

# Modeling Energy Consumption of Lock-Free Queue Implementations

Aras Atalar<sup>†</sup>, Anders Gidenstam<sup>†‡</sup>, Paul Renaud-Goud<sup>†</sup> and Philippas Tsigas<sup>†</sup>

<sup>†</sup>Chalmers University of Technology, 412 58 Göteborg; Email: name.surname@chalmers.se

<sup>‡</sup>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 enqueueers' 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.<sup>1</sup>

## Index Terms

lock-free; analysis; modeling; energy; power; throughput; queue; concurrent data structures

<sup>1</sup>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](http://www.excess-project.eu)).  
Technical report - Department of Computer Science and Engineering, Chalmers University of Technology and Gteborg University (2014:15)

# Modeling Energy Consumption of Lock-Free Queue Implementations

## CONTENTS

<b>I</b>	<b>Introduction</b>	3
<b>II</b>	<b>Related work</b>	4
<b>III</b>	<b>Low-level Power Models for CPU-based Platforms</b>	5
III-A	General Power Model . . . . .	5
III-B	Power Components Derivation . . . . .	6
III-C	Dynamic CPU Power . . . . .	6
III-D	Dynamic Memory and Uncore Power . . . . .	8
III-E	Summary of Micro-Benchmarking for Power Modeling on CPU Platform . . . . .	12
<b>IV</b>	<b>Framework</b>	13
IV-A	Synthetic Benchmark . . . . .	13
IV-A1	Skeleton . . . . .	13
IV-A2	Queue Implementations . . . . .	13
IV-B	Notations and Setting . . . . .	14
<b>V</b>	<b>Throughput Estimation</b>	14
V-A	Throughput Decomposition Principles . . . . .	14
V-B	Basic Throughputs . . . . .	15
V-B1	Low Intra-Contention . . . . .	15
V-B2	High Intra-Contention . . . . .	16
V-B3	Frontier . . . . .	16
V-C	Combining Basic Throughputs . . . . .	16
V-D	Instantiating the Throughput Model . . . . .	18
V-D1	Low Intra-Contention . . . . .	19
V-D2	High Intra-Contention . . . . .	19
V-E	Results . . . . .	20
<b>VI</b>	<b>Power Estimation</b>	20
VI-A	CPU Power . . . . .	21
VI-B	Memory and Uncore Power . . . . .	22
VI-C	Instantiating the Power Model . . . . .	25
VI-D	Results . . . . .	25
<b>VII</b>	<b>Energy per Operation Estimation</b>	28
<b>VIII</b>	<b>Description of the Implementations</b>	28
VIII-1	NOBLE [1], [2] . . . . .	28
VIII-2	Tsigas-Zhang [3] . . . . .	30
VIII-3	Valois [4] . . . . .	30
VIII-4	Michael-Scott [5] . . . . .	30
VIII-5	Moir- <i>et al.</i> [6] . . . . .	30

VIII-6	Hoffman-Shalev-Shavit [7] . . . . .	31
VIII-7	Gidenstam-Sundell-Tsigas [8] . . . . .	31
<b>IX</b>	<b>Towards Realistic Applications: Mandelbrot Set Computation</b>	<b>31</b>
IX-A	Description of Mandelbrot Set Application . . . . .	31
IX-B	Mandelbrot Prediction . . . . .	32
<b>X</b>	<b>Conclusion</b>	<b>37</b>
	<b>References</b>	<b>37</b>

## 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.* [3]–[8] 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 [9], 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: enqueueers and dequeuers modify only their respective ends of the queue, and compete mostly with 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 dequeuers  $\mathcal{T}_d$ , even those returning `NULL`, and throughput of enqueueers  $\mathcal{T}_e$ . 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 enqueueers.

Despite this decomposition, enqueueers' 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 IV introduces our modeling framework for lock-free concurrent queues. Section V describes how the throughput of lock-free concurrent queues is modeled, while Section VI describes how the power dissipation is modeled. In Section IX 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 X concludes this paper.

## II. RELATED WORK

Hunt *et al.* [10] 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.* [11] 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.* [12] estimated CPU and memory power. Wang *et al.* [13] provided a two level power model for multiprocessors, which uses frequency and IPC (Instructions Per Cycle) as the only PMC event. Isci *et al.* [14] 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.* [15] 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 [16] 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.* [17] 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. LOW-LEVEL POWER MODELS FOR CPU-BASED PLATFORMS

#### A. General Power Model

The power model presented in Equation 1 decomposes the total power into static, socket activation and dynamic components. In this equation,  $f$  is the clock frequency,  $soc$  the number of activated sockets on the chip,  $op$  is the considered operation and  $thr$  is the number of active cores; the active power is proportional to the number of active sockets, while the dynamic power is proportional to the number of active cores.

For modeling power consumption of data structures, we need to estimate the dynamic component which depends on the frequency, number of active cores, locality and amount of memory requests together with the instruction type. The  $loc$  parameter represents the locality of operands for instructions that can transfer data between memory and registers, such a move from L1, L2, last level cache, main memory or remote memory.

$$\begin{cases} P(f, op, soc, loc, thr) = P_{stat} + P_{active}(f, soc) + P^{(dyn,())} f, op, loc, thr \\ P_{active}(f, soc) = soc \times P_{active}(f) \end{cases} \quad (1)$$

	Static	Active	Dynamic
CPU	$P^{(stat,C)}$	$P^{(active,C)}$	$P^{(dyn,C)}$
Memory	$P^{(stat,M)}$	$P^{(active,M)}$	$P^{(dyn,M)}$
Uncore	$P^{(stat,U)}$	$P^{(active,U)}$	$P^{(dyn,U)}$

TABLE I: Power views

To create another perspective, we decompose the power into two orthogonal bases, each base having three dimensions. On the one hand, we define the model basis by separating the power into static, active and dynamic power, such that the total power is computed by:

$$P = P_{stat} + P_{active} + P_{dyn}.$$

On the other hand, the measurement basis corresponds to the components that actually dissipates the power, *i.e.* CPU, memory and uncore. The power dissipation measurement is done through Intel's RAPL energy counters [18] read via the PAPI library [19], [20]. These counters reflect this discrimination by outputting the power consumption along three dimensions:

- power consumed by CPU, which includes the consumption of the computational cores, and the consumption of the first two level of caches;
- power consumed by the main memory;
- remaining power, called “uncore”, which includes the ring interconnect, shared cache, integrated memory controller, home agent, power control unit, integrated I/O module, config agent, caching agent and Intel QPI link interface.

Also, total power is obtained by the sum:

$$P = P^{(C)} + P^{(M)} + P^{(U)}.$$

This latter additional orthogonal dimension will provide a better perspective for modeling power consumption of data structures, especially for the dynamic component. Table I sums up both dimensions.

In this section, we study each dimension, in each base, so that we are able to express the power dissipation from any perspective:

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

### B. Power Components Derivation

By definition, only the dynamic component of power is dependent on the type of instruction or more generally the executing program. In order to obtain dynamic component  $P^{(dyn,X)}$ , we first have to determine static  $P^{(stat,X)}$  and socket activation  $P^{(active,X)}$  costs.

We examine a large variety of instructions with respect to their power and energy consumption. We have observed a linear relation between the number of threads and power for instructions that do not lead to data transfer between the memory hierarchy and registers. For instance, addition operates on two registers and do not lead to data transfer. Data transfer is done via move instructions before or after addition instruction if required. So, the locality parameter  $loc$  is only valid for instructions that is dependent on the locality of data, like variants of the move instruction. These operations are also prone to variability due to cache and memory states which can also change with the interaction between threads. Briefly,  $P^{(dyn,M)}$  and  $P^{(dyn,U)}$  is ruled by the density of instructions that lead to data transfer in the memory hierarchy. Also, the  $loc$  parameter is not only meaningful for  $P^{(dyn,M)}$ ,  $P^{(dyn,U)}$  but also for  $P^{(dyn,C)}$  as it could lead to stall cycles and decrease instructions per cycle which has a considerable influence on  $P^{(dyn,C)}$ .

For derivation of  $P^{(stat,X)}$  and  $P^{(active,X)}$ , we just use the instructions that operate on the registers because the  $P^{(active,M)}$  and  $P^{(active,U)}$  parts can be neglected for these instructions. We refer to these instructions as  $op_{reg}$  and utilize them to obtain static and socket activation costs for each component (CPU, memory, uncore) of the orthogonal decomposition. A bunch of instructions belonging to  $op_{reg}$  is executed repeatedly for some time interval with varying number of threads for each frequency. We formulate the derivation process as, for all  $X \in \{C, M, U\}$ :

$$\begin{aligned}
 P^{(dyn,)}(f, op, loc, thr) &= P^{(dyn,M,U)}(f, op, loc, thr) + P^{(dyn,C)}(f, op, thr, loc) \\
 P^{(dyn,M,U)}(f, op_{reg}, loc, thr) &= 0 \\
 P^{(dyn,C)}(f, op_{reg}, thr) &= thr \times P^{(dyn,C)}(f, op_{reg}) \\
 P^{(dyn,X)}(f, op_{reg}) &= \frac{1}{2} \left( P^{(X)}(f, op_{reg}, soc, loc, 16) - P^{(X)}(f, op_{reg}, soc, loc, 14) \right) \\
 P^{(active,X)}(f) &= P^{(X)}(f, op_{reg}, 2, loc, 10) - P^{(X)}(f, op_{reg}, 1, loc, 8) - P^{(dyn,X)}(f, op_{reg}) \times 2 \\
 P^{(stat,X)}() &= P^{(X)}(f, op_{reg}, soc, loc, thr) - soc \times P^{(active,X)}(f) - thr \times P^{(dyn,X)}(f, op_{reg})
 \end{aligned}$$

Using above equations, we verified that  $P^{(stat,X \in \{C,M,U\})}$  is approximately constant according to instruction type, pinning, number of threads and frequency thus we take the mean of the values of  $P^{(stat,X)}$  over the whole space to find  $P^{(stat,X)}$ . We apply the same approach to find  $P^{(active,X \in \{C,M,U\})}$  which only depends on frequency, and not on the operation. Having obtained  $P^{(stat,X)}$  and  $P^{(active,X)}$ , we extract  $P^{(dyn,X \in \{C,M,U\})}$  for “all” types of instructions, thread, pinning and frequency setting, by removing the static and active part from the total power.

### C. Dynamic CPU Power

Having determined and excluded static and socket components, we obtain the dynamic power component for each instruction, thread count, pinning and frequency setting. Among a large variety of instructions that are surveyed, we pick a small set of instructions that can be representative for data structure implementations, namely *Compare-and-Swap*, pause, floating point division, addition together with vector addition. *Compare-and-Swap* can be representative for the retry loops and pauses/additions can be used to represent the parallel work which determines the contention on the data structures. The decomposition of dynamic power in terms of CPU, memory and uncore components for these instructions are illustrated in Figures 1, 2 and 3.

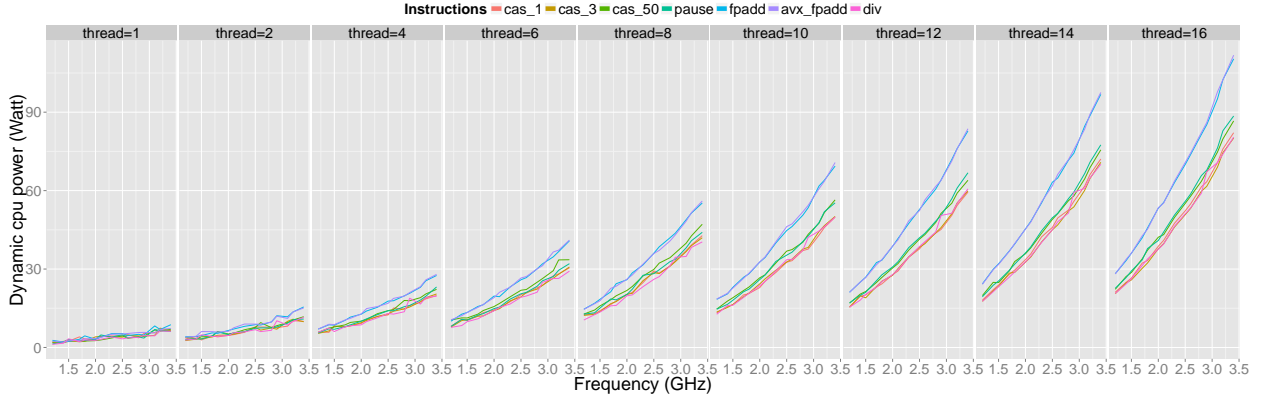


Fig. 1: Dynamic CPU power for micro-benchmarks

Based on the observation that  $P^{(dyn,C)}$  shows almost linear behavior with respect to number of threads, we model the convex  $P^{(dyn,C)}$  as:

$$P^{(dyn,C)}(f, op) = (A \times f^\alpha + B)$$

Each instruction might provide different power behavior as illustrated in Figure 1, therefore we find  $A$ ,  $B$ ,  $\alpha$  for each instruction separately.  $B$  could be different for each instruction because of the activation of different functional units, this is also why we included this constant in  $P^{(dyn,C)}$ .

	$A$	$\alpha$	$B$
cas	0.001392	1.6415	0.0510
fpdiv	0.001038	1.7226	0.0585
add	0.001004	1.8148	0.0912
avx-add	0.001130	1.7828	0.0894
pause	0.000854	1.7920	0.0736

TABLE II: Instruction power coefficients

To obtain  $A$ ,  $B$  and  $\alpha$ , we proceed in the following way. We are given an operation  $op$ , and we consider the executions of this operation with 16 threads on 2 sockets. Let  $v^{(freq)}$  be the vector of frequencies where we want to estimate the dynamic power (we dispose  $F$  different frequencies, expressed in  $10^{-1}$  GHz, such that  $v_1^{(freq)} = 12$  and  $v_F^{(freq)} = 34$ ). We note  $v^{(meas)}$  the vector of dynamic powers that have been computed from the measurements through the process described above, and  $v^{(est)}(A, B, \alpha)$  the vector of estimated dynamic powers. More especially, for all  $i \in \{1, \dots, F\}$ :

$$v_i^{(meas)} = P^{(dyn,C)}(v_i^{(freq)}, op)$$

$$v_i^{(est)}(A, B, \alpha) = \left( A \times \left( v_i^{(freq)} \right)^\alpha + B \right)$$

The Euclidean norm of a vector  $v$  is denoted  $\|v\|$ .

