Bottle Graphs: Visualizing Scalability Bottlenecks in Multi-Threaded Applications

Kristof Du Bois   Jennifer B. Sartor   Stijn Eyerman   Lieven Eeckhout
Ghent University, Belgium
{kristof.dubois,jennifer.sartor,stijn.eyerman,leeckhou}@elis.UGent.be

Abstract

Understanding and analyzing multi-threaded program performance and scalability is far from trivial, which severely complicates parallel software development and optimization. In this paper, we present bottle graphs, a powerful analysis tool that visualizes multi-threaded program performance, in regards to both per-thread parallelism and execution time. Each thread is represented as a box, with its height equal to the share of that thread in the total program execution time, its width equal to its parallelism, and its area equal to its total running time. The boxes of all threads are stacked upon each other, leading to a stack with height equal to the total program execution time. Bottle graphs show exactly how scalable each thread is, and thus guide optimization towards those threads that have a smaller parallel component (narrower), and a larger share of the total execution time (taller), i.e. to the 'neck' of the bottle.

Using light-weight OS modules, we calculate bottle graphs for unmodified multi-threaded programs running on real processors with an average overhead of 0.68%. To demonstrate their utility, we do an extensive analysis of 12 Java benchmarks running on top of the Jikes JVM, which introduces many JVM service threads. We not only reveal and explain scalability limitations of several well-known Java benchmarks; we also analyze the reasons why the garbage collector itself does not scale, and in fact performs optimally with two collector threads for all benchmarks, regardless of the number of application threads. Finally, we compare the scalability of Jikes versus the OpenJDK JVM. We demonstrate how useful and intuitive bottle graphs are as a tool to analyze scalability and help optimize multi-threaded applications.

OOPSLA '13, October 29–31, 2013, Indianapolis, Indiana, USA.
Copyright © 2013 ACM 978-1-4503-2374-1/13/10…$15.00.
http://dx.doi.org/10.1145/2509136.2509529

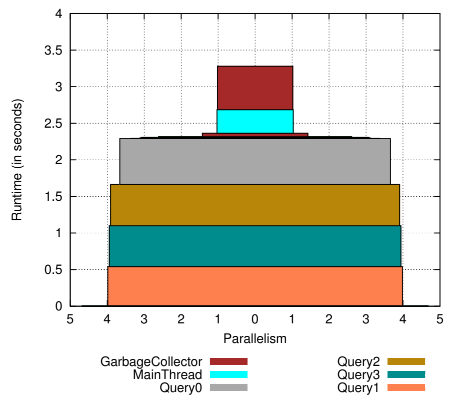


Figure 1. Example of a bottle graph: The lusearch DaCapo benchmark with 4 application threads (QueryX), a main thread that performs initialization, and garbage collection threads running on Jikes JVM.

Categories and Subject Descriptors D.2.8 [*Software Engineering*]: Metrics – Performance Measures

Keywords Performance Analysis, Multicore, Java, Bottlenecks

1. Introduction

Analyzing the performance of multi-threaded applications on today's multicore hardware is challenging. While processors have advanced in terms of providing performance counters and other tools to help analyze performance, the reasons scalability is limited are hard to tease apart. In particular, the interaction of threads in multi-threaded applications is complex: some threads perform sequentially for a period of time, others are stalled with no work to do, and synchronization behavior makes some threads wait on locks or barriers. Many papers have demonstrated the inability of multi-threaded programs to scale well, but studying the root causes of scalability bottlenecks is challenging. Previous research has not analyzed scalability on a per-thread basis, or not suggested specific threads for performance optimization that give the greatest potential for improvement (or those that are most imbalanced).

