Available online at www.sciencedirect.com

ScienceDirect

Journal of Systems and Software 00 (2015) 1–16

JSS

A Comprehensive Study on the Energy Efficiency of Java Thread-Safe Collections

Gustavo Pinto[a,*], Kenan Liu[b], Fernando Castor[a], Yu David Liu[b]

[a]*Federal University of Pernambuco, Recife, PE, 50.740-560, Brazil*
[b]*SUNY Binghamton, Binghamton, NY 13902, USA*

Abstract

Java programmers are served with numerous choices of collections, varying from simple sequential ordered lists to more sophisticated, thread-safe, and highly scalable hashtable implementations. These choices are well-known to have different characteristics in terms of performance, scalability, and thread-safety, and most of them are well studied. This paper analyzes an additional dimension, *energy efficiency*. Through an empirical investigation of 16 collection implementations grouped under 3 commonly used forms of data structures (lists, sets, and mappings), we show that small design decisions can greatly impact energy consumption. Our study presents a number of findings. For example, different implementations of the same thread-safe collection can have widely different energy consumption behavior. This variation also applies to the different operations that each collection implements, e.g., a collection implementation that performs traversals very efficiently can be more than an order of magnitude less efficient than another implementation of the same collection when it comes to insertions. Hence, to save energy, developers must weight the impact of the operations that an application will perform before choosing a collection implementation. The study serves as a first step toward understanding the energy efficiency of Java collections on parallel architectures.

Keywords: Energy Efficiency, Performance, Java Collections, Parallelism

1. Introduction

A question that often arises in software development forums is: "since Java has so many collection implementations, which one is more suitable to my problem?"[1]. Answers to this question come in different flavors: these collections serve for different purposes and have different characteristics in terms of performance, scalability and thread-safety. Developers should consider these characteristics in order to make judicious design decisions about which implementation best fits their problems. In this study, we consider one additional attribute: *energy efficiency*. In an era where mobile platforms are prevalent, there is considerable evidence that battery usage is a key factor for evaluating and adopting mobile applications [1]. Energy consumption estimation tools do exist [2, 3, 4], but they do not provide direct guidance on *energy optimization*, *i.e.*, bridging the gap between understanding where energy is consumed and understanding how the code can be modified in order to reduce energy consumption.

Traditionally addressed by hardware-level (*e.g.*, [5, 6]) and system-level approaches (*e.g.*, [7, 8]), energy optimization is gaining momentum in recent years through application-level software engineering techniques (*e.g.*, [9, 10, 11]).

*Corresponding author
Email addresses: ghlp@cin.ufpe.br (Gustavo Pinto), kliu20@binghamton.edu (Kenan Liu), castor@cin.ufpe.br (Fernando Castor), davidL@cs.binghamton.edu (Yu David Liu)

[1] http://stackoverflow.com/search?q=which+data+structure+use+java+is:question

The overarching premise of this crescent direction is that the high-level knowledge from software engineers on application design and implementation can make significant impact on energy consumption, as confirmed by recent empirical studies [12, 13]. The space for application-level energy optimization, however, is diverse. Developers sometimes rely on conventional wisdom, consult software development forums and blogs, or simply search online for "tips and tricks." Many of these guidelines are often anecdotal or even incorrect [14].

In this paper, we elucidate one important area of the application-level optimization space, focusing on understanding the energy consumption of different Java collections running on parallel architectures [15]. This is a critical direction at the junction of data-intensive computing and parallel computing, which deserves more investigation due to at least three reasons:

- Collections are one of the most important building blocks of computer programming. Multiplicity — a collection may hold many pieces of data items — is the norm of their use, and it often contributes to significant memory pressure — and performance problems in general — of modern applications where data are often intensive [16, 17].
- Not only high-end servers but also desktop machines, smartphones and tablets need concurrent programs to make best use of their multi-core hardware. A CPU with more cores (say 32) often consumes more power than one with fewer cores (say 1 or 2) [18].
- Mainstream programming languages often provide a number of implementations for the same collection and these implementations have potentially different characteristics in terms of energy efficiency.

We present an empirical study evaluating the performance and energy consumption characteristics of 16 Java collection implementations grouped by 3 well-known interfaces: `List`, `Set`, and `Map`. Through experiments conducted in a multi-core environment, we correlate energy behaviors of different thread-safe implementations of Java collections and their knobs. Our research is motivated by the following questions:

RQ1. Do different implementations of the same collection have different impacts on energy consumption?

RQ2. Do different operations in the same implementation of a collection consume energy differently?

RQ3. Do collections scale, from an energy consumption perspective, with an increasing number of concurrent threads?

RQ4. Do different collection configurations and usages have different impacts on energy consumption?

The goal of this study is to answer these research questions. In order to answer **RQ1** and **RQ2**, we select and analyze the behaviors of three common operations — traversal, insertion and removal — for each collection implementation. To answer **RQ3**, we analyze how different implementations scale in the presence of multiple threads. In this experiment, we cover the spectrum including both under-provisioning (the number of threads exceeds the number of CPU cores) and over-provisioning (the number of CPU cores exceeds the number of threads). In **RQ4**, we analyze how different configurations — such as the load factor and the initial capacity of the collection — impact energy consumption. To gain confidence in our results in the presence of platform variations and measurement environments, we employ two machines with different architectures (a 32-core AMD vs. a 16-core Intel). We further use two distinct energy measurement strategies: an external energy meter, and Machine-Specific Registers (MSRs).

Our study produces a list of interesting findings, some of which are not obvious. We summarize them in Section 3, at the end of each RQ's discussion. To highlight one of them, our experiments show that execution time is not always a reliable indicator for energy consumption. This is particularly true for various `Map` implementations. In other words, the consumption of power — the rate of energy consumption — is not a constant across different collection implementations.

2. Study Setup

In this section we describe the benchmarks that we analyzed, the infrastructure and the methodology that we used to perform the experiments.

2.1. Benchmarks

The benchmarks used in this study consist of 16 commonly used collections available in the Java programming language. Our focus is on the thread-safe implementations of the collection. Hence, for each collection, we selected a single non-thread-safe implementation to serve as a baseline. For each implementation, we analyzed insertion, removal and traversal operations. We grouped these implementations by the logical collection they represent, into three categories:

`Lists (java.util.List)`: `Lists` are ordered collections that allow duplicate elements. Using this collection, programmers can have precise control over where an element is inserted in the list. The programmer can access an element using its index, or traverse the elements using an `Iterator`. Several implementations of this collection are available in the Java language. We used `ArrayList`, which is not thread-safe, as our baseline. We studied the following thread-safe `List` implementations: `Vector`, `Collections.synchronizedList()`, and `CopyOnWriteArrayList`. The main difference between `Vector` and `Collections.synchronizedList()` is their usage pattern in programming. With `Collections.synchronizedList()`, the programmer creates a wrapper around the current `List` implementation, and the data stored in the original `List` object does not need to be copied into the wrapper object. It is appropriate in cases where the programmer intends to hold data in a non-thread-safe `List` object, but wishes to add synchronization support. With `Vector`, on the other hand, the data container and the synchronization support are unified so it is not possible to keep an underlying structure (such as `LinkedList`) separate from the object managing the synchronization. `CopyOnWriteArrayList` creates a copy of the underlying `ArrayList` whenever a mutation operation (*e.g.*, using the `add` or `set` methods) is invoked.

`Maps (java.util.Map)`: `Maps` are objects that map keys to values. Logically, the keys of a map cannot be duplicated. Each key is uniquely associated with a value. An insertion of a (key, value) pair where the key is already associated with a value in the map results in the old value being replaced by the new one. Our baseline thread-unsafe choice is `LinkedHashMap`, instead of the more commonly used `HashMap`. This is because the latter sometimes caused non-termination during our experiments[2]. Our choice of thread-safe `Map` implementations includes `Hashtable`, `Collections.synchronizedMap()`, `ConcurrentSkipListMap`, `ConcurrentHashMap`, and `ConcurrentHashMapV8`. The difference between `ConcurrentHashMap` and `ConcurrentHashMapV8` is that the latter is an optimized version released in Java 1.8, while the former is the version present in the JDK until Java 1.7. While all `Map` implementations share similar functionalities and operate on a common interface, they are particularly known to differ in the order of element access at iteration time. For instance, while `LinkedHashMap` iterates in the order in which the elements were inserted into the map, a `Hashtable` makes no guarantees about the iteration order.