We solve the following minimization problem, with the help of the Matlab “fminsearch” function:

$$\min_{A, B, \alpha} \left\| v^{(meas)} - v^{(est)}(A, B, \alpha) \right\|$$

Table II provides the values for power constants and exponent for selected instructions.



#### D. Dynamic Memory and Uncore Power

In the micro-benchmarks, we observe that many instructions do not lead to an increase in dynamic memory and uncore power because of the locality of operands. On the other hand, *Compare-and-Swap* micro-benchmark, in which threads execute *Compare-and-Swap* on the same cache line repeatedly, lead to an increase in memory and uncore power only when the threads are pinned to different sockets. The resulting *ping-pong* of the updated cache line between sockets is responsible for this effect. We observe this empirically but we do not know the exact reason why this fact leads to memory consumption. In our platform, 8 physical cores share a L3 cache in each socket which are connected via QPI links. The cache line is expected to reside in the L3 cache of the remote socket even if it is not in the local one so can be delivered without a memory access, as in intra-socket case. We do not have precise information about the cache coherence protocol but we can make use of the study [21] to make some predictions about relevant protocols. The remote transfer between L3 caches takes place with the assistance of Home Agents and QPI link. Once request is delivered to remote Home Agent by local one over QPI, remote agent initiates concurrent requests to memory and local cache domain. This might lead to memory consumption even the data is supplied from the cache domain. As a second possibility, remote access triggers a write-back to memory, which might again lead to memory consumption. In brief, it can be observed that the rate of the inter-socket cache line transfers influence the memory consumption.

*Compare-and-Swap* micro-benchmarks are indeed prone to unfairness among threads. When *Compare-and-Swap* is executed repeatedly by all threads on the same cache line without any work in between *Compare-and-Swap* attempts, the thread which gets the ownership of cache line succeeds repeatedly while others starve. This unfairness decreases the transfer rate of the cache line and grows when inter-socket communication is in the play. In addition, the inter-socket communication is limited with a single cache line as accesses are serialized due to atomicity requirement. Due to this, we introduce 3 different *Compare-and-Swap* micro-benchmarks looping on 1, 3 and 50 shared variables that are aligned to different cache lines. By doing so, we aim at increasing the traffic between cores and sockets. Figures 2 and 3 provide the dynamic memory and uncore power dissipations. It can be observed frequency play a role in any case and also number of threads, if there are multiple cache line, as they increase the transfer rate. As a remark for the provided figures, threads are pinned using a dense mapping strategy that leads to inter-socket communication only after 8 threads.

When threads are pinned to same socket, intra-socket communication between them takes place via the ring interconnect without introducing a memory access. Thus, absence of increase in memory power is reasonable. However, one might expect an increase in uncore power for these cases because RAPL uncore power includes L3 and ring interconnect power. We do observe a very slight increase in uncore power

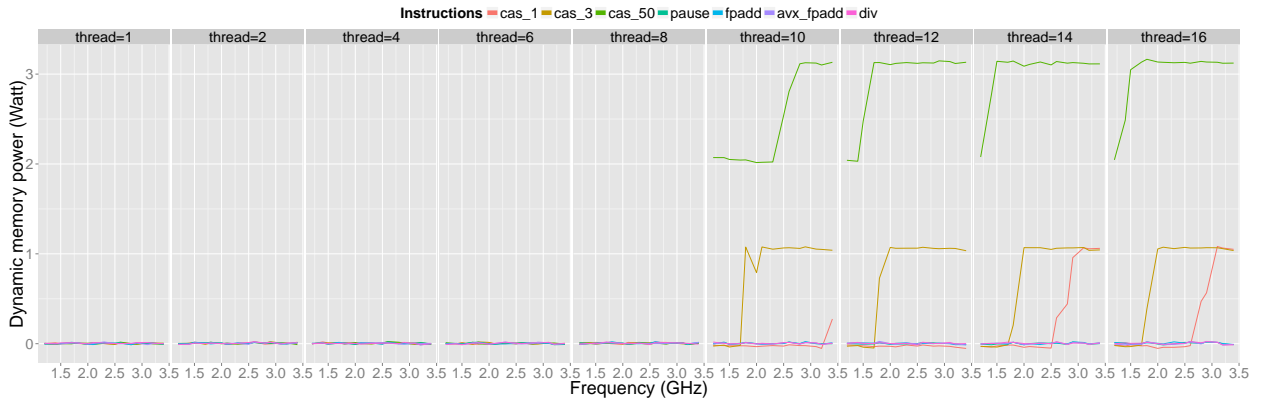


Fig. 2: Dynamic memory power for micro-benchmarks

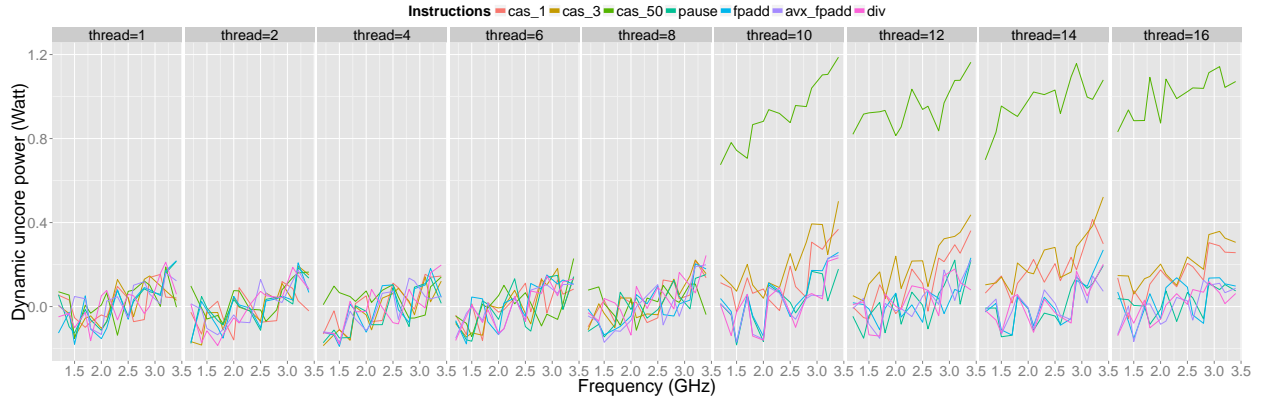


Fig. 3: Dynamic uncore power for micro-benchmarks

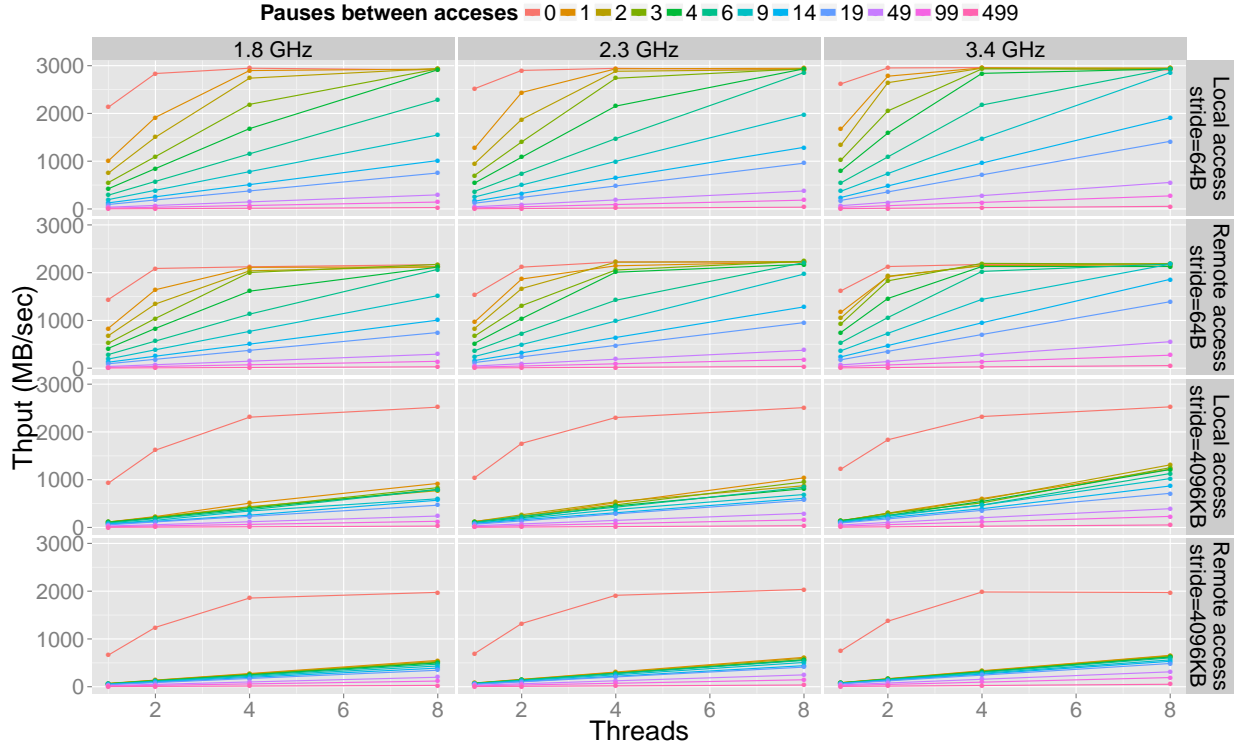


Fig. 4: Throughput for array traversal benchmark

with frequency, probably because it is in the same frequency domain, but not for number of threads. A noticeable increase of uncore can be observed when threads are pinned to different sockets, due to remote accesses which uses important uncore components such as the QPI link interface and Home Agent.

Briefly, dynamic memory and uncore power are seemed to follow the same trends as going off-chip requires usage of major uncore components and also increases the memory power. Thus, the memory and uncore power is ruled by the amount of local/remote memory accesses and inter-socket *ping-pong* of cache lines per unit of time. Frequency has a direct impact on this rate but number of threads is

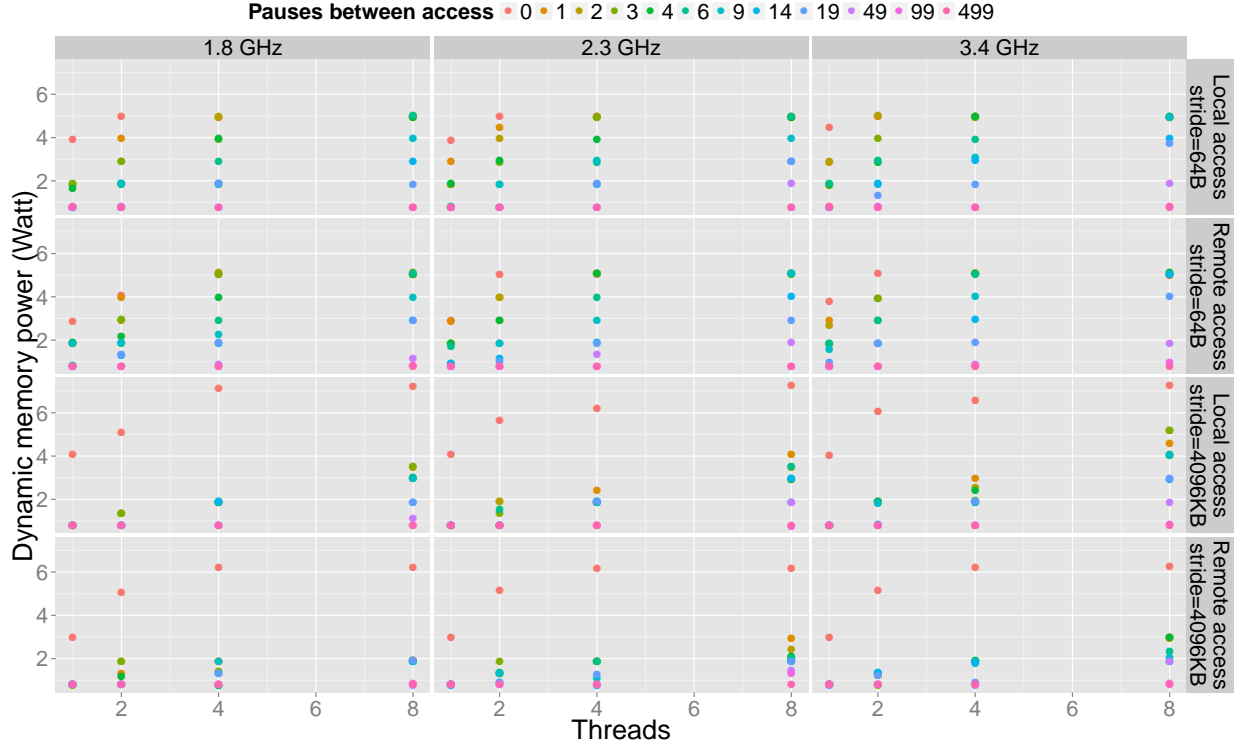


Fig. 5: Step-like power for array traversal benchmark

dependent on the multitude of cache lines in order to increase the rate of accesses.

An interesting observation regarding memory power is that it shows a step function behavior. We think that this is because of the RAPL power capping algorithm which determines a power budget based on memory bandwidth, as presented in the work of David *et al.* [22]. The RAPL algorithm specifies a power cap for a time window depending on the memory bandwidth requirements of previous time intervals and sets the memory in a power state that is expected to maximize energy efficiency. Based on the amount of memory accesses, it jumps between states finding a trade-off between bandwidth and power. The finite number of states leads to the step-like power curves in Figure 2.