We propose bottle graphs as an intuitive performance analysis tool that visualizes scalability bottlenecks in multi-threaded applications. Bottle graphs are stacked bar graphs, where the height along the y-axis is the total application execution time, see Figure 1 for an example. The stacked bar represents each thread as a box: the height is the thread's share of the total program execution time; the width is the number of parallel threads that this thread runs concurrently with, including itself; and the box area is the thread's total running time. The center of the x-axis is zero, and thus parallelism is symmetric, reported on both the left and right sides of the zero-axis. We stack threads' boxes, sorting threads by their parallelism, with widest boxes (threads with higher parallelism) shown at the bottom and narrower boxes at the top of the total application bar graph — yielding a bottle-shaped graph, hence the name *bottle graph*.

Bottle graphs provide a new way to analyze multi-threaded application performance along the axes of execution time and parallelism. Bottle graphs visualize how scalable real applications are, and which threads have a larger total running time (total box area), which threads have limited parallelism (narrow boxes), and which threads contribute significantly to execution time (tall boxes). Threads that represent scalability bottlenecks show up as narrow and tall boxes around the 'neck' of the bottle graph. Bottle graphs thus quickly point software writers and optimizers to the threads with the greatest optimization potential.

We measure bottle graphs of real applications running on real hardware through operating system support using light-weight Linux kernel modules. The OS naturally knows what threads are running at a given time, and when threads are created, destroyed, or scheduled in and out. When a thread is scheduled in or out, a kernel module is triggered to update per-thread counters keeping track of both the total thread running time, and number of concurrently running threads. Using kernel modules, our bottle measurements incur very little overhead (0.68% on average), require no recompilation of the kernel, and require no modifications to applications or hardware.

To demonstrate the power of bottle graphs as an analysis and optimization tool, we perform an experimental study of 12 single- and multi-threaded benchmarks written in Java, both from the DaCapo suite and pseudoJBB from SPEC. The benchmarks run on top of the Jikes Research Virtual Machine on real hardware. Because the applications run on top of a runtime execution environment, even single-threaded benchmarks have extra Java virtual machine (JVM) service threads, and thus we can analyze not only the application, but also JVM performance and scalability.

The work most related to ours is IBM's WAIT tool [2], which also analyzes the performance and scalability of multi-threaded Java programs. However, bottle graphs reveal much more information at a much lower overhead. Because WAIT samples threads' state, it gathers information only at particular points in time, while our tool aggregates our metrics at every thread status (active or inactive) change without loss of information. In order to approach bottle graph's amount of information, WAIT must sample more frequently, thus incurring an order of magnitude more overhead. Bottle graphs visualize performance per thread, allowing for easy grouping of thread categories (i.e., for thread pools or application versus garbage collection threads). Furthermore, WAIT can only analyze Java application threads, while our OS modules facilitate bottle graph creation for any multi-threaded program, including analysis of underlying Java virtual machine threads.

In our comprehensive study of Java applications, we vary the number of application threads, number of garbage collection threads, and heap size. We analyze performance differences between benchmark iterations, and study the scalability of JVM service threads in conjunction with application threads. We find sometimes surprising insights: a) counter to the common intuition that one should set the number of collection threads equal to the number of cores or the number of application threads, we find that Jikes performs optimally with two collector threads, both with single and multi-threaded benchmarks, regardless of the number of application threads or heap size; b) when increasing the number of application threads, *or* increasing the number of collection threads, the amount of garbage collection work also increases; c) although application time decreases when increasing the number of application threads, the amount of work the application performs also increases, thus showing that these applications are limited in their scalability.

Furthermore, we analyze why there is multi-threaded imbalance in one well-known benchmark, pmd, as the bottle graph reveals that one application thread is tall and narrow, while others are much better parallelized. Analysis reveals that this is due to input file imbalance, and we suggest opportunities to improve both performance and scalability. Finally, because we find that Jikes' garbage collector has limited scalability, we compare Jikes' behavior with that of OpenJDK. We reveal that increasing the number of garbage collector threads in Jikes leads to significantly more synchronization activity than for OpenJDK's garbage collector. OpenJDK's collector does scale to larger thread counts than Jikes, offering good performance for up to 8 collection threads. However, collection work still increases as we increase either application thread or collector thread count.