`Sets (java.util.Set)`: As its name suggests, the `Set` collection models the mathematical set abstraction. Unlike `Lists`, `Sets` do not count duplicate elements, and are not ordered. Thus, the elements of a set cannot be accessed by their indices, and traversals are only possible using an `Iterator`. Among the available implementations, we used `LinkedHashSet`, which is not thread-safe, as our baseline. Our selection of thread-safe `Set` implementations includes `Collections.synchronizedSet()`, `ConcurrentSkipListSet`, `ConcurrentHashSet`, `CopyOnWriteArraySet`, and `ConcurrentHashSetV8`. It should be noted that both `ConcurrentHashSet` and `ConcurrentHashSetV8` are not top-level classes readily available in the JDK library. Instead, they are supported through the returned object from `Collections.newSetFromMap(new ConcurrentHashMap<String,String>())` or the analogous implementation `Collections.newSetFromMap(new ConcurrentHashMapV8<String,String>())` respectively. The returned `Set` object observes the same ordering as the underlying map.

2.2. Experimental Environment

To gain confidence in our results in the presence of platform variations, we run each experiment on two significantly different platforms:

- `System#1`: A 2×16-core AMD Opteron 6378 processor (Piledriver microarchitecture), 2.4GHz, with 64GB of DDR3 1600 memory. It has three cache levels (L1, L2, L3): L1 with 32KB per core, L2 with 256KB per core,

[2] A possible explanation can be found here: http://mailinator.blogspot.com/2009/06/beautiful-race-condition.html

and L3 20480 (Smart cache). It is running Debian 3.2.46-1 x86-64 Linux (kernel 3.2.0-4-amd64), and Oracle HotSpot 64-Bit Server VM (build 21) JDK version 1.7.0_11.

- `System#2`: A 2×8-core (32-cores when hyper-threading is enabled) Intel(R) Xeon(R) E5-2670 processor, 2.60GHz, with 64GB of DDR3 1600 memory. It has three cache levels (L1, L2, L3): L1 with 48KB per core, L2 with 1024KB per core, and L3 20480 (Smart cache). It is running Debian 6 (kernel 3.0.0-1-amd64) and Oracle HotSpot 64-Bit Server VM (build 14), JDK version 1.7.0_71.

When we performed the experiments with `Sets` and `Maps`, we employed the jsr166e library[3], which contains the `ConcurrentHashMapV8` implementation. Thus, these experiments do not need to be executed under Java 1.8.

We also used two different energy consumption measurement approaches. For `System#1`, energy consumption is measured through current meters over power supply lines to the CPU module. Data is converted through an NI DAQ and collected by NI LabVIEW SignalExpress with 100 samples per second and the unit of the current sample is *deca-ampere* (10 ampere). Since the supply voltage is stable at 12V, energy consumption is computed as the sum of current samples multiplied by $12 \times 0.01 \times 10$. We measured the "base" power consumption of the OS when there is no JVM (or other application) running. The reported results are the measured results *modulo* the "base" energy consumption.

For `System#2`, we have used `jRAPL` [4], which is a framework that contains a set of APIs for profiling Java programs running on CPUs with Running Average Power Limit (RAPL) [6] support. Originally designed by Intel for enabling chip-level power management, RAPL is widely supported in today's Intel architectures, including Xeon server-level CPUs and the popular i5 and i7 processors. RAPL-enabled architectures monitor the energy consumption information and store it in Machine-Specific Registers (MSRs). Due to architecture design, the RAPL support for `System#2` can access CPU core, CPU uncore data (*i.e.* caches and interconnects), and in addition DRAM energy consumption data. RAPL-based energy measurement has appeared in recent literature (*e.g.*, [19, 20]); its precision and reliability have been extensively studied [21].

As we shall see in the experiments, DRAM power consumption is nearly constant. In other words, even though our meter-based measurement strategy only considers the CPU energy consumption, it is still indicative of the relative energy consumptions of different collection implementations. It should be noted that the stability of DRAM power consumption through RAPL-based experiments does not contradict the established fact that the energy consumption of memory systems is highly dynamic [22]. In that context, memory systems subsume the entire memory hierarchy, and most of the variations are caused by caches [23] — part of the "CPU uncore data" in our experiments.

All experiments were performed with no other load on the OS. We conform to the default settings of both the OS and the JVM. Several default settings are relevant to this context: (1) the power management of Linux is the default *ondemand* governor, which dynamically adjusts CPU core frequencies based on system workloads. (2) For the JVM, the parallel garbage collector is used, and just-in-time (JIT) compilation is enabled. The initial heap size and maximum heap size are set to be 1GB and 16GB respectively. We run each benchmark 10 times within the same JVM; this is implemented by a top-level 10-iteration loop over each benchmark. The reported data is the average of the last 3 runs. We chose the last three runs because, according to a recent study, JIT execution tends to stabilize in the latter runs [13]. Hyper-threading is enabled and turbo Boost feature is disabled on `System#2`.

3. Study Results

In this section, we report the results of our experiments. Results for **RQ1** and **RQ2** are presented in Section 3.1, describing the impact of different implementations and operations on energy consumption. In Section 3.2 we answer **RQ3** by investigating the impact of accessing data collections with different numbers of threads. Finally, in Section 3.3 we answer **RQ4** by exploring different "tuning knobs" of data collections.

3.1. Energy Behaviors of Different Collection Implementations and Operations

For **RQ1** and **RQ2**, we set the number of threads to 32 and, for each group of collections, we performed and measured insertion, traversal and removal operations.

[3] Source code available at: http://gee.cs.oswego.edu/cgi-bin/viewcvs.cgi/jsr166/src/jsr166e/

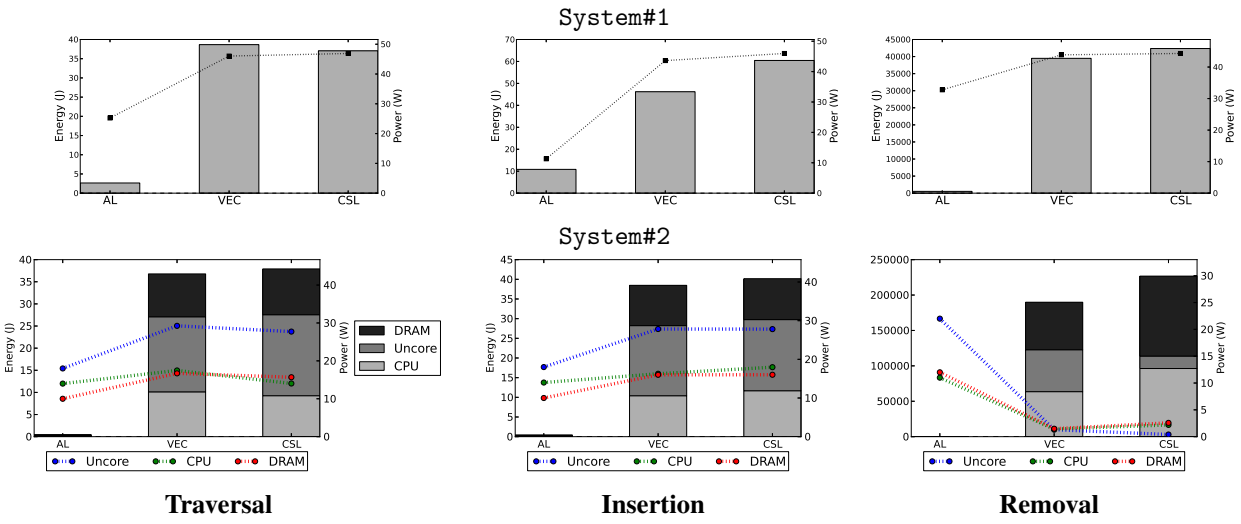

Figure 1. Energy and power results for traversal, insertion and removal operations for different `List` implementations. Bars denote energy consumption and lines denote power consumption. `AL` means `ArrayList`, `VEC` means `Vector`, and `CSL` means `Collections.synchronizedList()`.