To justify these observations, we use a benchmark which stresses the main memory. We allocate a huge contiguous array and align each element of the array to a separate cache line. In addition, we force the array to be allocated in the memory module residing in the first socket. Thus, we can regulate remote and local memory accesses by pinning strategies. We pin all threads either to first or second socket to separate cores, in this case the maximum number of threads is 8 as we disabled the logical cores. We change the number of threads, frequency and interleave varying amount of pause operations between array accesses to change the bandwidth requirements of the benchmark. Also, threads access independent portions of the array with a stride. The hardware prefetcher increases the performance remarkably when adjacent cache lines are accessed while traversing the array and a stride of page size can be used to disable the hardware prefetcher. We run the same experiment both with a stride of 64 Bytes, which is the size of a cache line, and 4096 kBytes which is the page size, to reveal effect of prefetching. As provided in Figure 4, the system reaches its maximum bandwidth more rapidly when the prefetcher is activated and attains better bandwidth. Moreover, the bandwidth difference between completely remote and local accesses is noticeable. Another point is that frequency does not influence the maximum achievable bandwidth.

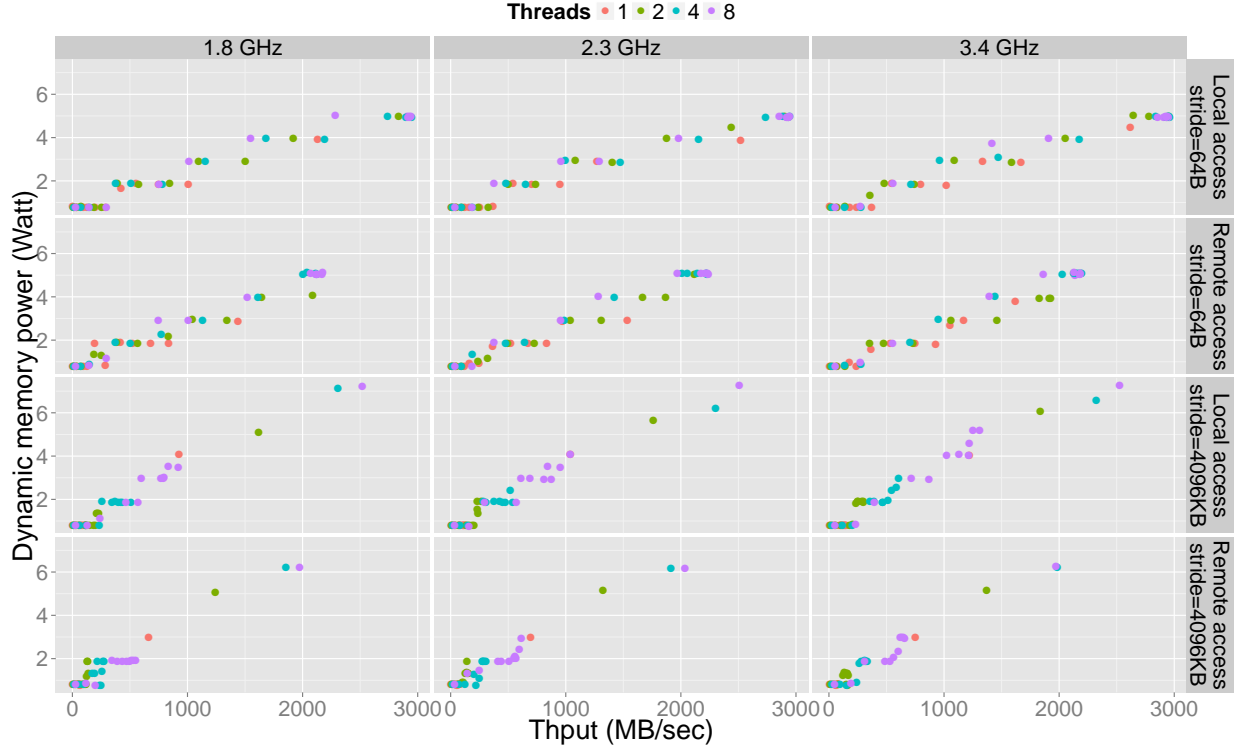


Fig. 6: Memory power for array traversal benchmark

This fact means that there is opportunity for energy savings with DVFS for memory-bounded regions of applications. We also illustrate the step-like power behavior, due to the RAPL algorithm, with this benchmark in Figure 5.

In Figure 6, dynamic memory power consumption is plotted according to the number of bytes accesses per second, in other words rate of accesses. From the analysis of the results, it can be deduced that the memory power is strongly correlated with the number of bytes accessed per second. There is no clear impact of the number of threads and frequency to the memory power except their indirect effect on bandwidth. In contrast, access stride has a direct, though limited, impact on the memory power together with its indirect impact as it increases the bandwidth. By accessing data with a stride of a cache line, we possibly make use of the open page mode of DRAM which could be influential in terms of energy efficiency due to avoidance of bit-line precharge and row access cost. But, we still observe a linear relation between throughput and memory power for both strides. In addition, remote or local accesses do not provide a noticeable difference for memory power in case the number of bytes accesses per second is equal.

On the other hand, it is observed that uncore power depends on the frequency, partially because it operates in the same frequency domain. Furthermore, the cost of accessing a byte remotely is larger than locally because it requires QPI link interface which adds an additional cost compared to local memory accesses as shown in Figure 7. All these observations regarding memory and uncore power will shed light to the analysis and modeling of data structures. One major source of differences in power consumption between different implementations is the memory and uncore consumption, which is related to locality and bandwidth requirements of the implementations.

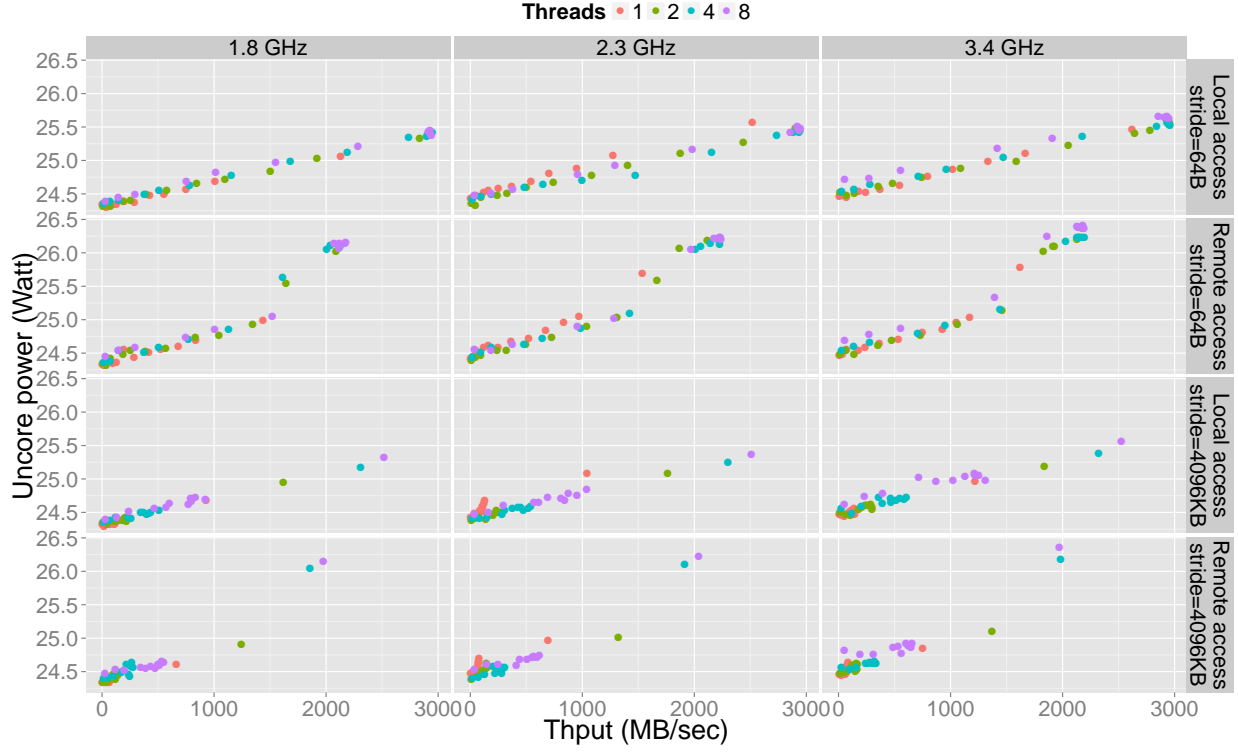


Fig. 7: Uncore power for array traversal benchmark

### E. Summary of Micro-Benchmarking for Power Modeling on CPU Platform

Figure 8 recalls the main achievements of the micro-benchmark study on CPU, where  $d$  is the amount of memory accessed per unit of time in the main memory or through QPI link.

	Static	Active	Dynamic
CPU	$P^{(stat,C)}(f, op, soc, loc, n)$	$P^{(active,C)}(f, op, soc, loc, n)$	$P^{(dyn,C)}(f, op, soc, loc, n)$
Memory	$P^{(stat,M)}(f, op, soc, loc, n)$	$P^{(active,M)}(f, op, soc, loc, n)$	$P^{(dyn,M)}(f, op, soc, loc, n)$
Uncore	$P^{(stat,U)}(f, op, soc, loc, n)$	$P^{(active,U)}(f, op, soc, loc, n)$	$P^{(dyn,U)}(f, op, soc, loc, n)$

Dependency  
removal

	Static	Active	Dynamic
CPU	$P^{(stat,C)}()$	$soc \times P^{(active,C)}(f)$	$n \times (A(op) \times f^{\alpha(op)} + B(op))$
Memory	$P^{(stat,M)}()$		$d \times P^{(dyn,M)}(op, loc)$
Uncore	$P^{(stat,U)}()$		$d \times P^{(dyn,U)}(op, loc)$

Fig. 8: Dependency shrinking

<pre> <b>while</b> ! done <b>do</b>     <b>el</b> <math>\leftarrow</math> Parallel_Work(<math>pw_e</math>);     Enqueue(<b>el</b>); <b>end</b> </pre> <p style="text-align: center;"><b>Procedure Enqueuer</b></p>	<pre> <b>while</b> ! done <b>do</b>     <b>el</b> <math>\leftarrow</math> Dequeue();     Parallel_Work(<math>pw_d</math>); <b>end</b> </pre> <p style="text-align: center;"><b>Procedure Dequeuer</b></p>
--	---

Fig. 9: Thread procedures

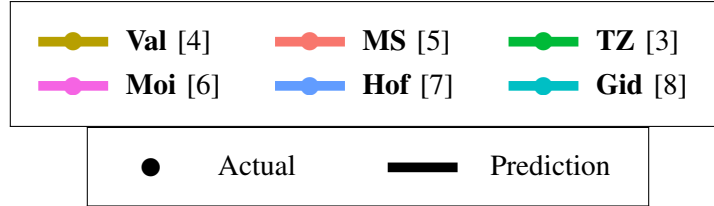


Fig. 10: Key legend of the graphs

#### IV. FRAMEWORK

##### A. Synthetic Benchmark

1) *Skeleton*: We run the synthetic benchmark composed of the two functions described in Figure 9, starting with an empty queue. Half of the threads are assigned to be enqueueers while the remaining ones are dequeuers. We disable logical 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 enqueueer/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 enqueueers 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 *pauses* (which corresponds to 1000 cycles) that compose the parallel work in the enqueueer (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 [2]. These queue algorithms are described in some detail in Section VIII. The legend depicted in Figure 10 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.

When we speak about implementations of the queues, we actually refer to the different implementations of enqueueing and dequeuing operations, along with their memory management schemes.

## B. 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 enqueue (resp. dequeue), as the number of bunches of 90 *pauses*. 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  $t(PS_o)$  (resp.  $t(RL_o)$  and  $t(SL_o)$ ).

In the same way, for  $o \in \{d, e\}$ , we denote by  $P_o^{(X)}$  (resp.  $P_{o,PS}^{(X)}$  and  $P_{o,RL}^{(X)}$ ) the dynamic 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 V and VI, in order to keep expressions as simple as possible, we define one unit of time as  $\lambda$  sec, where  $\lambda$  is the execution time of  $90 \times f$  *pauses* (as the *pause* instructions are perfectly scalable with clock frequency,  $\lambda$  is constant). Throughput is expressed in number of operations per unit of time, *i.e.* per  $\lambda$  secs. Finally, we derive the power in Watts.

All experiments and their underlying predictions are done on the following platform:

- CPU: Intel® Xeon® CPU E5-2687W v2
  - 2 sockets, 8 cores per socket
  - Maximum frequency: 3.4GHz, Minimum frequency: 1.2GHz, frequency speedstep by DVFS (Dynamic Voltage and Frequency Scaling): 0.1 – 0.2 GHz., Turbo mode: 4.0 GHz.
  - Hyperthreading (disabled)
  - L3 cache: 25 MB, internal write-back unified, L2 cache: 256 kB, internal write-back unified. L1 cache (data): 32 kB internal write-back
- DRAM: 16 GB in four 4 GB DDR3 REG ECC PC3-12800 modules run at 1600 MTransfers/sec. Each socket has 4 DDR3 channels, each supporting 2 modules. In this case 1 channel per socket is used.
- Motherboard: Intel Workstation W2600CR, BIOS version: 2.000.1201 08/22/2013
- Hard drive: Seagate ST10000DM003-9YN162 1 TB SATA.

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  $n \in \{1, \dots, 8\}$ .

## V. THROUGHPUT ESTIMATION

### A. Throughput Decomposition Principles

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

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

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

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

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

We have seen that the parallel sections of the benchmark are full of *pauses*, thus the time  $t(PS_o)$  spent in a given parallel section is straightforwardly given by  $t(PS_o) = pw_o/f$ . 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, enqueueers and dequeueers try to access the same data, then interference occurs; enqueueers make dequeueers 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-contended, 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 V-B, we describe the basic throughputs, we combine them in Section V-C, then we explain how to instantiate the parameters of the model in Section V-D, and finally exhibit results in Section V-E.

