which depends on the algorithm.

The comparison of Figures 5 and 6 illustrates the impact of inter-contention. A decrease of the highest point of $\mathcal{T}_e$, due to an increase of cw_e, can be observed for the more inter-contented case. When cw_e increases, some critical points shift slightly towards the right as the intra-contention starts with a larger pw_e. In Figure 7, decomposition of $\mathcal{T}_d$ is apparent. When enqueue rate is low, *i.e.* when pw_e is high, $\mathcal{T}_d$ is ruled by $\mathcal{T}_d^{(+)}$ due to majority of NULL dequeues, and it tends towards $\mathcal{T}_d^{(-)}$ when the enqueue rate increases.

Graphs on a wider set of parameters are available in the companion research report [20], in the form of animated figures.

V. Power Estimation

We recall that we are interested only in the dynamic powers as we assume that static and activation powers are known.

A. CPU Power

Firstly, as we map each thread on a dedicated core, there is no interference between the CPU power of different cores, so we can compute the dynamic power as

$$P^{(C)} = n \times P_e^{(C)} + n \times P_d^{(C)}. \tag{5}$$

Secondly, we assume that we can segment time and consider that, given a thread performing operation o, the power dissipated in the retry loop and the power dissipated in the parallel section are independent. There only remains to weight the previous powers by the time spent in each of these regions:

$$P_o^{(C)} = r_o \times P_{o,RL}^{(C)} + (1 - r_o) \times P_{o,PS}^{(C)}. \tag{6}$$

As shown in Section IV-C, the ratio can be obtained through

$$r_o = 1 - \frac{\mathcal{T}_o \times pw_o}{n \times f}. \tag{7}$$

Altogether, we obtain the final formula for dynamic CPU power

$$P^{(C)} = n \left(\sum_{o \in \{e,d\}} P_{o,RL}^{(C)} + \frac{\mathcal{T}_o \times pw_o \times \left(P_{o,PS}^{(C)} - P_{o,RL}^{(C)} \right)}{n \times f} \right) \tag{8}$$

B. Memory and Uncore Power

We have noticed in [20] that the dynamic memory power is proportional to the intensity (number of units of memory accessed per unit of time) of main memory accesses and remote accesses, when the threads read separate places of the memory.

Here, the data structure does not directly involve the main memory since we keep its size reasonably bounded (if the queue reaches the maximum size, we suspend the measurements, empty the queue, and resume), hence the power dissipation in memory is only due to remote accesses, which only appears as the threads are spread across sockets (*i.e.* when $n > 4$).

Moreover, as the parallel sections are full of *pause*s, communications can only take place in the retry loop, and there is no dynamic memory power dissipated in the parallel sections. Concerning the retry loops, we make the following assumption: the amount of data accessed per second in a retry loop depends on the implementation, but given an implementation, once a thread is in the retry loop, it will always try to access the same amount of data per second. When the queue is highly intra-contented, if a thread fails then it will retry and will access the data in the same way as in the previous try; and if there is expansion, then the thread will still try to access the data for the whole time it is in the retry loop.

In addition, the dequeuers (and the same line of reasoning holds for the enqueuers) tries here to access the same data. Therefore either memory requests are batched together when sent outside the socket, or the Home Agent keeps track of the previous requests. This implies that the number of threads attempting to access the data does not impact the dynamic memory power greatly when the rate of requests is high.

All things considered, as a thread working on operation o spends a fraction r_o of its time inside its retry loop, we obtain that the dynamic memory power dissipated in the retry loop is proportional to r_o (times the amount of data accessed per unit of time in the retry loop, which is a constant). Hence

$$P^{(M)} = r_e \times \rho_e^{(M)} + r_d \times \rho_d^{(M)}, \tag{9}$$

where $\rho_e^{(M)}$ and $\rho_d^{(M)}$ are constants.

The dynamic uncore power is computed exactly in the same way as the dynamic memory power.

C. Instantiating the Power Model

We use once again $p_s = 1$, $p_m = 20$ and $p_b = 1000$ as three distinctive amounts of work, that allows easy approximations for the power dissipation expressions.