- For the insertion operation, we start with an empty data collection, and have each thread insert 100,000 elements. Hence, at the end of the insertion operation, the total number of elements inside the collection is 3,200,000. To avoid duplicate elements, each insertion operation adds a `String` object with value *thread-id* + "-" + *current-index*.

- For the traversal operation, each thread traverses the entire collection generated by the insertion operation, *i.e.*, over 3,200,000 elements. On `Sets` and `Maps`, we first get the list of keys inserted, and then we iterate over these keys in order to get their values. On `Lists`, the traversal operation is performed using a top-level loop over the collection, accessing each element by its index using the `E get(int i)` method of each collection class, where `E` is the generic type (instantiated to `String` in our experiments).

- For the removal operation, we start with the collection with 3,200,000 elements, and remove the elements one by one. For `Maps` and `Sets`, the removals are based on keys, and we remove until the collection becomes empty. On `Lists`, however, the removal operation is based on indexes, and occurs *in-place* — that is, we do not traverse the collection to look up for a particular element before removal.

  Here, according to the `List` documentation, the `E remove(int i)` method "*removes the element at the specified position in this list (optional operation). Shifts any subsequent elements to the left (subtracts one from their indices). Returns the element that was removed from the list.*"[4]. As we shall see, removal operations on `Lists` are excessively expensive. In order to make it feasible to perform all experiments, we chose to remove only half of the elements.

Lists. Figure 1 shows the energy consumption (bars) and power consumption (lines) results of our `List` experiments. Each bar represents one `List` implementation. The three graphs at top of the figure are collected from `System#1`, whereas the three graphs in the bottom are from `System#2`. We do not show the figures for `CopyOnWriteArrayList` because the results for insertion and removal are an outlier and would otherwise skew the proportion of the figures.

First, we can observe that synchronization does play an important role here. As we can see, `ArrayList`, the non-thread-safe implementation, consumes much more energy than the other ones, thanks to its lack of synchronization. `Vector` and `Collection.synchronizedList()` are similar in energy behaviors. The greatest difference is seen on insertion, on `System#1`, in which the former consumed about 24.21% less energy than the former. `Vector`

[4]Documentation available at http://docs.oracle.com/javase/7/docs/api/java/util/List.html#remove(int)

and `Collection.synchronizedList()` are strongly correlated in their implementations, with some differences. While both of them are thread-safe on insertion and removal operations, `Collection.synchronizedList()` is not thread-safe on traversals, when performing through an `Iterator`, whereas `Vector` is thread-safe on the `Iterator`. `CopyOnWriteArrayList`, in contrast, is thread-safe in all operations. However, it does not need synchronization on traversal operations, which makes this implementation more efficient than the thread-safe ones (it consumes 46.38x less energy than `Vector` on traversal).

Furthermore, different operations can have different impacts. As we can see on traversal, the `Vector` implementation presents the worst result among the benchmarks: it consumes 14.58x more energy and 7.9x more time than the baseline on `System#1`(84.65x and 57.99x on `System#2`, respectively). This is due both `Vector` and `Collection.synchronizedList()` implementations need to synchronize in traversal operations. As mentioned contrast, the `CopyOnWriteArrayList` implementation is more efficient than the thread-safe implementation.

For insertion operations, `ArrayList` consumes the least energy for both `System#1` and `System#2`. When comparing the thread-safe implementations, `Collections.synchronizedList()` consumes 1.30x more energy than `Vector` (1.24x for execution time) on `System#1`. On `System#2`, however, they consume barely the same amount of energy (`Collections.synchronizedList()` consumes 1.01x more energy than `Vector`). On the other hand, `CopyOnWriteArrayList` consumes a total of 6,843.21 J, about 152x more energy than `Vector` on `System#1`. This happens because, for each new element added to the list, the `CopyOnWriteArrayList` implementation needs to synchronize and create a fresh copy of the underlying array using the `System.arraycopy()` method. As discussed elsewhere [13, 24], even though the `System.arraycopy()` behavior can be observed in sequential applications, it is more evident in highly parallel applications, when several processors are busy making copies of the collection, preventing them from doing important work. Although this behavior makes this implementation thread-safe, it is ordinarily too costly to maintain the collection in a highly concurrent environment where insertions are not very rare events.

Moreover, removals usually consumes much more energy than the other operations. For instance, removal on `Vector` consumes about 778.88x more energy than insertion on `System#1`. Execution time increases similarly, for instance, it took about 92 seconds to complete a removal operation on `Vector`. By way of contrast, insertions on a `Vector` takes about 1.2 seconds. We believe that several reasons can explain this behavior. First, the removal operations need to compute the size of the collection in each iteration of the for loop and, as we shall see in Section 3.4, such naive modification can greatly impact both performance and energy consumption. The second reason is that each call to the `List.remove()` method leads to a call to the `System.arrayCopy()` method in order to resize the `List`, since all these implementations of `List` are built upon arrays. In comparison, insertion operations only lead to a `System.arrayCopy()` call when the maximum number of elements is reached.

Power consumption also deserves attention. Since `System.arrayCopy()` is a memory intensive operation, power consumption decreases, and thus, execution time increase. Moreover, for most cases, power consumption follows the same shape of energy. Since energy consumption is the product of power consumption and time, when power consumption decreases and energy increase, execution time tends to increase. This is what happens on removal on `System#2`. The excessive memory operations on removals, also observed on DRAM energy consumption (the black top-most part of the bar), prevents the CPU to do useful work, which increases the execution time.

We also observed that the baseline benchmark on `System#2` consumes the least energy when compared to the baseline on `System#1`. We atribute that to our energy measurement approaches. While RAPL-based measurement can be efficient in retrieving only the necessary information (for instance, package energy consumption), our hardware-based measurement gathers energy consumption information pertaining to everything that happens in the CPU. Such noise can be particularly substantial when the execution time is small.

For all aforementioned cases, we observed that energy follows the same shape as time. At the first impression, this finding might seem to be "boring". However, recent studies have observed that energy and time are often not correlated [2, 13, 25], which is particularly true for concurrent applications. For this set of benchmarks, however, we believe that developers can safely use time as a proxy for energy, which can be a great help when refactoring an application to consume less energy.

Ultimately, although we have found some differences in the results, both `System#1` and `System#2` presented a compelling uniformity.