## 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-contended.

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 11, we have a cyclic execution, and the length of the shortest cycle is  $t(PS_o) + t(SL_o^{(b)})$ . Within each cycle, every thread performs exactly one successful operation, thus the throughput is easy to compute:

$$\mathcal{T}_o^{(b)} = \frac{n}{t(PS_o) + t(SL_o^{(b)})} = \frac{nf}{pw_o + cw_o^{(b)}}. \quad (2)$$



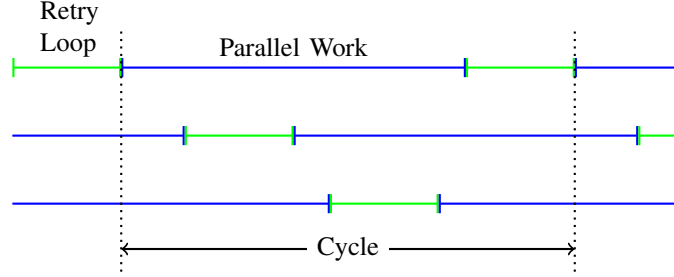


Fig. 11: Cyclic execution under low intra-contention

2) *High Intra-Contention*: As explained in Section V-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  $pw_o$ .

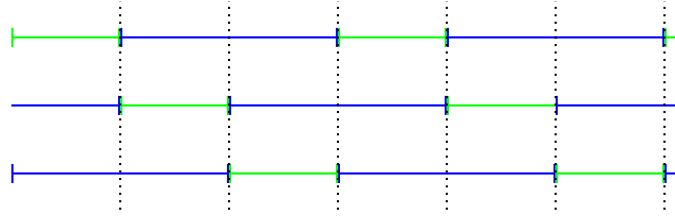


Fig. 12: Intra-contention frontier

3) *Frontier*: We now have to estimate whether the queue is highly intra-contended. We recall that, generally speaking, a long parallel section leads to a low intra-contended queue since threads are most of the time processing some computations and are not trying to access the shared data. Reversely, when the parallel section is short, the ratio of time that threads spend in the retry loop is higher, and gets even higher because of both expansion and retries.

That being said, 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 12 an ideal execution with  $n = 3$  threads and  $t(PS_o) = (n - 1) \times t(SL_o^{(b)})$ . 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 ( $t(PS_o) = (n - 1) \times t(SL_o^{(b)})$ ) 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 enqueueers' 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:

$$\mathcal{T}_d = \frac{\mathcal{T}_e}{\mathcal{T}_d^{(-)}} \times \mathcal{T}_d^{(-)} + \left(1 - \frac{\mathcal{T}_e}{\mathcal{T}_d^{(-)}}\right) \times \mathcal{T}_d^{(+)} \quad (3)$$

Concerning the enqueueers, we use the same assumption on inter-contention as used on intra-contention in Section V-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  $t(RL_d)$ , but we know that in average, to complete a successful operation, a thread needs  $t(PS_d) + t(RL_d)$  units of time, and among this time it will spend  $t(PS_d)$  in the parallel section. Therefore

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

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

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

In the second case, enqueueers 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 enqueueers' 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 enqueueers' throughput, several solutions could be consistent.

*Theorem 1:* Given  $(\mathcal{T}_e^{(+)}, \mathcal{T}_e^{(-)}, \mathcal{T}_d^{(+)}, \mathcal{T}_d^{(-)})$ , 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, this solution is unique and is such that  $\mathcal{T}_e = \mathcal{T}_e^{(+)}$  and  $\mathcal{T}_d = \mathcal{T}_d^{(-)}$ .

*Proof 1:* ( $\Rightarrow$ ) If the queue is growing, then  $\mathcal{T}_e > \mathcal{T}_d$ . Moreover, dequeues never occur on an empty queue, hence  $\mathcal{T}_d = \mathcal{T}_d^{(-)}$ , and there is no inter-contention, thus  $\mathcal{T}_e = \mathcal{T}_e^{(+)}$ .

( $\Leftarrow$ ) Let us assume now that  $\mathcal{T}_e^{(+)} > \mathcal{T}_d^{(-)}$ .  $\mathcal{T}_e = \mathcal{T}_e^{(+)}$  and  $\mathcal{T}_d = \mathcal{T}_d^{(-)}$  is a valid solution, such that the queue is growing, since then  $\mathcal{T}_e > \mathcal{T}_d$ .

By construction,  $\mathcal{T}_e \leq \mathcal{T}_e^{(+)}$ ; if we had another solution such that the queue grows and  $\mathcal{T}_e < \mathcal{T}_e^{(+)}$ , it would mean that enqueuees are inter-contended, which is possible only when the queue is mostly empty. This is absurd, hence the uniqueness.

*Theorem 2:* Given  $(\mathcal{T}_e^{(+)}, \mathcal{T}_e^{(-)}, \mathcal{T}_d^{(+)}, \mathcal{T}_d^{(-)})$ , there exists a solution  $(\mathcal{T}_d, \mathcal{T}_e)$  with a mostly empty queue if and only if

$$\frac{\mathcal{T}_e^{(-)}}{\mathcal{T}_d^{(-)}} \leq 1 - \frac{pw_d}{n \times f} (\mathcal{T}_e^{(+)} - \mathcal{T}_e^{(-)}) \quad (5)$$

In addition, this solution is unique and is given by Equations 4 and 3.

*Proof 2:* ( $\Rightarrow$ ) Let a solution with a mostly empty queue. By construction, the throughputs follow Equations 4 and 3. As  $\mathcal{T}_e$  is an increasing function according to  $\mathcal{T}_d$  (because  $\mathcal{T}_e^{(+)} \geq \mathcal{T}_e^{(-)}$ ), we derive

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

The queue is mostly empty, thus the dequeues of non-NULL items have to be faster than the enqueues, which translates into  $\mathcal{T}_d^{(-)} \geq \mathcal{T}_e$ . The two inequalities combined show the implication.  
 $(\Leftarrow)$  Let us assume now that Inequality 5 is fulfilled. Equation 3 can be rewritten into

$$\mathcal{T}_e = \frac{\mathcal{T}_d - \mathcal{T}_d^{(+)}}{1 - \frac{\mathcal{T}_d^{(+)}}{\mathcal{T}_d^{(-)}}}.$$

Let us consider now  $\mathcal{T}_e'$  and  $\mathcal{T}_e''$  two functions of  $\mathcal{T}_d'$  that fulfill the following system of equations:

$$\begin{cases} \mathcal{T}_e'(\mathcal{T}_d') = \frac{\mathcal{T}_d' - \mathcal{T}_d^{(+)}}{1 - \frac{\mathcal{T}_d^{(+)}}{\mathcal{T}_d^{(-)}}} \\ \mathcal{T}_e''(\mathcal{T}_d') = \frac{\mathcal{T}_d' \times pw_d}{n \times f} \times \mathcal{T}_e^{(+)} + \left(1 - \frac{\mathcal{T}_d' \times pw_d}{n \times f}\right) \times \mathcal{T}_e^{(-)}. \end{cases}$$

We have  $\mathcal{T}_e'(\mathcal{T}_d^{(+)}) = 0$  and  $\mathcal{T}_e'(\mathcal{T}_d^{(-)}) = \mathcal{T}_d^{(-)}$ . According to Inequality 5, we know also that  $\mathcal{T}_e''(\mathcal{T}_d^{(+)}) \leq \mathcal{T}_d^{(+)}$ . In addition,  $\mathcal{T}_e''$  is a linearly increasing function of  $\mathcal{T}_d'$  and  $\mathcal{T}_e'$  a linearly decreasing function of  $\mathcal{T}_d'$ . This shows that there exists a unique  $\mathcal{T}_d$  such that  $\mathcal{T}_e'(\mathcal{T}_d) = \mathcal{T}_e''(\mathcal{T}_d)$ , and if we define  $\mathcal{T}_e$  as  $\mathcal{T}_e = \mathcal{T}_e'(\mathcal{T}_d) = \mathcal{T}_e''(\mathcal{T}_d)$ , the pair  $(\mathcal{T}_d, \mathcal{T}_e)$  is such that

$$\begin{cases} \mathcal{T}_d^{(-)} \leq \mathcal{T}_d \leq \mathcal{T}_d^{(+)} \\ \mathcal{T}_e^{(-)} \leq \mathcal{T}_e \leq \mathcal{T}_e^{(+)} \\ \mathcal{T}_e \leq \mathcal{T}_d \end{cases}.$$

This implies that it is a solution with an empty queue, and we have shown that this solution is unique.

*Corollary 1:* Given  $(\mathcal{T}_e^{(+)}, \mathcal{T}_e^{(-)}, \mathcal{T}_d^{(+)}, \mathcal{T}_d^{(-)})$ , there exists at least one solution  $(\mathcal{T}_d, \mathcal{T}_e)$ .

*Proof 3:* We show that if the inequality of Theorem 1 is not fulfilled, i.e. if  $\mathcal{T}_e^{(+)} \leq \mathcal{T}_d^{(-)}$ , then the inequality of Theorem 2 is true. We have indeed

$$\begin{aligned} \mathcal{T}_d^{(-)} \times \left(1 - \frac{pw_d}{n \times f} (\mathcal{T}_e^{(+)} - \mathcal{T}_e^{(-)})\right) - \mathcal{T}_e^{(-)} &= \mathcal{T}_d^{(-)} \times \left(1 - \frac{pw_d \times \mathcal{T}_e^{(+)}}{n \times f}\right) - \mathcal{T}_e^{(-)} \times \left(1 - \frac{pw_d \times \mathcal{T}_d^{(-)}}{n \times f}\right) \\ &\geq \mathcal{T}_d^{(-)} \times \left(1 - \frac{pw_d \times \mathcal{T}_e^{(+)}}{n \times f}\right) - \mathcal{T}_e^{(+)} \times \left(1 - \frac{pw_d \times \mathcal{T}_d^{(-)}}{n \times f}\right) \\ &\geq \mathcal{T}_d^{(-)} - \mathcal{T}_e^{(+)} \\ \mathcal{T}_d^{(-)} \times \left(1 - \frac{pw_d}{n \times f} (\mathcal{T}_e^{(+)} - \mathcal{T}_e^{(-)})\right) - \mathcal{T}_e^{(-)} &\geq 0, \end{aligned}$$

which proves the Corollary.

One can notice that if  $\mathcal{T}_e^{(+)} > \mathcal{T}_d^{(-)}$  and Inequality 5 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. enqueueers') throughput as the amount of work in the parallel section of the dequeuers is  $pw_d$  and enqueueers' 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 that 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-contended 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 3, we have

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

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

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 3, 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 enqueueers: 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:

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

Finally, dequeuers' and enqueueers' throughput are reconstituted as explained in Section V-C: if Equation 5 is fulfilled, then they are computed through Equations 3 and 4 that can be rewritten as:

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

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

The throughput predictions are animated in Figure 13 for the enqueueers, and in Figure 14 for the dequeuers (the legend is in Figure 10, p13). You need to run animations using a recent version of Adobe Acrobat and in case they do not appear we also provide graphs in Figures 15, 17, 16 and 18 for some  $pw_d$  values. Points are measurements, while lines 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 13. 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 animation in Figure 13 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-contended cases. When  $cw_e$  increases, some critical points shift slightly towards the right as the intra-contention starts with a larger  $pw_e$ . In Figure 14, 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.

## VI. POWER ESTIMATION

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

Fig. 13: Enqueue throughput

#### 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

$$P^{(C)} = n \times P_e^{(C)} + n \times P_d^{(C)}. \quad (6)$$

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:

$$P_o^{(C)} = r_o \times P_{o,RL}^{(C)} + (1 - r_o) \times P_{o,PS}^{(C)}. \quad (7)$$

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

$$r_o = 1 - \frac{\mathcal{T}_o \times pw_o}{n \times f}. \quad (8)$$

Altogether, we obtain the final formula for dynamic CPU power

$$P^{(C)} = n \left( \sum_{o \in \{e,d\}} P_{o,RL}^{(C)} + \frac{\mathcal{T}_o \times pw_o \times (P_{o,PS}^{(C)} - P_{o,RL}^{(C)})}{n \times f} \right) \quad (9)$$




Fig. 14: Dequeue throughput

### B. Memory and Uncore Power

We have noticed 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 *pauses*, 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-contended, 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 enqueueers) 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  $r_o$