We have seen that if $X \in \{M, U\}$, then $P^{(X)} = r_d \times \rho_d^{(X)} + r_e \times \rho_e^{(X)}$, which can be approximated at $(pw_d, pw_e) = (p_b, p_s)$ by $P^{(X)}(p_b, p_s) = r_e(p_s) \times \rho_e^{(X)}$, since r_d is then nearly 0. It implies that

$$\widehat{\rho_e^{(X)}} = \frac{P^{(X)}(p_b, p_s)}{1 - \frac{\mathcal{T}_e(p_b, p_s) \times p_s}{n \times f}}.$$

We obtain $\widehat{\rho_d^{(X)}}$ similarly at $(pw_d, pw_e) = (p_s, p_b)$.

Concerning the dynamic CPU power, we firstly estimate the power dissipated in the parallel sections. According to the implementation, the CPU power dissipated by the parallel section of enqueuers and dequeuers is the same for both, and this power does not depend on the amount of work. These restrictions are not a loss of generality, since the aim here is to study the queue implementations. It can then be estimated by using (p_b, p_b), where the ratios r_o can be considered as 0, which leads to

$$\widehat{P_{o,PS}^{(C)}} = \frac{P^{(C)}(p_b, p_b)}{2n}.$$

We reuse the point (p_b, p_s), where r_d is very close to 0, to derive that

$$P^{(C)} = n\left(r_e(p_s) \times \widehat{P_{e,RL}^{(C)}} + (1 - r_e(p_s))\widehat{P_{e,PS}^{(C)}}\right) + n\widehat{P_{d,PS}^{(C)}},$$

which is equivalent to

$$\widehat{P_{e,RL}^{(C)}} = \frac{P^{(C)}(p_b, p_s)}{n\left(1 - \frac{\mathcal{T}_e(p_b, p_s) p_s}{n \times f}\right)} - \left(\frac{2}{1 - \frac{\mathcal{T}_e(p_b, p_s) p_s}{n \times f}} - 1\right) \widehat{P_{o,PS}^{(C)}}$$

Once again, we obtain $\widehat{P_{d,RL}^{(C)}}$ with the same line of reasoning at $(pw_d, pw_e) = (p_s, p_b)$.

Finally, $\widehat{P^{(M)}}$ and $\widehat{P^{(U)}}$ (resp. $\widehat{P^{(C)}}$) are computed by using Equation 9 (resp. Equations 5 and 6), and the estimates of the ratios that are issued from Section IV.

D. Results

As the retry loop, which is particular to each implementation, is mainly composed of memory operations, the main difference between the various implementations in terms of power occurs in the dynamic memory power, which we represent in Figure 8 (legend is in Figure 2).

Overall, the prediction reacts correctly to the variations of parallel section sizes, and some specifics of the algorithms are caught, *e.g.* **Hof** detached from the others when $pw_e = 50$ or **Gid** mostly well-predicted both absolutely and relatively as the less power-dissipating implementation.

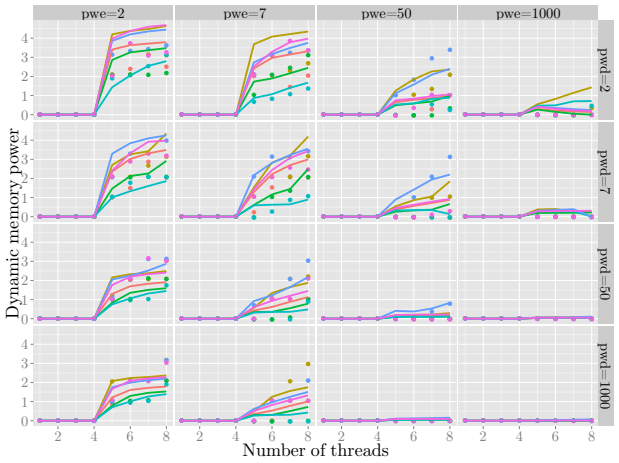

Fig. 8: Dynamic memory power at $f = 3.4\,\text{GHz}$

One can observe once again the asymmetry between enqueue and dequeue operations by comparing the power values at $(pw_d, pw_e) = (2, 1000)$ and $(1000, 2)$; this asymmetry is predicted by the model, with a lower impact though.

Other power comparisons can be found in the companion research report [20], along with the results about the last metric, namely energy per operation.