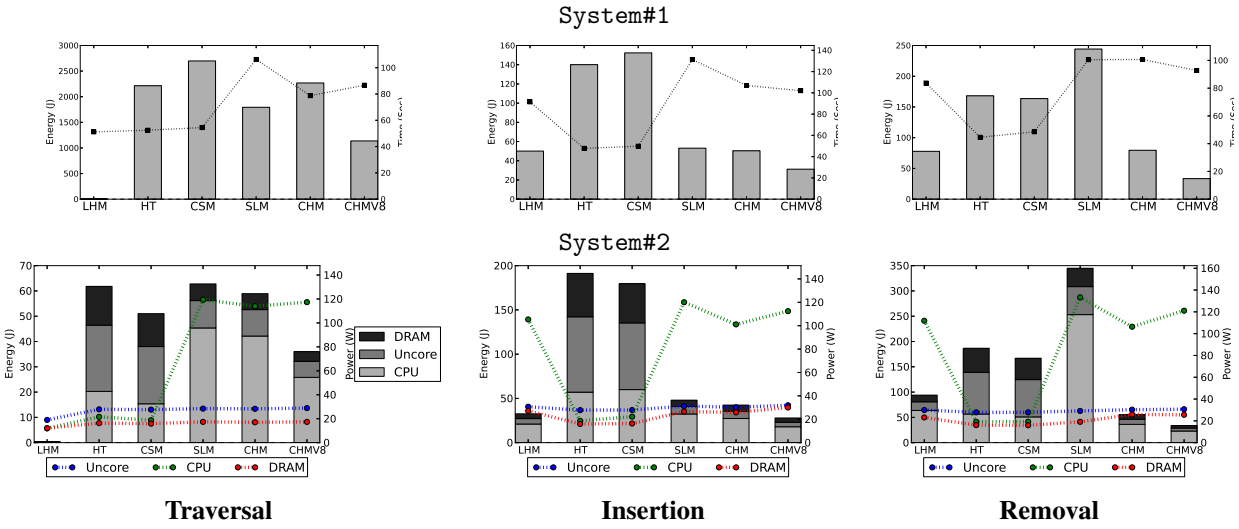

Figure 2. Energy and power results for traversal, insertion and removal operations for different `Map` implementations. Bars mean energy consumption and line means power consumption. `LSM` means `LinkedHashMap`, `HT` means `Hashtable`, `CSM` means `Collections.synchronizedMap()`, `SLM` means `ConcurrentSkipListMap`, `CHM` means `ConcurrentHashMap`, and `CHMV8` means `ConcurrentHashMapV8`.

Maps. Figure 2 presents a different picture for the `Map` implementations. For the `LinkedHashMap`, `Hashtable`, and `Collections.synchronizedMap()` implementations, energy follows the same curve as time, for both traversal and insertion operations, on both `System#1` and `System#2`. Surprisingly, however, the same cannot be said for the removal operations. Removal operations on `Hashtable` and `Collections.synchronizedMap()` exhibited energy consumption that is proportionally smaler than their execution time for both systems. Such behavior is due to a drop on power consumption. Since such collections are single-lock based, for each removal operation, the other threads need to wait until the underling structure is rebuilt. This synchronization prevents the collection to speed-up, and also decreases power usage.

On the other hand, for the `ConcurrentSkipListMap`, `ConcurrentHashMap` and `ConcurrentHashMapV8` implementations, more power is being consumed behind the scenes. As we mentioned that energy consumption is the product of power consumption and time, if the benchmark receives a 1.5x speed-up but, at the same time, yields a threefold increase in power consumption, energy consumption will increase twofold. This scenario is roughly what happens in traversal operations, when transitioning from `Hashtable` to `ConcurrentHashMap`. Even though `ConcurrentHashMap` produces a speedup of 1.46x over the `Hashtable` implementation on `System#1`, it achieves that by consuming 1.51x more power. As a result, `ConcurrentHashMap` consumed slightly more energy than `Hashtable` (2.38%). On `System#2`, energy consumption for `Hastable` and `ConcurrentHashMap` are roughly the same. This result is relevant mainly because several textbooks [26], research papers [27] and internet blog posts [28] suggest `ConcurrentHashMap` as the *de facto* replacement for the old associative `Hashtable` implementation. Our result suggests that the decision on whether or not to use `ConcurrentHashMap` should be made with care, in particular, in scenarios where the energy consumption is more important than performance. However, the newest `ConcurrentHashMapV8` implementation, released in the version 1.8 of the Java programming language, handles large maps or maps that have many keys with colliding hash codes more gracefully. On `System#1`, `ConcurrentHashMapV8` provides performance savings of 2.19x when compared to `ConcurrentHashMap`, and energy savings of 1.99x in traversal operations (these savings are, respectively, 1.57x and 1.61x in insertion operations, and 2.19x and 2.38x in removal operations). In addition, for insertions and removals operations on both systems, `ConcurrentHashMapV8` has performance similar or even better than the not thread-safe implementation.

`ConcurrentHashMapV8` is a completely rewritten version of its predecessor. The primary design goal of this implementation is to maintain concurrent readability (typically on the `get()` method, but also on `Iterators`) while minimizing update contention. This map acts as a binned hash table. Internally, it uses tree-map-like structures to

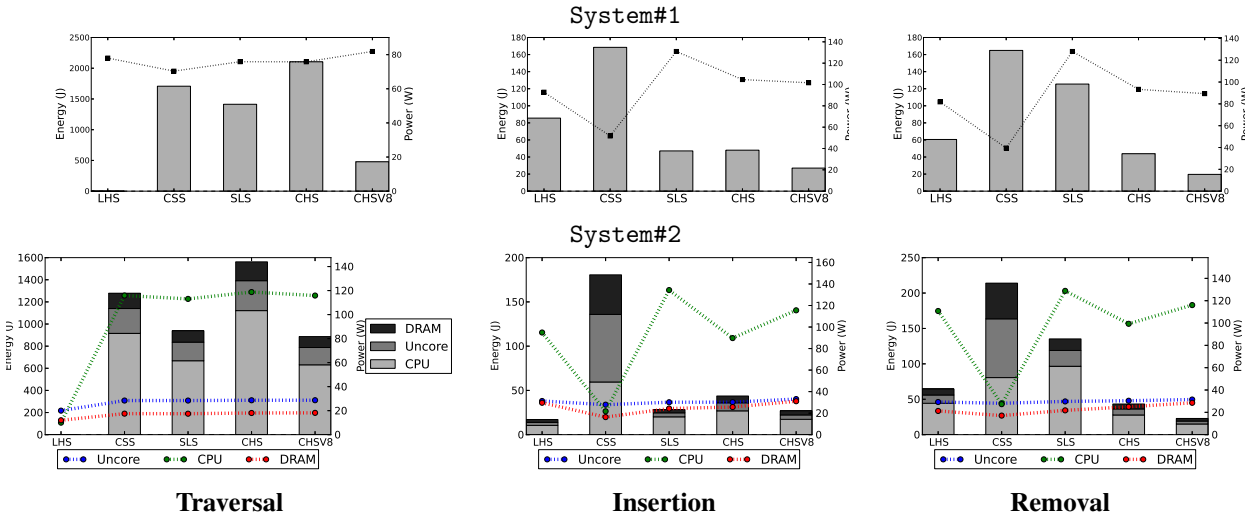

Figure 3. Energy and power results for traversal, insertion and removal operations for different `Set` implementations. Bars mean energy consumption and lines mean power consumption. `LSH` means `LinkedHashSet`, `CSS` means `Collections.synchronizedSet()`, `SLS` means `ConcurrentSkipListSet`, `CHS` means `ConcurrentHashSet`, and `CHSV8` means `ConcurrentHashSetV8`.

maintain bins containing more nodes than would be expected under ideal random key distributions over ideal numbers of bins. This tree also requires an additional locking mechanism. While list traversal is always possible by readers even during updates, tree traversal is not, mainly because of tree-rotations that may change the root node and its links. Insertion of the first node in an empty bin is performed with a Compare-And-Set operation. Other update operations (insertional, removal, and replace) require locks. Locking support for these locks relies on builtin "synchronized" monitors.

Sets. Figure 3 shows the results of our experiments with `Set`. We did not present the results for `CopyOnWriteHashSet` in this figure because it exhibited a much higher energy consumption, which made the figure difficult to read. First, for all of the implementations of `Set`, we can observe that energy consumption follows the same behavior of power on traversal operations for both `System#1` and `System#2`. However, for insertion and removal operations, they are not always proportional. Notwithstanding, an interesting trade-off can be observed when performing traversal operations. As expected, the non-thread-safe implementation, `LinkedHashSet`, achieved the least energy consumption and execution time results, followed by the `CopyOnWriteArraySet` implementation. We believe that the same recommendation for `CopyOnWriteArrayList` fits here: this collection should only be used in scenarios where reads are much more frequent than insertions. For all other implementations, the `ConcurrentHashSetV8` presents the best results among the thread-safe ones. Interestingly, for traversals, `ConcurrentHashSet` presented the worst results, consuming 1.23x more energy and 1.14x more time than `Collections.synchronizedSet()` on `System#1` (1.31x more energy and 1.19x more time on `System#2`).