In summary, bottle graphs are a powerful and intuitive way to visualize parallel performance. They facilitate fast analysis of scalability bottlenecks, and help target optimization to the threads most limited in parallelism, or those slowing down the total execution the most. Such tools are critical as we move forward in the multicore era to efficiently use hardware resources while running multi-threaded applications.

2. Bottle Graphs

This section introduces our novel contribution of bottle graphs, such as the example in Figure 1. A bottle graph consists of multiple boxes, with different heights and widths, stacked on top of each other. The total height of the stack is the total running time of the application. Each box represents a thread. The height of each box can be interpreted as the share of that thread in the total execution time. The width of each box represents the amount of parallelism that that thread exploits, i.e., the average number of threads that run concurrently with that thread, including itself. The area of each box is the total running time of that thread.

The boxes are stacked by decreasing width, i.e., the widest at the bottom and the narrowest at the top, and they are centered, which makes the resulting graph look like a (two-dimensional) bottle. As a design choice, we center the graph around a vertical parallelism line of zero, making the graph symmetric. The amount of parallelism can be read either left or right from this line, e.g., a thread with a parallelism of 2 has a box that stretches to 2 on both sides of the parallelism axis.

The example bottle graph in Figure 1 represents a multi-threaded Java program, namely the DaCapo lusearch benchmark running with Jikes RVM on an 8-core Intel processor (see Section 3 for a detailed description of the experimental setup). This program takes 3.28 seconds to execute. There are 7 threads with visible bottle graph boxes, each having a different execution time share and different parallelism. The bottom four boxes represent application threads with a parallelism of approximately 4, and there is a main thread that performs benchmark initialization that has limited parallelism. There are two garbage collection (GC) threads, but the one with limited execution time share is a GC initialization thread, while the other GC thread that performs stop-the-world collection has a parallelism of only 1, because it runs alone.

Bottle graphs are an insightful way of visualizing multi-threaded program performance. Looking at the width of the boxes shows how well a program is parallelized. Threads that have low parallelism and have a large share in the total execution time appear as a large 'neck' on the bottle, which shows that they are a performance bottleneck. Bottle graphs thus naturally point to the most fruitful directions for efficiently optimizing a multi-threaded program. The next section defines the two dimensions of the boxes. In Section 2.2, we explain how we measure unmodified programs running on existing processors in order to construct bottle graphs.

2.1 Defining Box Dimensions

Each box in the bottle graph has two dimensions, the height and the width, representing the thread's share in the total execution time and parallelism, respectively.

2.1.1 Quantifying a thread's execution time share

Attributing shares of the total execution time to each of the threads in a multi-threaded program is not trivial. One cannot simply take the individual execution times of each of the threads, because their sum is larger than the program's execution time due to the fact that threads run in parallel. Individual execution times also do not account for variations in parallelism: threads that have low parallelism contribute more to the total execution time than threads with high parallelism. To account for this effect, we define the *share each thread has in the total execution time* (or *the height of the boxes*) as its individual execution time divided by the number of threads that are running concurrently (including itself), as proposed in our previous work [7]. So, if n threads run concurrently, they each get accounted one nth of their execution time.

Of course, the number of threads that run in parallel with a specific thread is not constant. While a thread is running, other threads can be created or destroyed or scheduled in or out by the operating system. To cope with this behavior, we divide the total execution time into intervals. The intervals are delimited by events that cause a thread to activate (creation or scheduling in) or deactivate (destruction/joining or scheduling out). As a result, the number of active threads is constant in each interval. Then we calculate the share of the active threads in each interval as the interval time divided by the number of active threads. Inactive threads do not get accounted any share. The final share of the total execution time of each thread is then the sum of the shares over all intervals for which that thread was active. The sum of all threads' shares equals the total execution time.

We formalize the time share accounting in the following way. Assume t_i is the duration of interval i, r_i is the number of running threads in that interval, and R_i is the set containing the thread IDs of the running threads (therefore $|R_i| = r_i$). Then the total share of thread j equals

$$S_j = \sum_{\forall i: j \in R_i} \frac{t_i}{r_i}. \tag{1}$$

The execution time share of a thread is the height of its box in the bottle graph. Therefore, the sum of all box heights, or the height of the total graph, is the total program execution time.