VI. Towards Realistic Applications: Mandelbrot Set Computation

The performance and energy behavior of an application using a lock-free queue depends on both the application specific code and the implementation of the data structure. For applications where the queue is used in a steady state manner, predictions can be made using the model instantiated with the synthetic benchmark, combined with information about the behavior of the application specific code. What is needed is:

- The size of the parallel work part of the application, both for enqueuers and dequeuers. These may be distributions rather than single values.
- The dynamic power for these parts (as it may differ from that of the parallel work in the synthetic benchmark).

A. Description of Mandelbrot Set Application

As a case-study we have used an existing application[2] that computes and renders an 8192×8192 pixel image of the Mandelbrot set [22] in parallel using the producer/consumer pattern. The program uses a concurrent queue to communicate between two major phases:

- Phase 1 consists of computing the number (with a maximum of 255) of iterations for a given set of points within a chosen region of the image. The results for each region together with its coordinates are then enqueued.
- Phase 2 consists of, for each region dequeued from the queue, computing the RGB values for each contained point and draw these pixels to the resulting image.

[2]Previously used for evaluation in [21].

Half of the threads perform phase 1 and the rest perform phase 2. The size of each square region is chosen to be one of 16×16, 4×4, or 2×2 pixels which also determines the amount of work to perform per queue operation and, hence, the level of contention. Similarly to the synthetic benchmark, the application uses a dense pinning strategy, pinning producer/consumer pairs to consecutive pairs of cores.

B. Mandelbrot Prediction

There are two main differences between the Mandelbrot application and the synthetic benchmark: (i) the instructions in the parallel section differ; and (ii) the size of the parallel section for producers varies in Mandelbrot.

Firstly, we need to measure the CPU power dissipation for Mandelbrot; we cannot expect to be able to predict the power dissipation of any application that uses a queue without having any knowledge about the power characteristics of the application. In contrast, memory power dissipation for the computation intensive Mandelbrot parallel section is negligible in comparison to queue operations; hence, the dynamic memory power that we have measured and extrapolated in the synthetic benchmark is unchanged.

Secondly, Mandelbrot provides a variety of producer parallel works. To deal with this, the pixel region is decomposed row-wise in an interleaved manner among producer threads. This decomposition leads to long enough execution intervals in which the parallel sections of the producer threads are similar and constant. This is due to the computationally expensive pixels belonging to the Mandelbrot set being concentrated together in the center of the domain and surrounded by cheaper pixels which diverge quickly. This characteristic is congruent with our model where the data structure is used in a steady state manner. Thus, predictions can be made using the instantiated model over a linear combination of execution intervals.

We measure the latency of the computation intensive producer and consumer parallel works for each frequency and contention level (2×2, 4×4, 16×16). For this process, we make use of CPUID, RDTSC and RDTSCP instructions as specified in [23]. The distribution of parallel works reveals that there are two main groups for producers, that corresponds to regions belonging to the Mandelbrot set or not. Concerning 2×2 contention, due to the wide distribution, we gather the parallel works into bins of width 10 pauses; the number of elements in the i^{th} bin is then denoted by $size^{(i)}$ and its average amount of work by $pw_e^{(i)}$. We scale the width of bins linearly with the area of the region for other contention levels. For the consumers, parallel works are similar for the whole execution.

To make predictions, we assume that all consumer/producer pair $(pw_d, pw_e^{(i)})$ is executed in a steady state during an interval of time. For each frequency, thread, algorithm and contention of interest, we obtain the throughput $\mathcal{T}^{(i)} = \mathcal{T}(pw_d, pw_e^{(i)})$ and the powers $P_i^{(X)} = P^{(X)}(pw_d, pw_e^{(i)})$ for this interval from the corresponding synthetic benchmark input. The only part of the model, instantiated with the synthetic benchmark that needs to be replaced by an application specific entry, is the dynamic CPU power parameter. Then, we combine intervals to obtain total execution time and average power dissipation. This accumulation strategy should be applied with care as the synthetic benchmark is based upon the steady state assumption. An interval which is assumed to take place with a mostly empty queue, could actually not be in this state due to leftover items from the previous interval. Although our model is capable of taking this initial state into consideration and provide metrics accordingly, we assume that each interval is independent. This approximation is reasonable since the consumer parallel work corresponds to the producer bin with one of smallest values, hence a mostly empty queue.