Another interesting result is observed with `ConcurrentSkipListSet`, which consumes only 1.31x less energy than a `Collections.synchronizedList()` on removal operations on `System#1`, although it saves 4.25x in execution time. Such energy consumption overhead is also observed on `System#2`. Internally, `ConcurrentSkipListSet` relies on a `ConcurrentSkipListMap`, which is non-blocking, linearizable, and based on the compare-and-swap (CAS) operation. During traversal, this collection marks the "next" pointer to keep track of triples (predecessor, node, successor) in order to detect when and how to unlink deleted nodes. Also, because of the asynchronous nature of these maps, determining the current number of elements (used in the `Iterator`) requires a traversal of all elements. These behaviors are susceptible to create the energy consumption overhead observed in Figure 3.

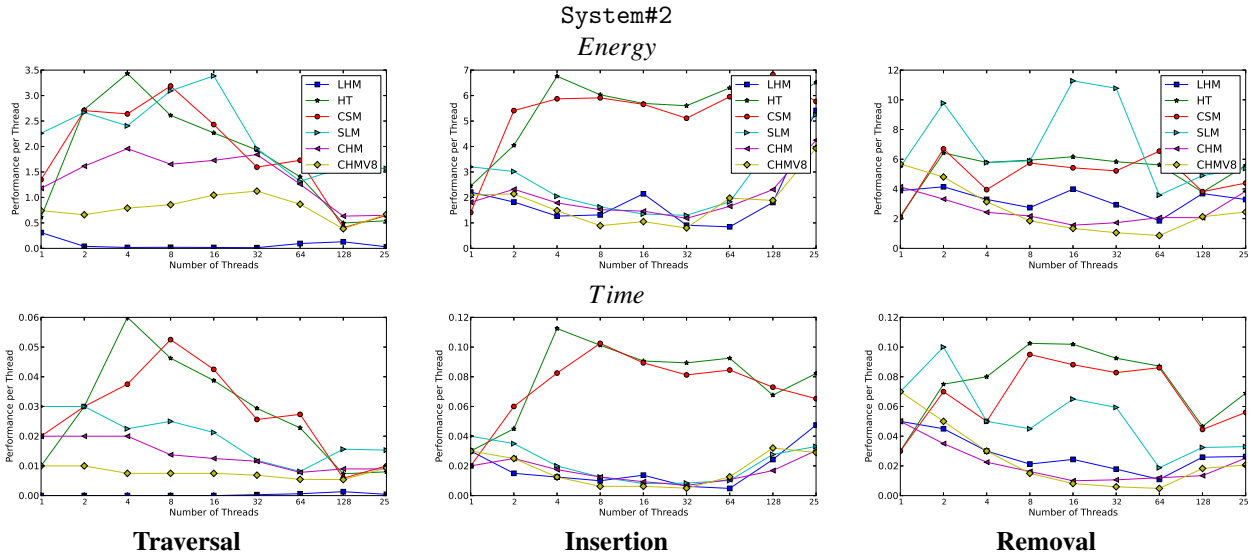

Figure 4. Energy consumption and execution time in the presence of concurrent threads (X axis: the number of threads, Y axis: energy consumption normalized against the number of element accesses, in joules per 100,000 elements)

RQ1 Summary: We observed that different implementations of the same collection can greatly impact both energy consumption and execution time. When comparing `CopyOnWriteArrayList` with the non-thread-safe implementation, the difference can be higher than 152x.

RQ2 Summary: We observed that different operations of the same collection can greatly impact both energy consumption and execution time. For instance, removing an element from a `Vector` can consume 200x more energy than inserting an element.

3.2. Energy Behaviors with Different Number of Threads

In this group of experiments, we aim to answer **RQ3**. For this experiment, we chose `Map` implementations only, due to presence of both single-lock and high-performatic implementations. We vary the number of threads (1, 2, 4, 8, 16, 32, 64, 128, and 256 concurrent threads) and study how such variations impact energy consumption. An increment in the number of threads also increments the total number of elements inside the collection. Since each thread inserts 100,000 elements, when performing with one thread, the total number of elements is also 100,000. When performing with 2 threads, the final number of elements is 200,000, and so on. To give an impression on how `Map` implementations scale in the presence of multiple threads, Figure 4 demonstrates the effect of different thread accesses on benchmarks.

In this figure, each data point is normalized by the number of threads, so it represents the energy consumption per thread, per configuration. Generally speaking, `Hashtable` and `Collections.synchronizedMap()` scale up well. For instance, we observed a great increment of energy consumption when using `Collections.synchronizedMap()` when we move from 32 to 64 threads performing traversals, but this trend can also observed for insertions and removals. Still on traversals, all `Map` implementations greatly increase the energy consumed as we add more threads. Also, all thread-safe implementations have their own "15 minutes of fame". Despite the highly complex landscape, some patterns do seem to recur. For instance, even though `ConcurrentHashMapV8` provides the best scalability among the thread-safe collection implementations, it still consumes about 11.6x more energy than the non-thread-safe implementation. However, the most interesting fact is the peak of `ConcurrentSkipListMap`, when performing with 128 and 256 threads. As discussed earlier, during traversal, `ConcurrentSkipListMap` marks or unlinks a node with `null` value from its predecessor (the map uses the nullness of value fields to indicate deletion). Such mark is a compare-and-set operation, and happens every time it finds a `null` node. When this operation fails, it forces a re-traversal from caller.

For insertions, we observed a great disparity; while `Hashtable` and `Collections.synchronizedMap()` scale up well, `ConcurrentSkipListMap`, `ConcurrentHashMap` and `ConcurrentHashMapV8` scale up very well. One particular characteristic about `ConcurrentHashMapV8` is that the insertion of the first element in an empty map employs compare-and-set operations. Other update operations (insert, delete, and replace) require locks. Locking support for these locks relies on builtin "synchronized" monitors. When performing using from 1 to 32 threads, they have energy and performance behaviors similar to the non-thread-safe implementation. Such behavior was previously discussed in Figure 2.

For removals, interestingly, both `ConcurrentHashMap` and `ConcurrentHashMapV8` scale better than all other implementations, even the non-thread-safe implementation, `LinkedHashMap`. `ConcurrentSkipListMap`, on the other hand, presents the worst scenario, in particular with 16, 32 and 128 threads, even when compared to the single-lock implementations, such as `Hashtable` and `Collections.synchronizedMap()`.

RQ3 Summary: We observed that, regardless of the operation used, `ConcurrentHashMapV8` presents the best scalability. For traversals, each thread-safe implementations have its own "15 minutes of fame". However, insertion presents a great disparity; `Hashmap` and `Collections.synchronizedMap()` scale up well, whereas `ConcurrentSkipListMap`, `ConcurrentHashMap` and `ConcurrentHashMapV8` provide scalability comparable to the non-thread-safe implementation.

3.3. Collection configurations and usages

We now focus on **RQ4**, studying the impact of different collection configurations and usage patterns on program energy behaviors. The `Map` implementations have two important "tuning knobs": the *initial capacity* and *load factor*. The capacity is the total number of elements inside a `Map` and the initial capacity is the capacity at the time the `Map` is created. The default initial capacity of the `Map` implementations is only 16 locations. We report a set of experiments where we configured the initial capacity to be 32, 320, 3,200, 32,000, 320,000, and 3,200,000 elements — the last one is the total number of elements that we insert in a collection. Figure 5 shows how energy consumption behaves using these different initial capacity configurations.

As we can observe from this figure, the results can vary greatly when using different initial capacities, in terms of both energy consumption and execution time. The most evident cases are when performing with a high initial capacity in `Hashtable` and `ConcurrentHashMap`. `ConcurrentHashMapV8`, on the other hand, presents the least variation on energy consumption.