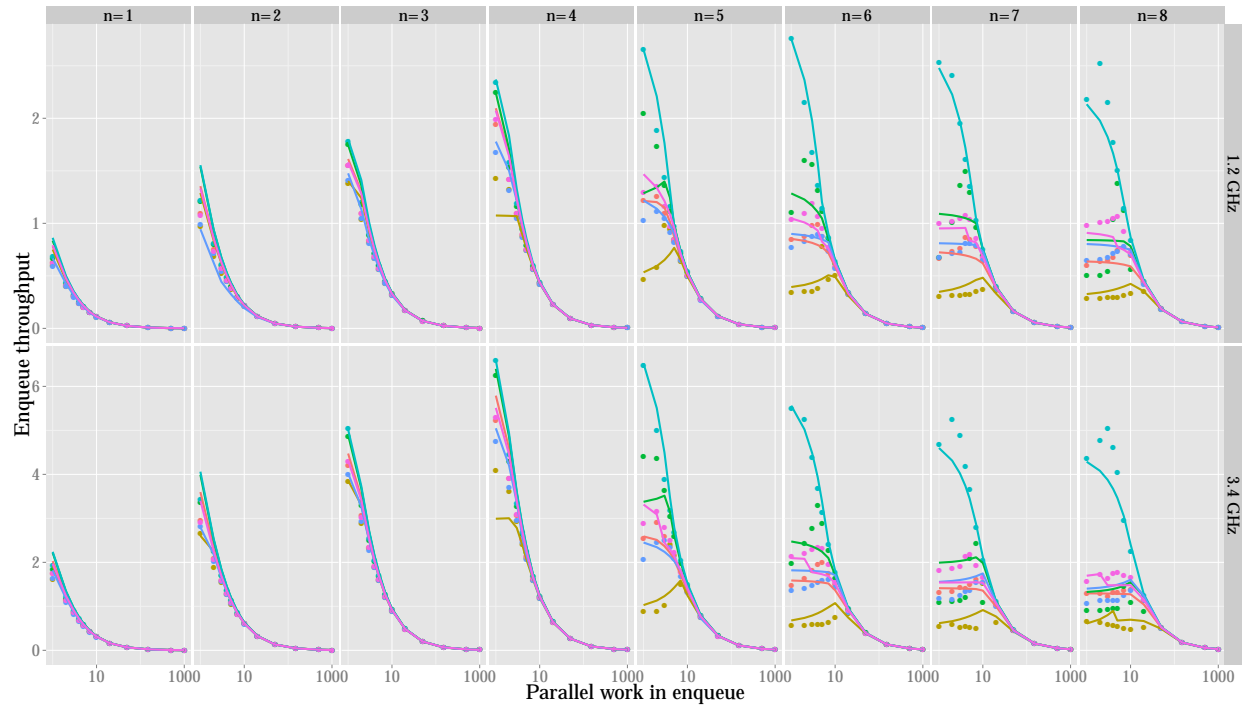


Fig. 15: Enqueue throughput with  $pw_d = 7$

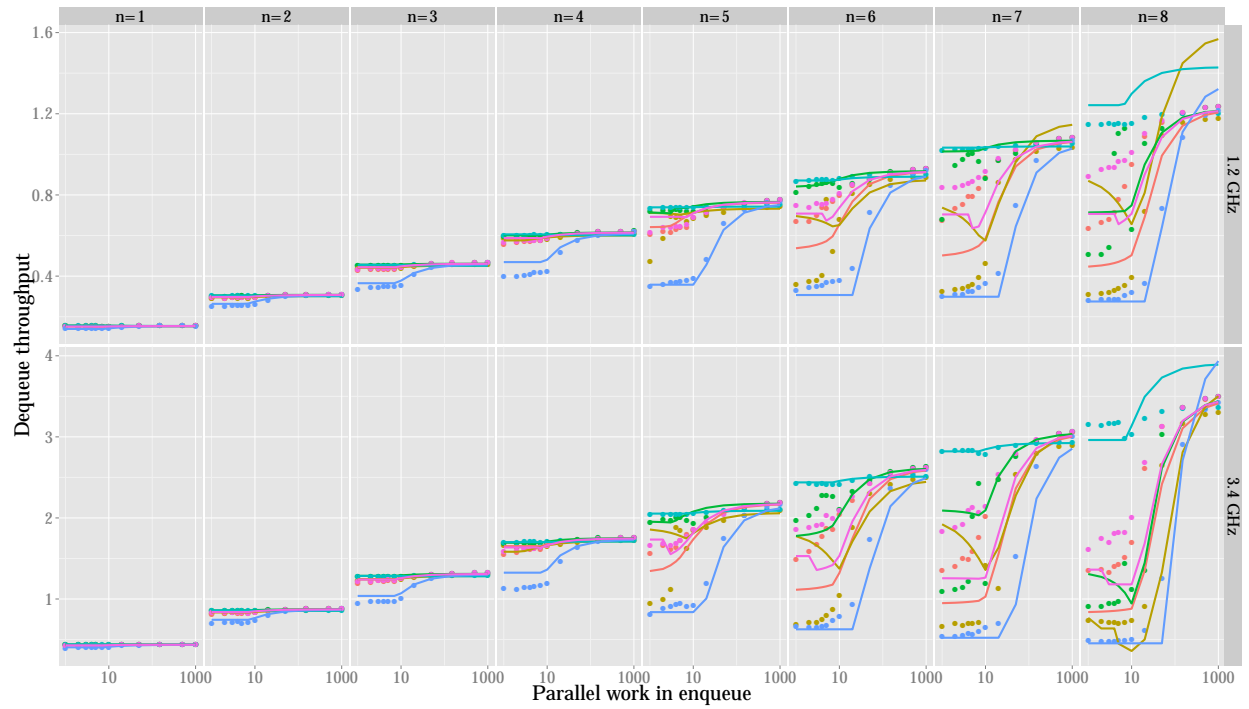


Fig. 16: Dequeue throughput with  $pw_d = 7$



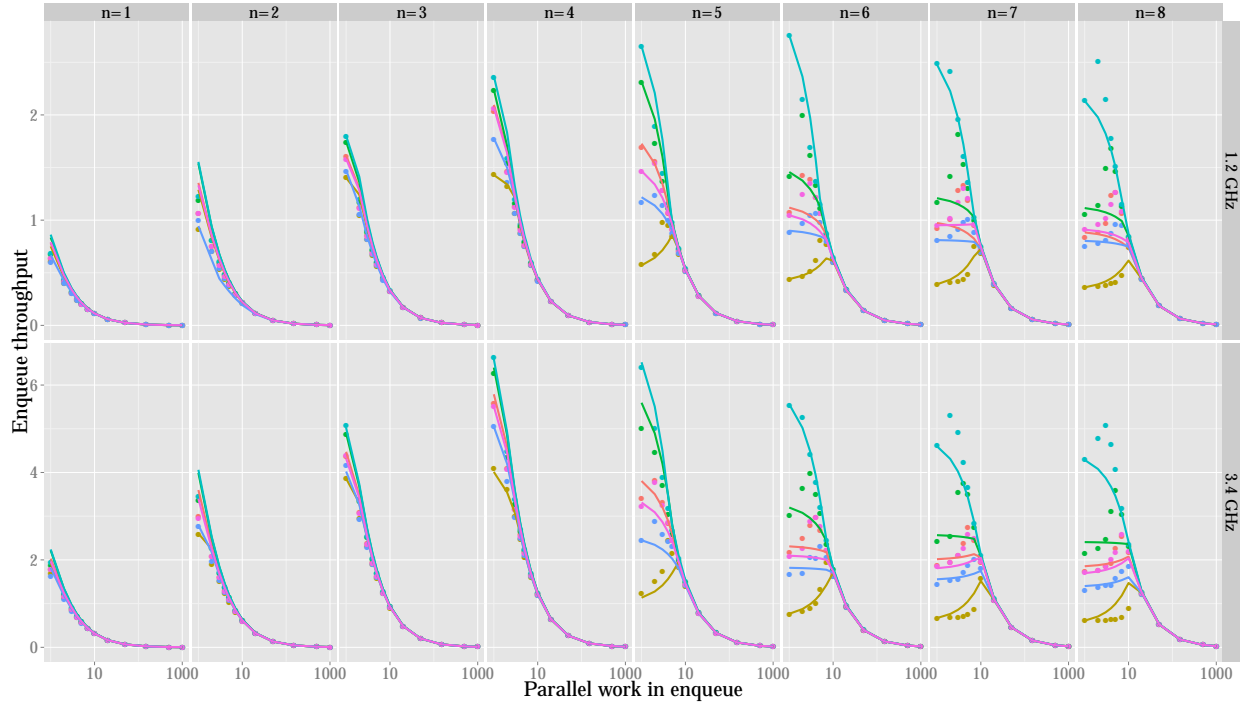


Fig. 17: Enqueue throughput with  $pw_d = 50$

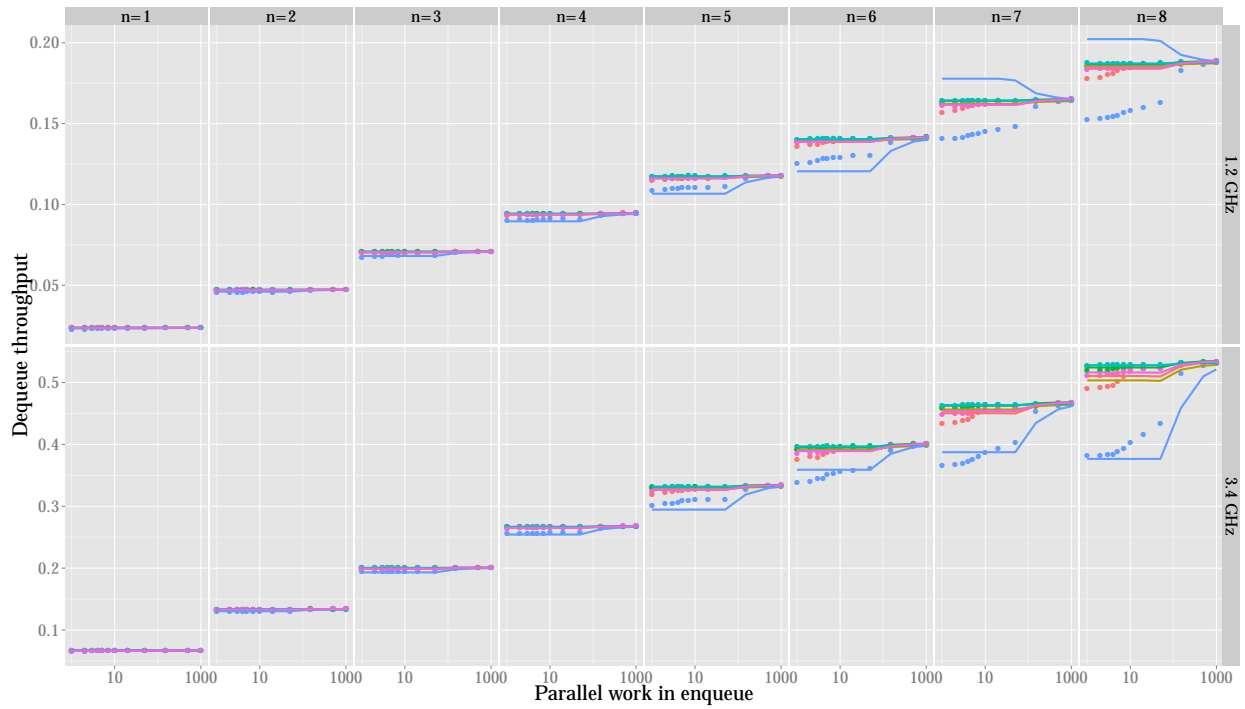


Fig. 18: Dequeue throughput with  $pw_d = 50$

(times the amount of data accessed per unit of time in the retry loop, which is a constant). Hence

$$P^{(M)} = r_e \times \rho_e^{(M)} + r_d \times \rho_d^{(M)}, \quad (10)$$

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 enqueueers 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 10 (resp. Equations 6 and 7), and the estimates of the ratios that are issued from Section V

### D. Results

The prediction and measurements, regarding power, are plotted in Figures 19, 20, 21, 22 and 23, where we observe that the most significant differences lie in the dynamic memory power. The differences in CPU power are almost invisible, since the dynamic power of the parallel sections (composed of *pauses* instructions) is very close to the dynamic power of the retry loops. As in Section III-D, we remark some steps in the measured memory power, but we prefer to keep a continuous estimate.

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 19 (legend is in Figure 10). Overall, the prediction reacts correctly to the variations of parallel section sizes, and some specifics of the algorithms are caught,