Note that we have implemented a constant back-off equivalent to the consumer parallel work, after dequeuing a NULL item instead of retrying immediately, because of several advantages. It cannot decrease the performance, since either the queue is growing, and then the back-off never takes place, or the queue is mostly empty, and then the producers are the bottleneck of the queue. Conversely, it can increase the performance by diminishing the queue contention. Those motivations drove the design of the synthetic benchmark, that we can accordingly reuse here.

For each frequency, thread, algorithm and contention configuration, execution time and power estimates for Mandelbrot application are obtained with the following equations:

$$Time_{total} = \sum_{i=1}^{BinCount} size^{(i)} \times \frac{\lambda}{\mathcal{T}^{(i)}}$$

$$P^{(X)} = \frac{\sum_{i=1}^{BinCount} (size^{(i)} \times \frac{\lambda}{\mathcal{T}^{(i)}}) \times P_i^{(X)}}{Time_{total}}$$

CPU power estimation is straightforward and memory power results are very similar to the synthetic benchmark in Figure 8, so we just present and discuss them in [20].

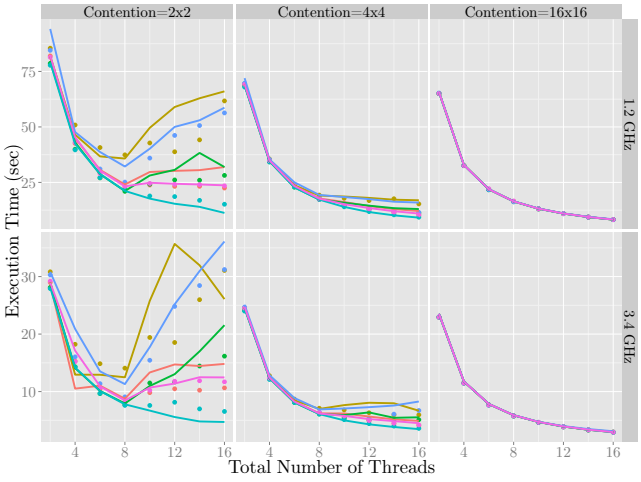

Fig. 9: Mandelbrot Execution Time

In Figure 9, execution time estimates catch the queue algorithm specific trend for high contention cases, which exhibit a more complicated behavior than the low contention cases. Also, they reveal the impact of different queue implementations to overall application performance, which does not appear under low contention. For the highest contention level with region size 2×2, an increasing trend in execution time is observed after 8 threads for many algorithms. The reason is the increasing latency of atomic synchronization primitives originating from two main sources: (i) inter-socket communication, which starts after 8 threads due to our pinning strategy, and (ii) the increasing serialization (expansion) probability for atomic primitives due to increasing number of threads that interfere in the retry loop. The ratio of atomic primitives and the size of queue operations show variations between algorithms which in turn leads to different behaviors. For the 4×4 contention case, the difference between algorithms can still be observed but the parallel sections are large enough to avoid interference in the retry loop. Therefore, execution time decreases with the increasing number of threads. The difference between algorithms is due to different queue operation sizes which loses its significance gradually with the decreasing contention level, as observed in low contention cases.

VII. Conclusion

In this paper we have:
(i) proposed models for predicting the throughput and power behavior of lock-free concurrent queues under steady state usage;
(ii) shown how these models can be instantiated for the queue implementations and machine on hand using 10 measurements per frequency and number of threads via a synthetic benchmark; and
(iii) demonstrated that the energy behavior of a parallel application that uses a lock-free queue in a steady state manner can be predicted using these models and only a small amount of queue-implementation-independent empirical information about the application.

As a future work, it would be of interest to study the strength of the model that has been presented here by testing it on other applications, in particular on more memory-intensive ones.

Furthermore, the model can hopefully be extended to several directions. While staying focused on the queue data structure, lock-based implementations may be included, and behave in a similar way as their lock-free counterparts. To conclude, it would be interesting to generalize the model to other data types.

References

[1] J. D. Valois, "Implementing Lock-Free Queues," in *Proceedings of International Conference on Parallel and Distributed Systems (ICPADS)*, December 1994, pp. 64–69.