The other tuning knob is the load factor. It is a measure of how full the hash table is allowed to get before its capacity is automatically increased. When the number of elements inside a `Map` exceeds the product of the load factor and the current capacity, the hash table is rehashed; that is, its internal structure is rebuilt. The default load factor value in most `Map` implementation is 0.75. It means that, using initial capacity as 16, and the load factor as 0.75, the product of capacity is 12 (16 * 0.75 = 12). Thus, after inserting the 12th key, the new map capacity after rehashing will be 32. If the initial capacity is greater than the maximum number of entries divided by the load factor, no rehash operations will ever occur. Figure 6 shows how energy consumption behaves using different load factors configurations. We perform these experiments only with insertion operations.[5].

From this figure we can observe that, albeit small, the load factor also influences both energy consumption and time. For instance, when using a load factor of 0.25, we observed the most energy inefficient results on `System#1`, except in one case (the energy consumption of `LinkedHashMap`). On `System#2`, the 0.25 configuration was the worst in three out of 5 of the benchmarks. We believe they is due to the successive rehashing operations that must occur. Generally speaking, the default load factor (.75) offers a good tradeoff between performance, energy, and space costs. Higher values decrease the space overhead but increase the time cost to look up an entry, which can reflect in most of the `Map` operations, including `get()` and `put()`). It is possible to observe this cost when using a load factor of 1.0, which means that the map will be only rehashed when the number of current elements reaches the current maximum size. The maximum variation was found when performing operations on a `Hastable` on `System#1`, in the default load factor, achieving 1.17x better energy consumption over the 0.25 configuration, and 1.09x in execution time.

[5]We did not performed experiments with `ConcurrentSkipListMap` because it does not provide access to initial capacity and load factor variables.

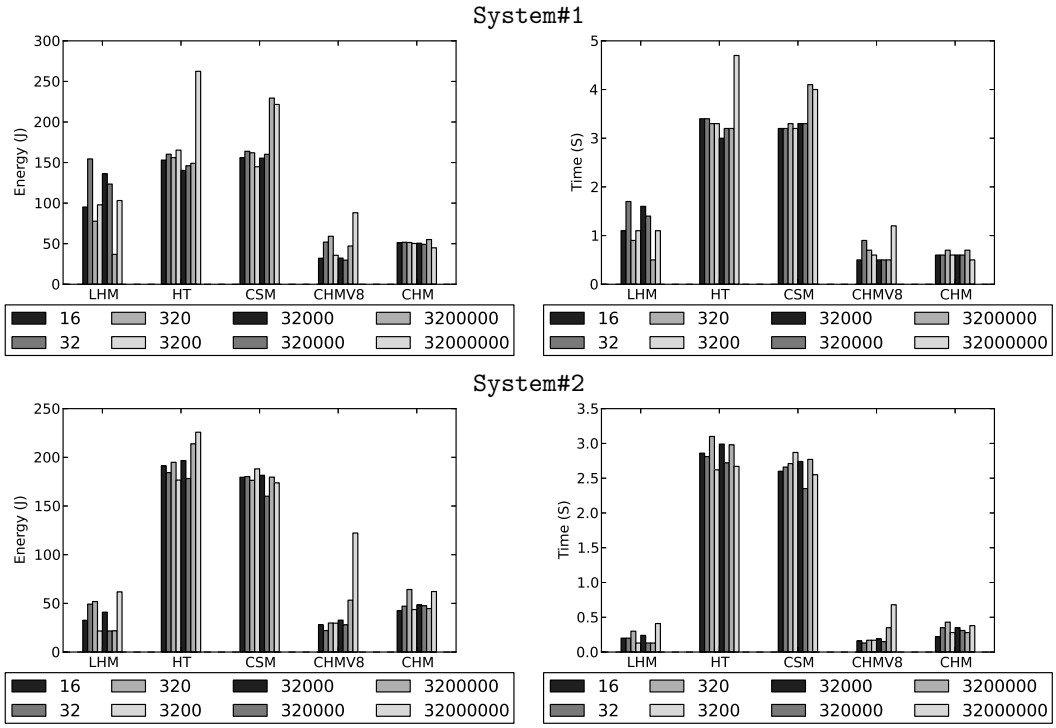

Figure 5. Energy consumption and performance variations with different initial capacities.

RQ4 Summary: Different maps configurations can play an important role on energy behavior. For initial capacity, the results varied considerably among the various analyzed configurations and there is no clear winner. Moverover, we observed that a small load factor can increase energy consumption up to 15%. In our experiments, the default configuration provided the least energy consumption.

3.4. The Devil is in the Details

In this section we further analyze some implementation details that can greatly energy consumption.

Upper bound limit. We also observed that, on traversal and insertion operations, when the upper bound limit needs to be computed in each iteration, for instance, when using

```
for(int i=0; i < list.size(); i++) {
// do stuf...
}
```

the `Vector` implementation consumed about twice as much as it consumed when this limit is computed only once on (1.98x more energy and 1.96x more time), for instance, when using

```
int size = list.size();
for(int i=0; i < size; i++) {
// do stuf...
}
```

When this limit is computed beforehand, energy consumption and time drop by half. Such behavior is observed on both `System#1` and `System#2`. We believe it happens mainly because for each loop iteration, the current thread needs

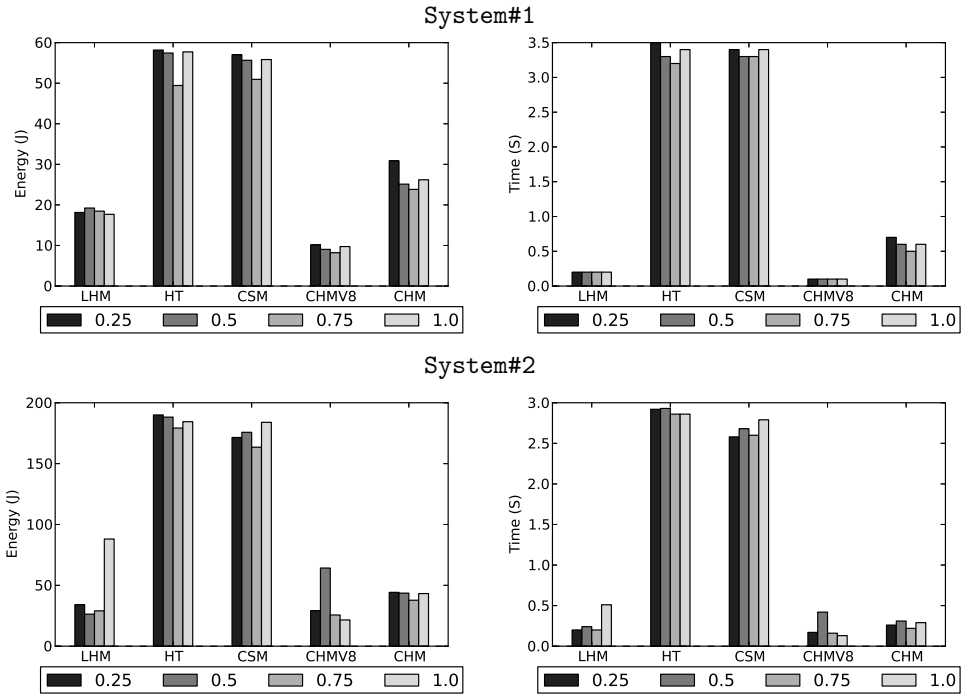

Figure 6. Energy consumption and performance variations with different load factors.

to fetch the `list.size()` variable from memory, which would incur in some cache misses. When initializing a `size` variable close to the loop statement, we believe that such variable will be stored in a near memory location, and thus, can be fetched all together with the remaining data. Using this finding, well-known IDEs, such as Eclipse and IntelliJ, can take advantage of it and implement refactoring suggestions for developers. Currently, the Eclipse IDE does not provide such feature. Also, recent studies have shown that programmers are more likely to follow IDE tips [29]. One concern, however, is related to removal operations. Since removal on `Lists` shift any subsequent elements to the left, if the limit is computed beforehand, the `i++` operation will skip one element.