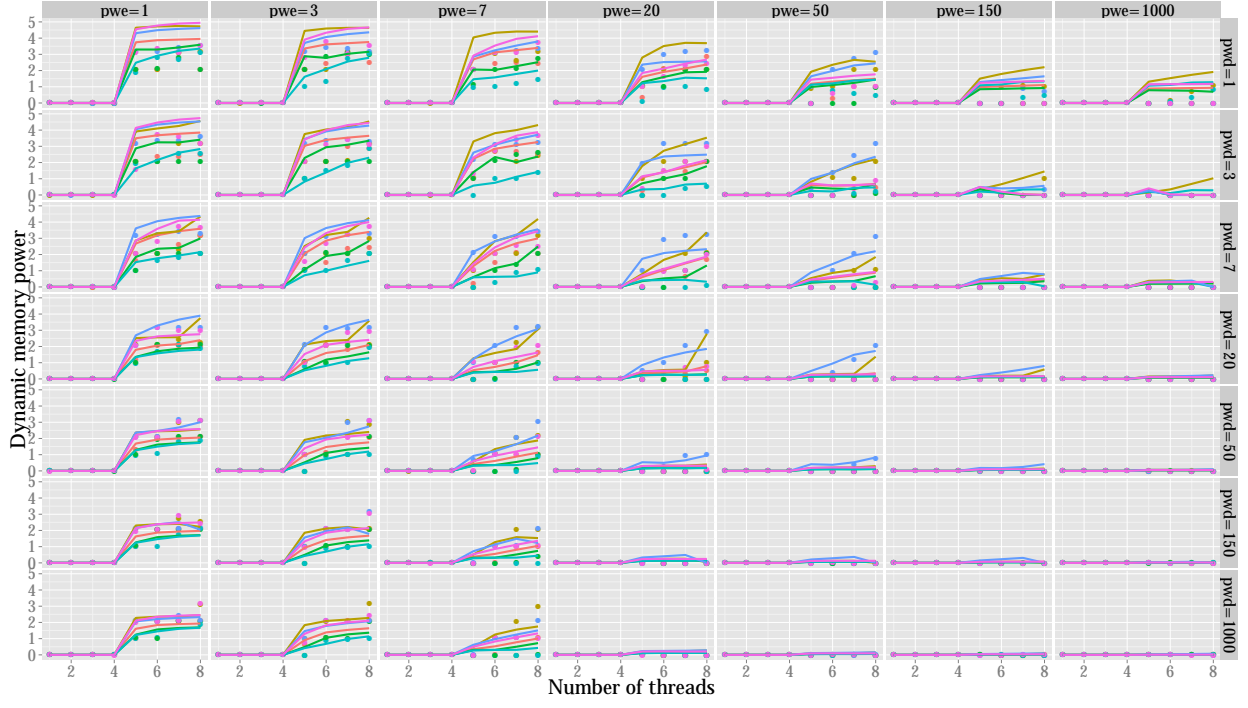


Fig. 19: Dynamic memory power at  $f = 3.4$  GHz

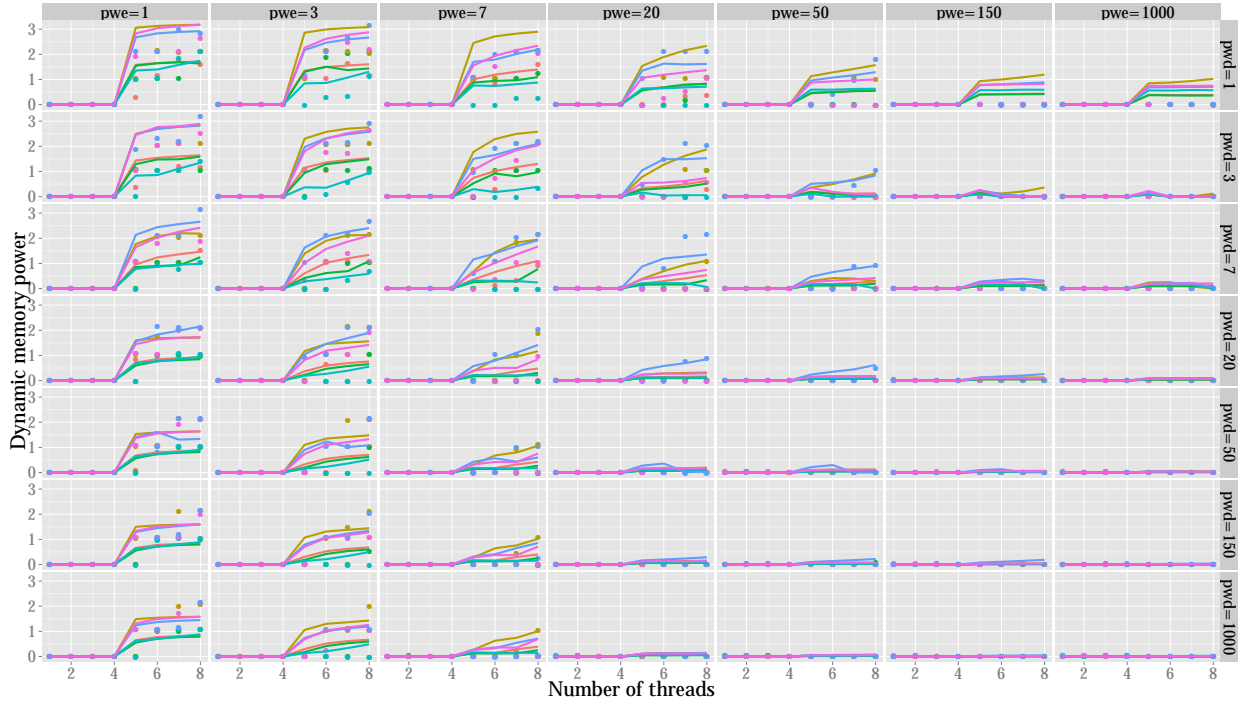


Fig. 20: Dynamic memory power at  $f = 1.2$  GHz

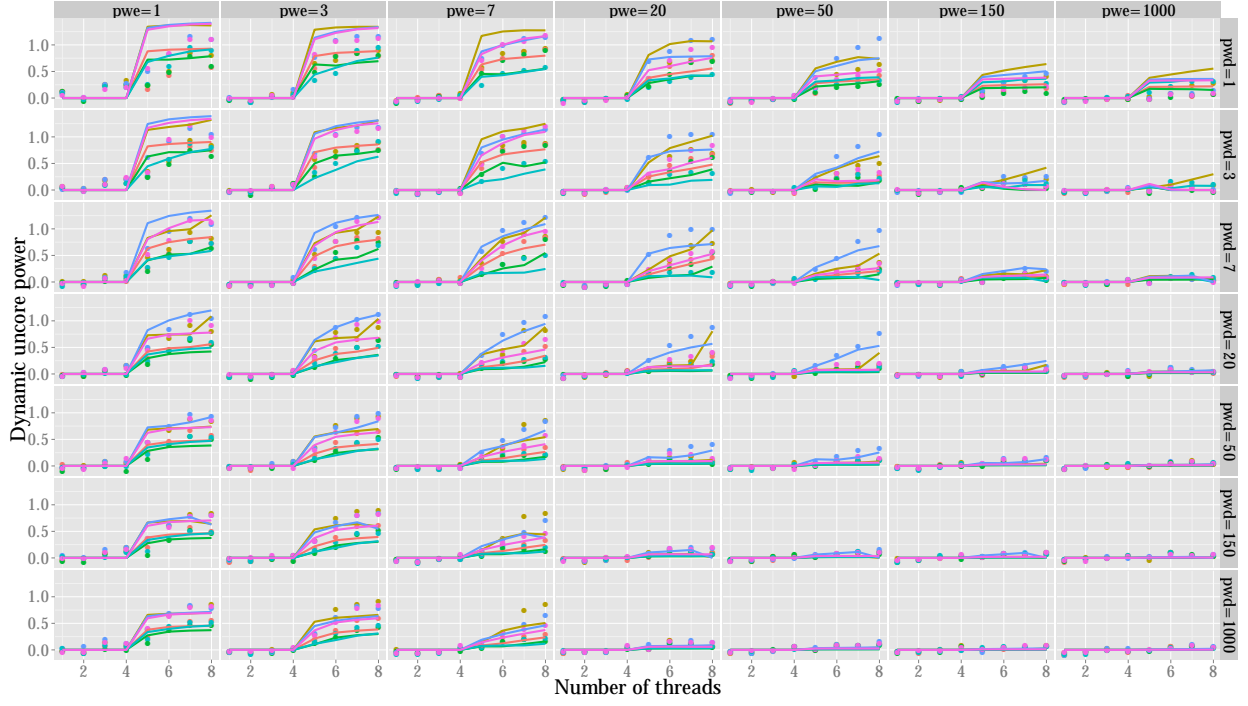


Fig. 21: Dynamic uncore power at  $f = 3.4$  GHz

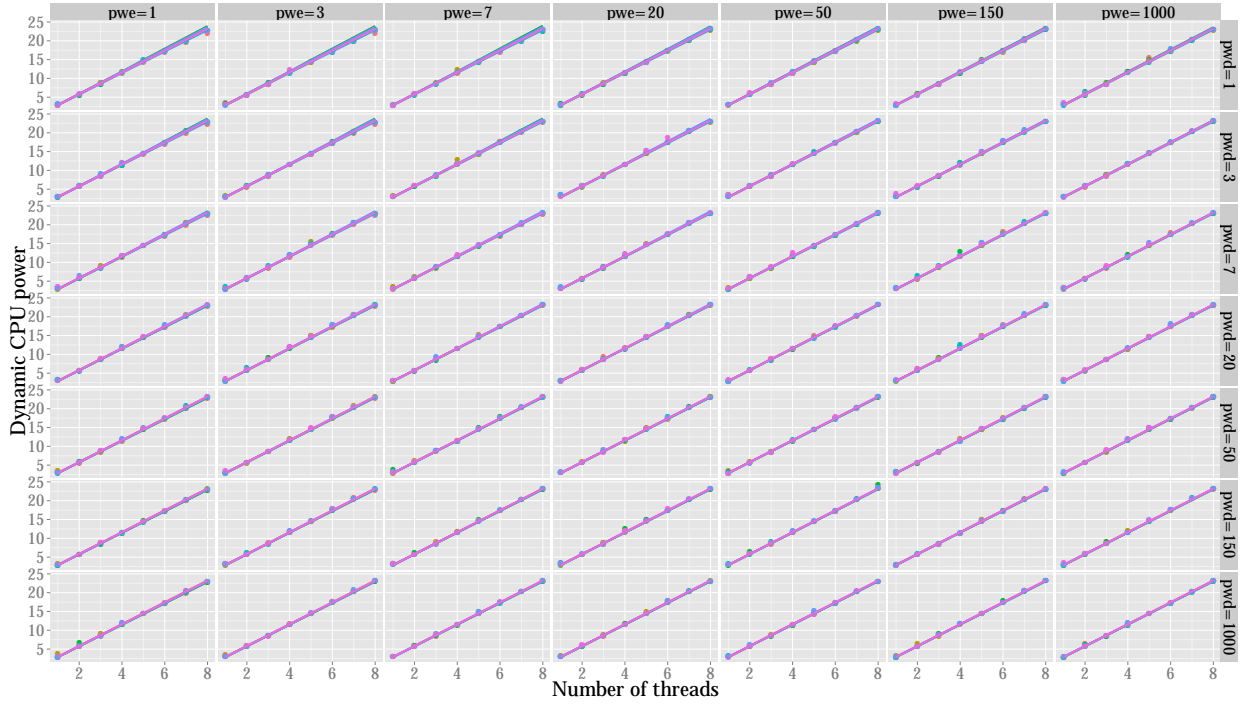


Fig. 22: Dynamic CPU power at  $f = 1.2$  GHz

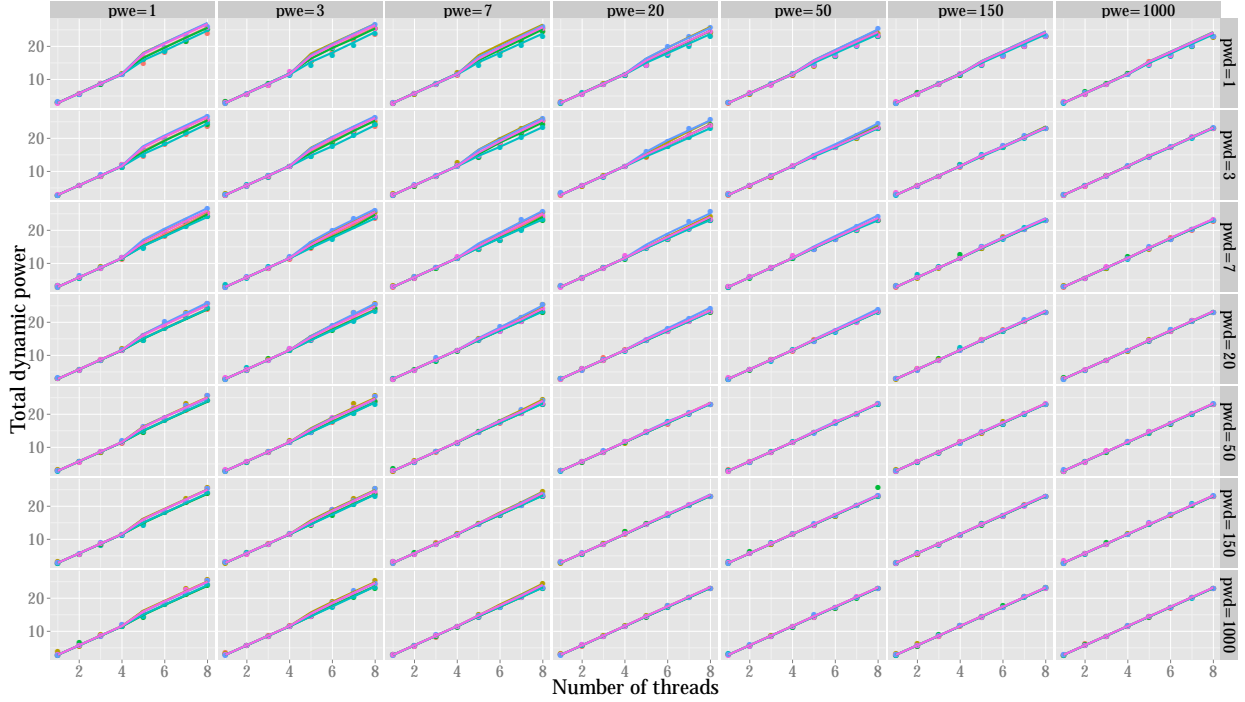


Fig. 23: Sum of dynamic powers at  $f = 1.2$  GHz

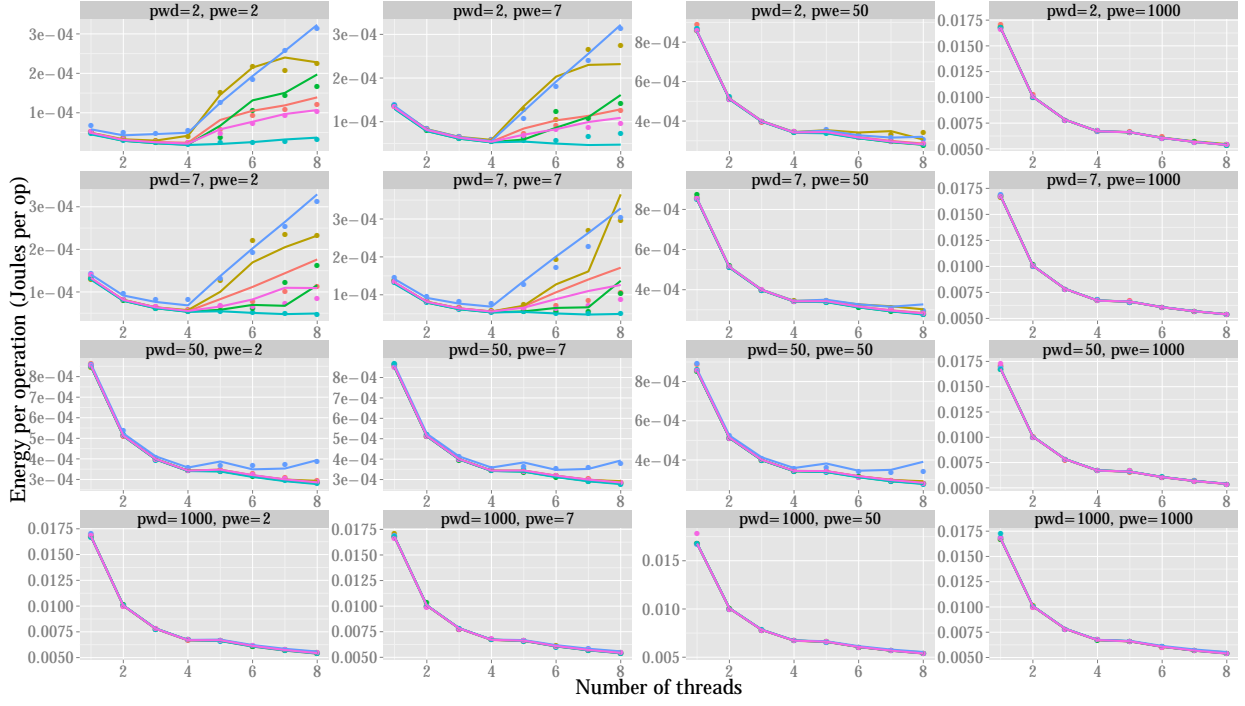
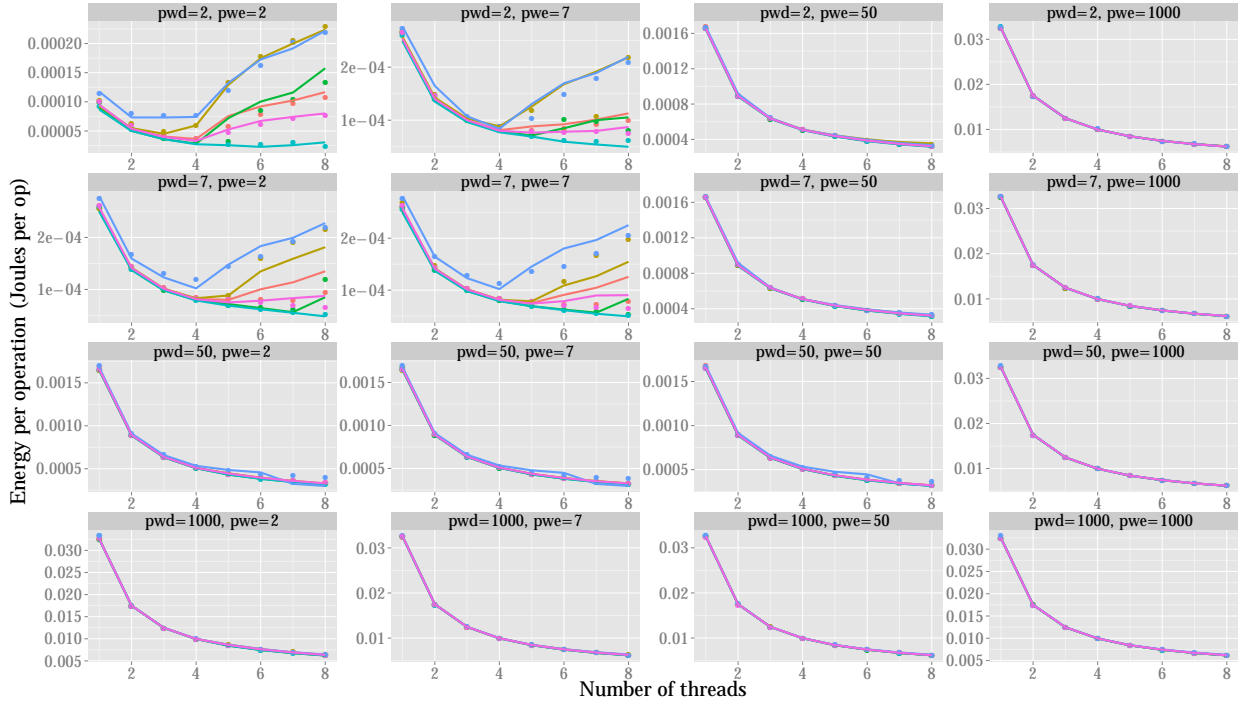
e.g. **Hof** detached from the others when  $pw_e = 50$  or **Gid** mostly well-predicted both absolutely and relatively as the less power-dissipating implementation. One can observe once again the asymmetry between enqueue and dequeue operations by comparing the power values at  $(pw_d, pw_e) = (2, 1000)$  and  $(1000, 2)$ ; this asymmetry is predicted by the model, with a lower impact though.