[2] M. M. Michael and M. L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *Proceedings of the ACM Symposium on Principles of Distributed Computing (PoDC)*, May 1996, pp. 267–275.

[3] P. Tsigas and Y. Zhang, "A simple, fast and scalable non-blocking concurrent FIFO queue for shared memory multiprocessor systems," in *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, July 2001, pp. 134–143.

[4] M. Moir, D. Nussbaum, O. Shalev, and N. Shavit, "Using elimination to implement scalable and lock-free fifo queues," in *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, July 2005, pp. 253–262.

[5] M. Hoffman, O. Shalev, and N. Shavit, "The baskets queue," in *Proceedings of the International Conference on Principle of Distributed Systems (OPODIS)*, December 2007, pp. 401–414.

[6] A. Gidenstam, H. Sundell, and P. Tsigas, "Cache-aware lock-free queues for multiple producers/consumers and weak memory consistency," in *Proceedings of the International Conference on Principle of Distributed Systems (OPODIS)*, vol. 6490, December 2010, pp. 302–317.

[7] J. Dongarra and P. Beckman, "The international exascale software roadmap," *International Journal of High Performance Computing Applications (IJHPCA)*, vol. 25, no. 1, pp. 3–60, 2011.

[8] N. Hunt, P. Sandhu, and L. Ceze, "Characterizing the performance and energy efficiency of lock-free data structures," in *Workshop on Interaction between Compilers and Computer Architectures (INTERACT)*, February 2011, pp. 63–70.

[9] A. Gautham, K. Korgaonkar, P. SLPSK, S. Balachandran, and K. Veezhinathan, "The implications of shared data synchronization techniques on multi-core energy efficiency," in *Workshop on Power-Aware Computing and Systems*, October 2012.

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

[11] 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.

[12] C. Isci and M. Martonosi, "Runtime power monitoring in high-end processors: Methodology and empirical data," in *International Symposium on Microarchitecture (MICRO)*, December 2003, pp. 93–104.

[13] V. Tiwari, S. Malik, and A. Wolfe, "Power analysis of embedded software: a first step towards software power minimization," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, November 1994, pp. 384–390.

[14] R. Ge and K. W. Cameron, "Power-aware speedup," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, March 2007, pp. 1–10.

[15] J. Choi, M. Dukhan, X. Liu, and R. Vuduc, "Algorithmic time, energy, and power on candidate HPC compute building blocks," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, Phoenix, AZ, USA, May 2014.

[16] H. Sundell and P. Tsigas, "NOBLE: Non-blocking programming support via lock-free shared abstract data types," *SIGARCH Computer Architecture News*, vol. 36, no. 5, 2008.

[17] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le, "Rapl: Memory power estimation and capping," in *Proceedings of International Symposium on Low Power Electronics and Design (ISLPED)*, August 2010, pp. 189–194.

[18] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, "A portable programming interface for performance evaluation on modern processors," *International Journal of High Performance Computing Applications (IJHPCA)*, vol. 14, no. 3, pp. 189–204, August 2000.

[19] V. M. Weaver, M. Johnson, K. Kasichayanula, J. Ralph, P. Luszczek, D. Terpstra, and S. Moore, "Measuring energy and power with papi," in *Proceedings of International Conference on Parallel Processing Workshops (ICPPW)*, September 2012, pp. 262–268.

[20] A. Atalar, A. Gidenstam, P. Renaud-Goud, and P. Tsigas, "Modeling energy consumption of lock-free queue implementations," Chalmers University of Technology, Tech. Rep. 2014:15, October 2014. [Online]. Available: http://graal.ens-lyon.fr/%7Eprenaud/queues/

[21] H. Sundell, A. Gidenstam, M. Papatriantafilou, and P. Tsigas, "A lock-free algorithm for concurrent bags," in *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, June 2011.

[22] B. B. Mandelbrot, "Fractal aspects of the iteration of $z \to \lambda z(1-z)$ for complex λ and z," *Annals of the New York Academy of Sciences*, vol. 357, pp. 249–259, 1980.

[23] G. Paoloni, "How to benchmark code execution times on Intel® ia-32 and ia-64 instruction set architectures," Intel, Tech. Rep. 324264-001, September 2010.