Enhanced for loop. We also analyzed traversal operations when the programmer iterates using an *enhanced for loop*, for instance, when using

```
for (String e: list) { ... }
```

which is translated to an `Iterator` at compile time. Figure 7 shows the results. In this configuration, `Vector` need to synchronize in two different moments: during the creation of the `Iterator` object, and in every call of the `next()` method. By contrast, the `Collections.synchronizedList()` does not synchronize on the `Iterator`, and thus has similar performance and energy usage when compared to our baseline, `ArrayList`. On `System#1`, energy decreased from 37.07J to 2.65J, whereas time decreased from 0.81 to 0.10. According to the `Collections.synchronizedList()` documentation, the programmer must ensure external synchronization when using `Iterator`.

Removal on objects. When using `Lists`, instead of perform removals based on the indexes, one can perform removals based on object instances, for instance, when using

```
for (int i = 0; i < threads; i++) {
  for (int j = 0; j < list.size(); j++) {
```

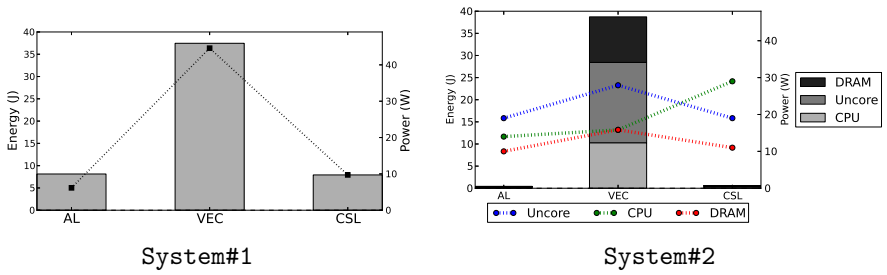

Figure 7. traversal operations using the `get()` method. We use the same abbreviations of Figure 1.

```
    boolean b = list.remove(i + "-" + j);
  }
}
```

When using this operation, we observed an increment on energy consumption of 39.21% on `System#1` (32.28% on execution time). This additional overhead is due to the traversal needed for this operations. Since the collection does not know in which position the given object is placed, it needs to traversal and compare each element until it finds the object – or until the collection ends.

4. Related Work

The energy impacts of different design decisions made by software engineers have been previously investigated in several empirical studies. These studies analyzed a number of factors, varying from sorting algorithms [30], constructs for managing concurrent execution [13], design patterns [31], refactoring [11], cloud offloading [10, 32, 33], VM services [34], test suite minimization [35], code obfuscation [36], among many others. Zhang *et al.* [33] presented a mechanism for automatically refactoring an Android app into one implementing the on-demand computation offloading design pattern, which can transfer some computation-intensive tasks from a smartphone to a server so that the task execution time and battery power consumption of the app can be reduced significantly. Cao *et al.* [34] described how different VM services (such as the Just-In-Time compiler, interpretation and/or the garbage collector) impact energy consumption. Li *et al.* [12] presented an evaluation of a set of programming practices suggested in the official Android developers web site. They observed that some practices such as the network packet size can provide impressive energy savings, while others, such as limiting memory usage, had minimal impact on energy usage. Vallina-Rodriguez*et al.* [37] surveys the energy-efficient software-centric solutions on mobile devices, ranging from operating system solutions to energy savings via process migration to the cloud and protocol optimizations.

The performance of collections is also an active area of research, with great improvements in lock-free collections [38], spatial collections [39], dynamic-sized collections [40], among many others. Java collections are the focus of several studies [16, 41, 42]. Our work and related work cited here are complementary. Together, they attempt to understand the energy/performance trade-offs of different collections.

To the best of our knowledge, two prior studies intersect with our goal of understanding the energy consumption of collection [15, 43], but neither with the same focus or scope. Hunt *et al.* [15] studied the energy consumption of three collections: a simple FIFO, a double-ended queue, and a sorted linked list. SEEDS [43] is a general decision-making framework for optimizing software energy consumption. As a case study to validate the effectiveness of their framework, the authors demonstrated how SEEDS can identify energy-inefficient uses of Java collections, and help automate the process of selecting more efficient ones. The focus of their work is the methodology itself, not the energy footprint of Java collections. As a result, they do not provide a fine-grained analysis (*e.g.* different use scenarios of collections under different configurations), nor were their experiments constructed in a multi-core environment.

5. Threat to Validity

We divide our discussion on threats to validity into internal factors and external factors.

Internal factors: First, the elements which we used are not randomly generated. We chose to not use random number generators because they can greatly impact the performance and energy consumption of our benchmarks. We observed standard deviation of over 70% between two executions when using the random number generators. We mitigate this problem by combining the index of the for loop plus the thread id that inserted the element. This approach also prevents compiler optimizations that may happen when using only the index of the for loop as the element to be inserted in the collection.

External factors: First, our results are limited by our selection of benchmarks. Nonetheless, our corpus spans a wide spectrum of collections, ranging from lists, sets, and maps. Second, there are other possible collections implementations beyond the scope of this paper. With our methodology, we expect similar analysis can be conducted by others. Third, our results are reported with the assumption that JIT is enabled. This stems from our observation that later runs of JIT-enabled executions do stabilize in terms of energy consumption and performance [13]. We experienced differences in standard deviation of over 30% when comparing the warmup run (first 3 executions) and later runs, but less than 5% when comparing the last 3 runs.

6. Conclusions

In this paper, we presented an empirical study that investigates the impacts of using different collections on energy usage. As subjects for the study, we analyzed the main methods of 16 types of commonly used collection in the Java language. Some of the findings of this study include: (1) Different operations of the same implementation also have different energy footprints. For example, a removal operation in a `ConcurrentSkipListMap` can consume more than 4 times of energy than an insertion to the same collection. Also, for `CopyOnWriteArraySet`, an insertion consumes three order of magnitude more than a read. (2) Execution time is not always a reliable indicator for energy consumption; this is particularly true for various `Map` implementations. In other words, the consumption of power — the rate of energy consumption — is not a constant across different collection implementations.

Based on these conclusions, there are several potential areas for future work. First, we plan on enlarging the scope of our study. Although we considered a significant number of subjects, adding additional collection, and their methods, would potentially allow us to refute or confirm some of our observations in addition to perform the removal experiments for all collections available. Second, we believe that other tuning knobs should be studied, such as varying the number of concurrent threads accessing the collection, and varying the data size being manipulated in the collection. With insights of this study, we plan to introduce the concept of relaxed collection. One step towards this goal is to reduce their accuracy [44]. Since Java8 introduced the concept of `Streams`, which use implicitly parallelism and is well-suitable for data-parallel programs, an approximate solution for a given function, for instance `sum` the values of all elements, over a huge collection can take a fraction of memory, time and, last but not least, energy consumption.

References

[1] C. Wilke, S. Richly, S. Gotz, C. Piechnick, U. Assmann, Energy consumption and efficiency in mobile applications: A user feedback study, in: Green Computing and Communications (GreenCom), 2013 IEEE and Internet of Things (iThings/CPSCom), IEEE International Conference on and IEEE Cyber, Physical and Social Computing, 2013, pp. 134–141.

[2] D. Li, S. Hao, W. G. J. Halfond, R. Govindan, Calculating source line level energy information for android applications, in: Proceedings of the 2013 International Symposium on Software Testing and Analysis, ISSTA 2013, 2013, pp. 78–89.

[3] C. Seo, S. Malek, N. Medvidovic, Component-level energy consumption estimation for distributed java-based software systems, in: M. Chaudron, C. Szyperski, R. Reussner (Eds.), Component-Based Software Engineering, Vol. 5282 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2008, pp. 97–113.

[4] K. Liu, G. Pinto, D. Liu, Data-oriented characterization of application-level energy optimization, in: Proceedings of the 18th International Conference on Fundamental Approaches to Software Engineering, FASE'15, 2015.

[5] A. Chandrakasan, S. Sheng, R. Brodersen, Low-power cmos digital design, Solid-State Circuits, IEEE Journal of 27 (4) (1992) 473–484.