## VII. ENERGY PER OPERATION ESTIMATION

In Figures 24 and 25 is represented the energy per operation. Overall we observe that the successful operations (dequeue of a non-NULL item) are cheaper and cheaper when the number of threads is increasing on the same socket: the cost of turning the machine on is made profitable by an increase in performance. However, under high-contention, the lack of performance improvement while increasing the number of cores makes the use of supplementary cores useless. The inefficiency of adding cores is even more apparent when cores are spread across the sockets. In this case, under high-contention, performance could even be degraded by the implication of new cores, then, as performance decreases and power increases, the energy per operation dramatically increases.

## VIII. DESCRIPTION OF THE IMPLEMENTATIONS

1) **NOBLE** [1], [2]: Most of the implementations that we use are part of the NOBLE library. The NOBLE library offers support for non-blocking multi-process synchronization in shared memory systems. NOBLE has been designed in order to: i) provide a collection of shared data objects in a form which allows them to be used by non-experts, ii) offer an orthogonal support for synchronization where the developer can change synchronization implementations with minimal changes, iii) be easy to port to different multi-processor systems, iv) be adaptable for different programming languages and v) contain efficient known implementations of its shared data objects. The library provides a collection of the most

Fig. 24: Energy per operation at  $f = 3.4$  GHzFig. 25: Energy per operation at  $f = 1.2$  GHz

commonly used data types. The semantics of the components, which have been designed to be the very same for all implementations of a particular abstract data type, are based on the sequential semantics of common abstract data types and adopted for concurrent use. The set of operations has been limited to those which can be practically implemented using both non-blocking and lock-based techniques. Due to the concurrent nature, also new operations have been added, e.g. Update which cannot be replaced by Delete followed by Insert. Some operations also have stronger semantics than the corresponding sequential ones, e.g. traversal in a List is not invalidated due to concurrent deletes, compared to the iterator invalidation in STL. As the published algorithms for concurrent data structures often diverge from the chosen semantics, a large part of the implementation work in NOBLE, besides from adoption to the framework, also consists of considerable changes and extensions to meet the expected semantics.

The various lock-free concurrent queue algorithms that we include in this study are briefly described below.

2) *Tsigas-Zhang [3]*: Tsigas and Zhang [3] presented a lock-free extension of [23] for any number of threads where synchronization is done both on the array elements and the shared head and tail indices using CAS, and the ABA problem is avoided by exploiting two (or more) null values. In [3] synchronization is done both directly on the array elements and the shared head and tail indices using CAS<sup>2</sup>, thus supporting multiple producers and consumers. In order to avoid the ABA problem when updating the array elements, the algorithm exploits using two (or more) null values; the ABA problem is due to the inability of CAS to detect concurrent changes of a memory word from a value (A) to something else (B) and then again back to the first value (A). A CAS operation can not detect if a variable was read to be A and then later changed to B and then back to A by some concurrent processes. The CAS primitive will perform the update even though this might not be intended by the algorithm's designer. Moreover, for lowering the memory contention the algorithm alternates every other operation between scanning and updating the shared head and tail indices.

3) *Valois [4]*: Valois [4], [24] makes use of linked list in his lock-free implementation which is based on the CAS primitive. He was the first to present a lock-free implementation of a linked-list. The list uses auxiliary memory cells between adjacent pairs of ordinary memory cells. The auxiliary memory cells were introduced to provide an extra level of indirection so that normal memory cells can be removed by joining the auxiliary ones that are adjacent to them. His design also provides explicit cursors to access memory cells in the list directly and insert or delete nodes on the places the the cursors point to.

4) *Michael-Scott [5]*: Michael and Scott [5] presented a lock-free queue that is more efficient, synchronizing via the shared head and tail pointers as well as via the next pointer of the last node. Synchronization is done via shared pointers indicating the current head and tail node as well via the next pointer of the last node, all updated using CAS. The tail pointer is then moved to point to the new item, with the use of a CAS operation. This second step can be performed by the thread invoking the operation, or by another thread that needs to help the original thread to finish before it can continue. This helping behavior is an important part of what makes the queue lock-free, as a thread never has to wait for another thread to finish. The queue is fully dynamic as more nodes are allocated as needed when new items are added. The original presentation used unbounded version counters, and therefore required double-width CAS which is not supported on all contemporary platforms. The problem with the version counters can easily be avoided by using some memory management scheme as e.g. [25].

5) *Moir-et al. [6]*: Moir et al. [6] presented an extension of the Michael and Scott [5] lock-free queue algorithm where elimination is used as a back-off strategy to increase scalability when contention on the queue's head or tail is noticed via failed CAS attempts. However, elimination is only possible when the queue is close to empty during the operation's invocation.

<sup>2</sup>The Compare-And-Swap (CAS) atomic primitive will update a given memory word, if and only if the word still matches a given value (e.g. the one previously read). CAS is generally available in contemporary systems with shared memory, supported mostly directly by hardware and in other cases in combination with system software.

6) *Hoffman-Shalev-Shavit [7]*: Hoffman *et al.* [7] takes another approach in their design in order to increase scalability by allowing concurrent `Enqueue` operations to insert the new node at adjacent positions in the linked list if contention is noticed during the attempted insert at the very end of the linked list. To enable these "baskets" of concurrently inserted nodes, removed nodes are logically deleted before the actual removal from the linked list, and as the algorithm traverses through the linked list it requires stronger memory management than [25], such as [26] or [27] and a strategy to avoid long chains of logically deleted nodes.

7) *Gidenstam-Sundell-Tsigas [8]*: Gidenstam *et al.* [8] combines the efficiency of using arrays and the dynamic capacity of using linked lists, by providing a lock-free queue based on linked lists of arrays, all updated using `CAS` in a cache-aware manner. In resemblance to [3], [23], [28] this algorithm uses arrays to store (pointers to) the items, and in resemblance to [3] it uses `CAS` and two null values. Moreover, shared indices [28] are avoided and scanning [3] is preferred as much as possible. In contrast to [3], [23], [28] the array is not static or cyclic, but instead more arrays are dynamically allocated as needed when new items are added, making the queue fully dynamic.

## IX. 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 enqueueers 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<sup>3</sup> that computes and renders an  $8192 \times 8192$  pixel image of the Mandelbrot set [30] 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. The colors for the corresponding number of iterations are chosen according to a rainbow scheme, where low numbers are rendered within the red and high numbers are rendered within the violet spectrum.

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  $16 \times 16$ ,  $8 \times 8$ ,  $4 \times 4$ , or  $2 \times 2$  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. This is just one of many possible ways to divide the work and pin threads, it remains as future work to explore other ways.

<sup>3</sup>Previously used for evaluation in [29].



### 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 ( $2 \times 2$ ,  $4 \times 4$ ,  $16 \times 16$ ). For this process, we make use of CPUID, RDTSC and RDTSCP instructions as specified in [31]. 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  $2 \times 2$  contention, due to the wide distribution, we gather the parallel works into bins of width 10 pauses; the number of elements in the  $i^{\text{th}}$  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:

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

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

In Figure 26, 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  $2 \times 2$ , 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  $4 \times 4$  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.

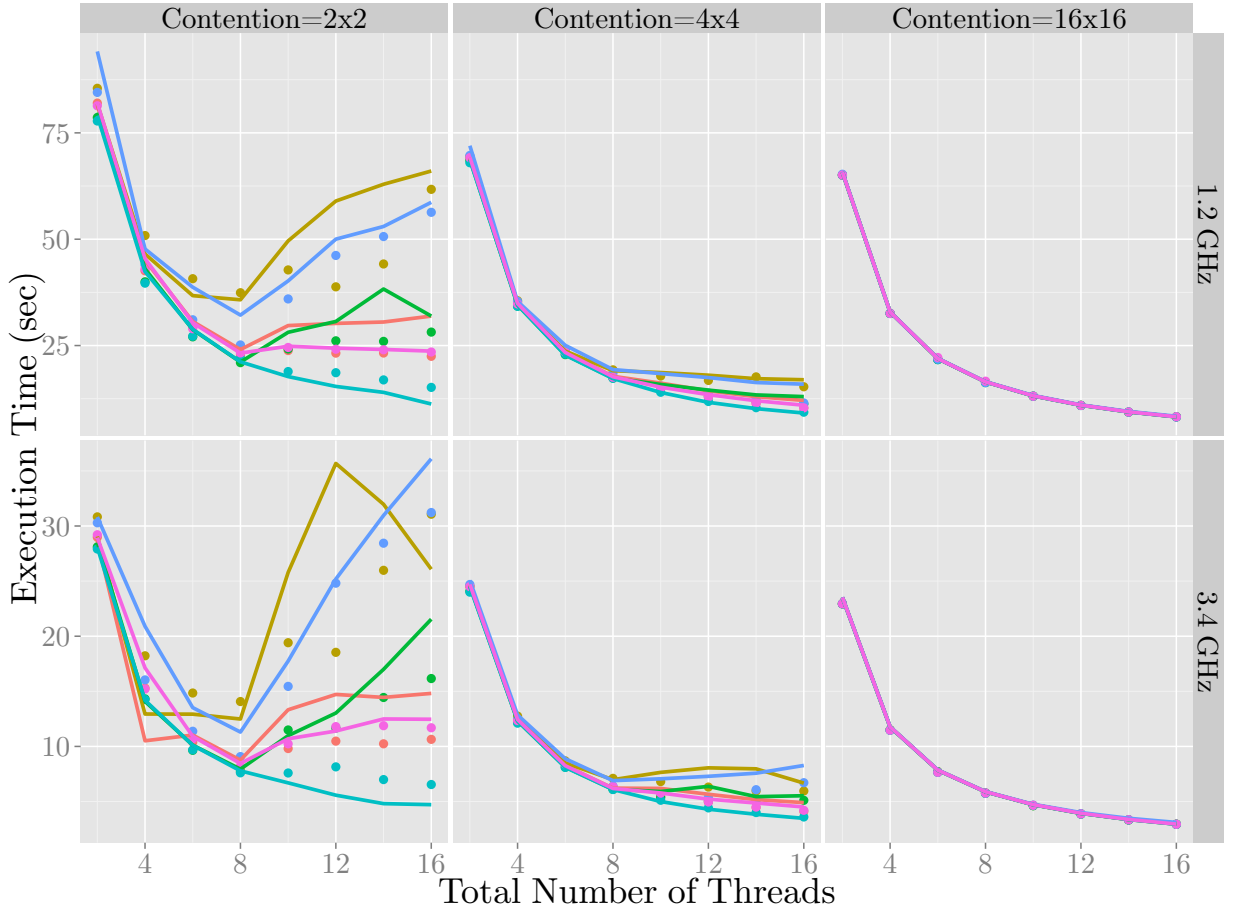


Fig. 26: Mandelbrot Execution Time

Power estimates are quite satisfactory except algorithm **Hof** which is overestimated. In the power versus time plot which is not presented here, we observe a step like decrease in power at the end of

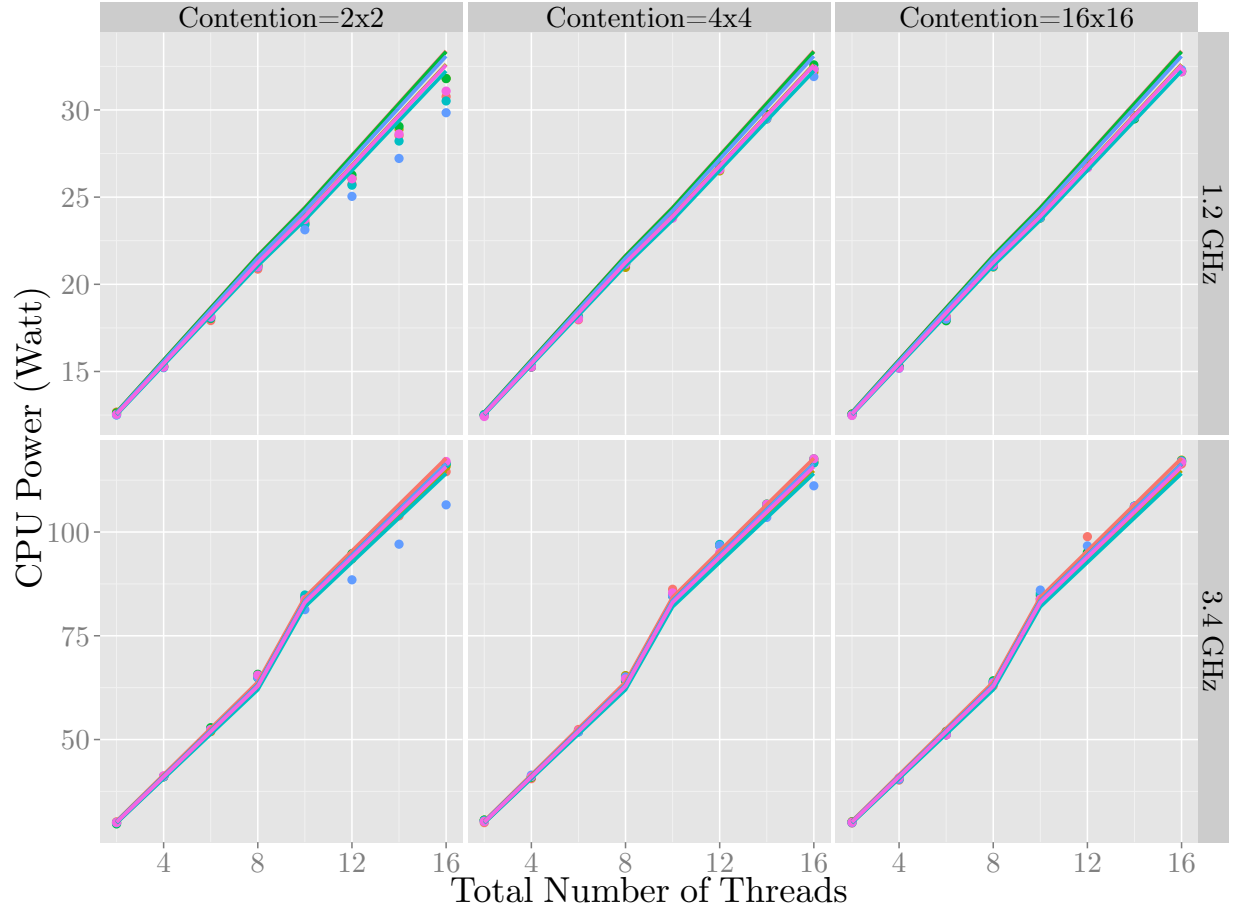


Fig. 27: Mandelbrot CPU power

the execution, implying that **Hof** is prone to unfairness among producers. Some producers finish their regions early and go to sleep which decreases the power dissipation.

As mentioned before, dynamic memory and uncore power are dominated by the queue implementations so we do not use any application specific memory/uncore power samples in our estimations, due to compute intensive character of the Mandelbrot parallel works. Even if this was not the case, memory/uncore power in the parallel sections could have been extracted. One can get the memory/uncore power measurement from the application and subtract the memory/uncore power that we have measured and extrapolated in the synthetic benchmark. Then, using the ratio of retry loops and parallel sections thanks to our throughput model, the memory/uncore power can be estimated.

Similar to the synthetic benchmarks, Mandelbrot dynamic memory/uncore power becomes noticeable with the inter-socket communication, after 8 threads, and decreases gradually with the decreasing ratio of retry loops, with contention level.

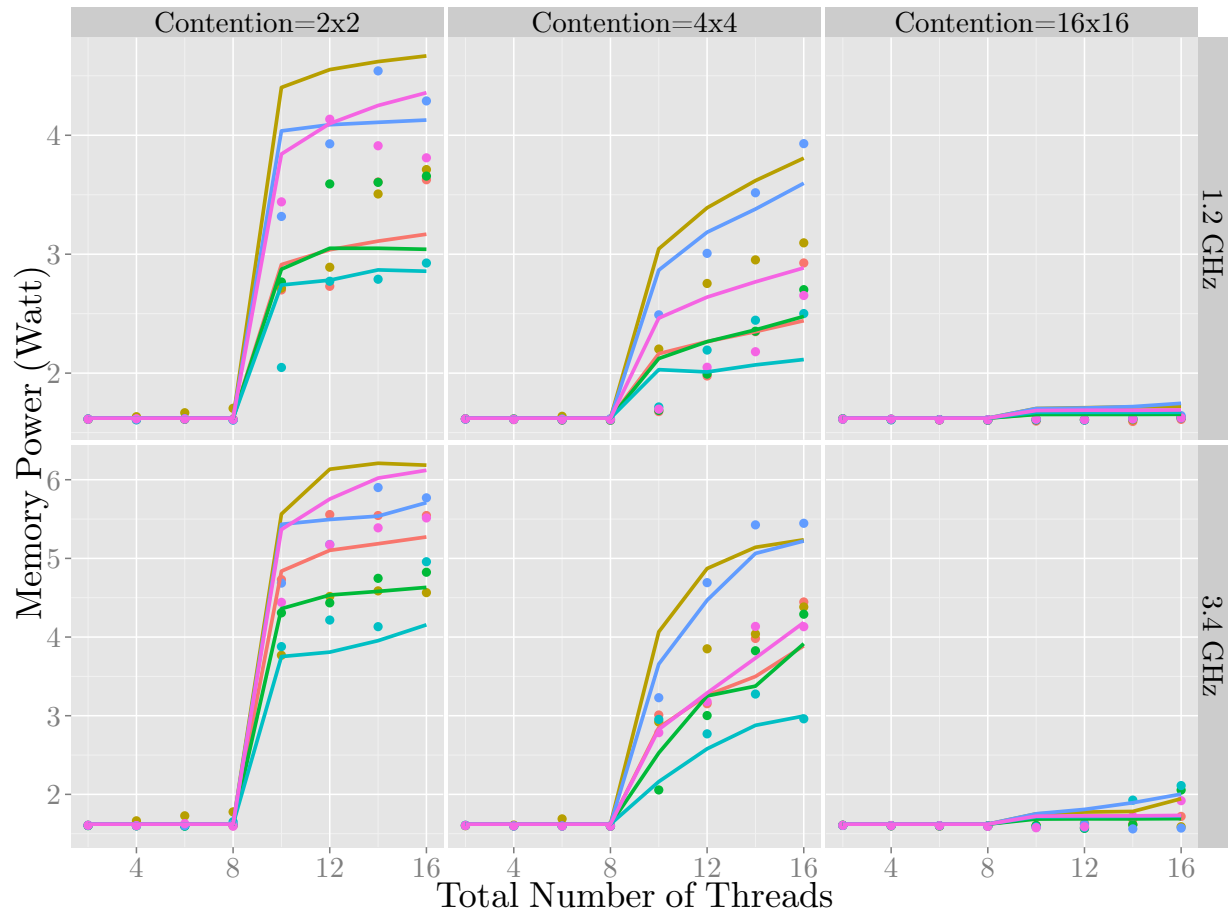


Fig. 28: Mandelbrot Memory Power

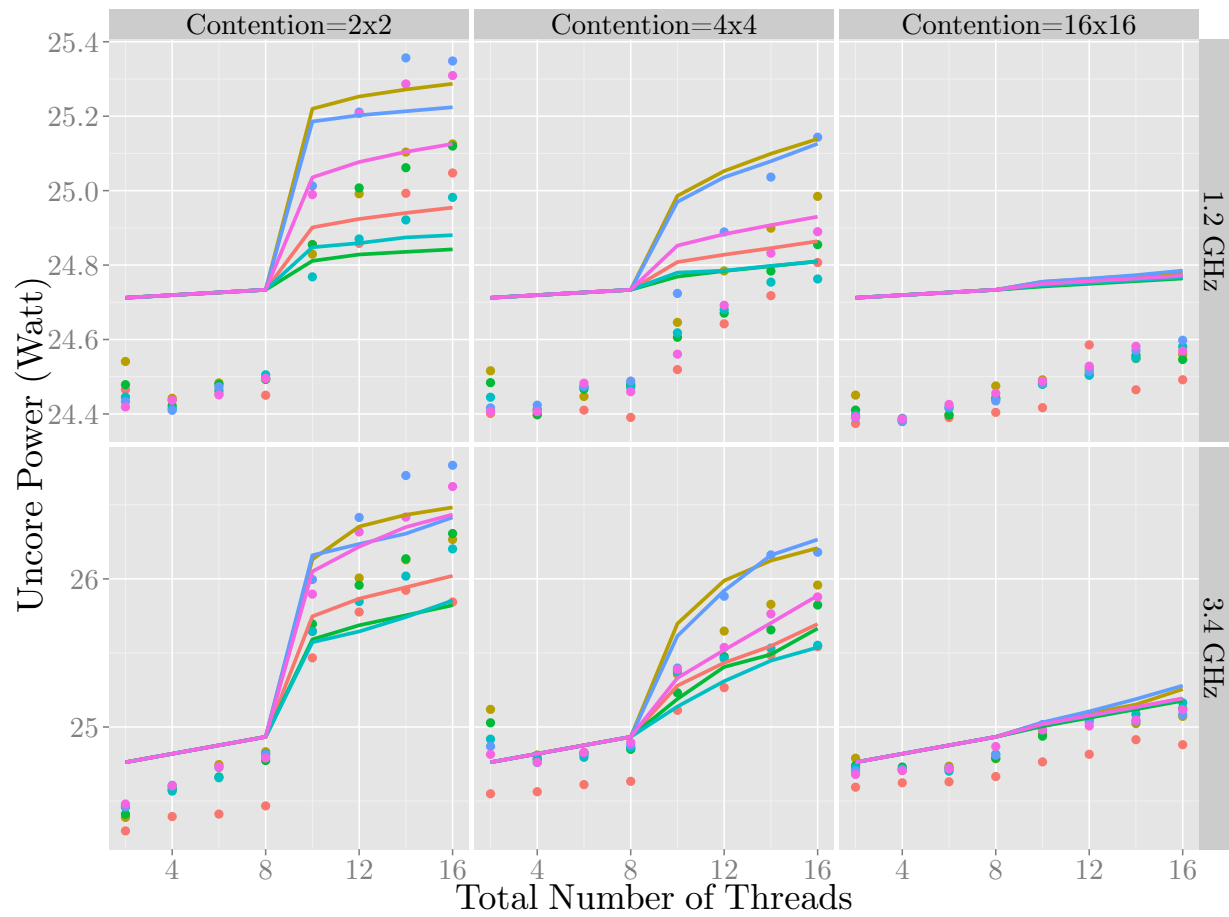


Fig. 29: Mandelbrot Uncore Power

## X. 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] H. Sundell and P. Tsigas, “NOBLE: A Non-Blocking Inter-Process Communication Library,” in *Proceedings of the Workshop on Languages, Compilers and Run-time Systems for Scalable Computers (LCR)*, ser. Lecture Notes in Computer Science, 2002.
- [2] —, “NOBLE: Non-blocking programming support via lock-free shared abstract data types,” *SIGARCH Computer Architecture News*, vol. 36, no. 5, 2008.
- [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] J. D. Valois, “Implementing Lock-Free Queues,” in *Proceedings of International Conference on Parallel and Distributed Systems (ICPADS)*, December 1994, pp. 64–69.
- [5] 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.
- [6] 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.
- [7] 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.
- [8] 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.
- [9] 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.
- [10] 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.
- [11] 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.
- [12] 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.
- [13] 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.
- [14] 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.
- [15] 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.
- [16] 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.
- [17] 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.

- [18] H. David, E. Gorbato, 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.
- [19] 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.
- [20] 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.
- [21] J. R. Goodman and H. H. J. Hum, "Mesif: A two-hop cache coherency protocol for point-to-point interconnects," University of Auckland, Tech. Rep., November 2009. [Online]. Available: <http://hdl.handle.net/2292/11594>
- [22] H. David, E. Gorbato, U. R. Hanebutte, R. Khanaa, 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.
- [23] L. Lamport, "Specifying concurrent program modules," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 5, no. 2, pp. 190–222, 1983.
- [24] J. D. Valois, "Lock-Free Data Structures," Ph.D. dissertation, Rensselaer Polytechnic Institute, Troy, New York, 1995.
- [25] M. M. Michael, "Hazard pointers: Safe memory reclamation for lock-free objects," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 15, no. 8, August 2004.
- [26] A. Gidenstam, M. Papatriantafilou, H. Sundell, and P. Tsigas, "Efficient and reliable lock-free memory reclamation based on reference counting," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 20, no. 8, pp. 1173–1187, 2009.
- [27] M. Herlihy, V. Luchangco, P. Martin, and M. Moir, "Nonblocking Memory Management Support for Dynamic-sized Data Structures," *ACM Transactions on Computer Systems (TOCS)*, vol. 23, pp. 146–196, May 2005.
- [28] J. Giacomoni, T. Moseley, and M. Vachharajani, "Fastforward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue," in *Principles and Practice of Parallel Programming (PPoPP)*. New York, NY, USA: ACM, February 2008, pp. 43–52.
- [29] 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.
- [30] B. B. Mandelbrot, "Fractal aspects of the iteration of  $z \rightarrow \lambda z(1 - z)$  for complex  $\lambda$  and  $z$ ," *Annals of the New York Academy of Sciences*, vol. 357, pp. 249–259, 1980.
- [31] 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.