[6] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanaa, C. Le, RAPL: memory power estimation and capping, in: Proceedings of the 2010 International Symposium on Low Power Electronics and Design, 2010, Austin, Texas, USA, August 18-20, 2010, 2010, pp. 189–194.

[7] H. Ribic, Y. D. Liu, Energy-efficient work-stealing language runtimes, in: Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, 2014, pp. 513–528.

[8] T. W. Bartenstein, Y. D. Liu, Rate types for stream programs, in: Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '14, 2014, pp. 213–232.

[9] M. Cohen, H. S. Zhu, E. E. Senem, Y. D. Liu, Energy types, in: OOPSLA'12, 2012, pp. 831–850.

[10] Y.-W. Kwon, E. Tilevich, Reducing the energy consumption of mobile applications behind the scenes, in: ICSM, 2013, pp. 170–179.
[11] C. Sahin, L. Pollock, J. Clause, How do code refactorings affect energy usage?, in: Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '14, 2014, pp. 36:1–36:10.
[12] D. Li, W. Halfond, An investigation into energy-saving programming practices for android smartphone app development, in: GREENS, 2014.
[13] G. Pinto, F. Castor, Y. D. Liu, Understanding energy behaviors of thread management constructs, in: Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '14, 2014, pp. 345–360.
[14] G. Pinto, F. Castor, Y. D. Liu, Mining questions about software energy consumption, in: Proceedings of the 11th Working Conference on Mining Software Repositories, MSR 2014, 2014, pp. 22–31.
[15] N. Hunt, P. S. Sandhu, L. Ceze, Characterizing the performance and energy efficiency of lock-free data structures, in: Proceedings of the 2011 15th Workshop on Interaction Between Compilers and Computer Architectures, 2011.
[16] G. Xu, Coco: Sound and adaptive replacement of java collections, in: Proceedings of the 27th European Conference on Object-Oriented Programming, ECOOP'13, 2013, pp. 1–26.
[17] Y. Bu, V. Borkar, G. Xu, M. J. Carey, A bloat-aware design for big data applications, in: Proceedings of the 2013 International Symposium on Memory Management, ISMM '13, 2013, pp. 119–130.
[18] J. Li, J. F. Martínez, Power-performance considerations of parallel computing on chip multiprocessors, ACM Trans. Archit. Code Optim. 2 (4) (2005) 397–422.
[19] M. Kambadur, M. A. Kim, An experimental survey of energy management across the stack, in: Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014, 2014, pp. 329–344.
[20] B. Subramaniam, W.-c. Feng, Towards energy-proportional computing for enterprise-class server workloads, in: Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering, ICPE '13, 2013, pp. 15–26.
[21] M. Hähnel, B. Döbel, M. Völp, H. Härtig, Measuring energy consumption for short code paths using rapl, SIGMETRICS Perform. Eval. Rev. 40 (3) (2012) 13–17.
[22] L. Benini, A. Bogliolo, G. De Micheli, A survey of design techniques for system-level dynamic power management, IEEE Trans. Very Large Scale Integr. Syst. 8 (3) (2000) 299–316.
[23] M. S. Papamarcos, J. H. Patel, A low-overhead coherence solution for multiprocessors with private cache memories, in: Proceedings of the 11th Annual International Symposium on Computer Architecture, ISCA '84, 1984, pp. 348–354.
[24] M. De Wael, S. Marr, T. Van Cutsem, Fork/join parallelism in the wild: Documenting patterns and anti-patterns in java programs using the fork/join framework, in: Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, PPPJ '14, 2014, pp. 39–50.
[25] A. E. Trefethen, J. Thiyagalingam, Energy-aware software: Challenges, opportunities and strategies, Journal of Computational Science 4 (6) (2013) 444 – 449, scalable Algorithms for Large-Scale Systems Workshop (ScalA2011), Supercomputing 2011.
[26] T. Peierls, B. Goetz, J. Bloch, J. Bowbeer, D. Lea, D. Holmes, Java Concurrency in Practice, Addison-Wesley Professional, 2005.
[27] D. Dig, J. Marrero, M. D. Ernst, Refactoring sequential java code for concurrency via concurrent libraries, in: Proceedings of the 31st International Conference on Software Engineering, ICSE '09, 2009, pp. 397–407.
[28] B. Goetz, Java theory and practice: Concurrent collections classes, http://www.ibm.com/developerworks/java/library/j-jtp07233/index.html, accessed: 2014-09-29.
[29] E. Murphy-Hill, R. Jiresal, G. C. Murphy, Improving software developers' fluency by recommending development environment commands, in: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE '12, 2012, pp. 42:1–42:11.
[30] C. Bunse, H. Hpfner, S. Roychoudhury, E. Mansour, Energy efficient data sorting using standard sorting algorithms, in: J. Cordeiro, A. Ranchordas, B. Shishkov (Eds.), Software and Data Technologies, Vol. 50 of Communications in Computer and Information Science, Springer Berlin Heidelberg, 2011, pp. 247–260.
[31] C. Sahin, F. Cayci, I. Gutierrez, J. Clause, F. Kiamilev, L. Pollock, K. Winbladh, Initial explorations on design pattern energy usage, in: GREENS, 2012, pp. 55–61.
[32] Y.-W. Kwon, E. Tilevich, Cloud refactoring: automated transitioning to cloud-based services, Autom. Softw. Eng. 21 (3) (2014) 345–372.
[33] Y. Zhang, G. Huang, X. Liu, W. Zhang, H. Mei, S. Yang, Refactoring android java code for on-demand computation offloading, in: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '12, 2012, pp. 233–248.
[34] T. Cao, S. M. Blackburn, T. Gao, K. S. McKinley, The yin and yang of power and performance for asymmetric hardware and managed software, in: Proceedings of the 39th Annual International Symposium on Computer Architecture, ISCA '12, 2012, pp. 225–236.
[35] D. Li, Y. Jin, C. Sahin, J. Clause, W. G. J. Halfond, Integrated energy-directed test suite optimization, in: Proceedings of the 2014 International Symposium on Software Testing and Analysis, ISSTA 2014, 2014, pp. 339–350.
[36] C. Sahin, P. Tornquist, R. McKenna, Z. Pearson, J. Clause., How does code obfuscations impact energy usage?, in: ICSME, 2014.
[37] N. Vallina-Rodriguez, J. Crowcroft, Energy management techniques in modern mobile handsets, Communications Surveys Tutorials, IEEE 15 (1) (2013) 179–198.
[38] M. Fomitchev, E. Ruppert, Lock-free linked lists and skip lists, in: Proceedings of the Twenty-third Annual ACM Symposium on Principles of Distributed Computing, PODC '04, 2004, pp. 50–59.
[39] B.-U. Pagel, H.-W. Six, H. Toben, P. Widmayer, Towards an analysis of range query performance in spatial data structures, in: Proceedings of the Twelfth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS '93, 1993, pp. 214–221.
[40] M. Herlihy, V. Luchangco, M. Moir, W. N. Scherer, III, Software transactional memory for dynamic-sized data structures, in: PODC, 2003.
[41] W. Torres, G. Pinto, B. Fernandes, J. a. P. Oliveira, F. A. Ximenes, F. Castor, Are java programmers transitioning to multicore?: A large scale study of java floss, in: Proceedings of the Compilation of the Co-located Workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, & VMIL'11, SPLASH '11 Workshops, 2011, pp. 123–128.
[42] Y. Lin, D. Dig, Check-then-act misuse of java concurrent collections, in: Proceedings of the 2013 IEEE Sixth International Conference on

Software Testing, Verification and Validation, ICST '13, 2013, pp. 164–173.
[43] I. Manotas, L. Pollock, J. Clause, Seeds: A software engineer's energy-optimization decision support framework, in: ICSE, 2014.
[44] M. Carbin, D. Kim, S. Misailovic, M. C. Rinard, Proving acceptability properties of relaxed nondeterministic approximate programs, in: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, 2012, pp. 169–180.