2.1.2 Quantifying a thread's parallelism

The other dimension of the graph – *the width of the boxes* – represents the *amount of parallelism of that thread*, or the number of threads that are co-running with that thread, including itself. A thread that runs alone has a parallelism of one, while threads that run concurrently with $n-1$ other threads have a parallelism of n. Due to the fact that the amount of parallelism changes over time, this number can also be a rational number.

The calculation of the execution time share already incorporates the amount of parallelism by dividing the execution time by the number of concurrent threads. We define parallelism as the time a thread is active (its individual running time) divided by its execution time share. Formally, the parallelism of thread j is calculated as

$$P_j = \frac{\sum_{\forall i:j \in R_i} t_i}{S_j} = \frac{\sum_{\forall i:j \in R_i} t_i}{\sum_{\forall i:j \in R_i} \frac{t_i}{r_i}}, \tag{2}$$

where $\sum_{\forall i:j \in R_i} t_i$ is the sum of all interval times where thread j is active, which is its individual running time.

Equation 2 in fact calculates the weighted harmonic mean of the number of concurrent threads, i.e., the harmonic mean of r_i weighted by the interval times t_i. It is therefore truly the average number of concurrent threads for that specific thread. We choose harmonic mean because metrics that are inversely proportional to time (e.g., IPC or parallelism) should be averaged using the harmonic mean while those proportional to time (e.g., CPI) should be averaged using the arithmetic mean [14].

Another interesting result from this definition of parallelism is that the execution time share multiplied by the parallelism – $S_j \times P_j$ – equals the individual running time of the thread, or in bottle graph terms: the height multiplied by the width, i.e., *the area of the box*, equals *the running time of a thread*. If we consider the running time of a thread as a measure of *the amount of work a thread performs*, then we can interpret the area of a box in the bottle graph as that thread's work. Due to parallelism (the width of the box), a thread's impact on execution time (the height of the box) is reduced.

This bottle graph design enhances their intuitiveness – a lot of information can be seen in one visualization – and quickly facilitates targeted optimization. Reducing the amount of work (area) of a thread that has a narrow box will result in a higher total program execution time (height) reduction compared to reducing the work for a wide box. The impact of a thread on program execution time can also be reduced by increasing its parallelism, which increases the width of the box and therefore decreases its height, while the area (amount of work) remains the same.

2.2 Measuring Bottle Graph Inputs

Now that we have defined how the bottle graphs are constructed, we design a tool that measures the values in order to construct a bottle graph of an application running on an actual processor. The tool needs to detect:

1. The number of active threads, to calculate the r_i values.
2. The IDs of the active threads, to know which threads should be accounted time shares.
3. Events that cause a thread to activate and deactivate, to delimit intervals.
4. A timer that can measure the duration of intervals, to get the t_i numbers.

The operating system (OS) is a natural place to construct our tool, as it already keeps track of thread creation, destruction, and scheduling, and has fine-grained timing capabilities. We build a tool to gather the necessary information to construct bottle graphs using kernel modules with Linux versions 2.6 and 3.0. The kernel modules are loaded using a script that requires root privileges. Communication with the modules (e.g., for communicating the name of the process that should be monitored) is done using writes and reads in the /proc directory. We use kernel modules to intercept system calls that perform thread creation and destruction, that schedule threads in and out, and that do synchronization with futex (which implements thread yielding). Our tool keeps track of the IDs of active threads and the timestamp of interval boundaries. Our modules also keep track of two counters per thread: one to accumulate the running time of the thread (i.e., the total time it is active) and the other to accumulate the execution time share (i.e., the running time divided by the number of concurrent threads).

A kernel module is triggered upon a (de)activation call, and updates state and thread counters in the following way. The module obtains the current timestamp, and by subtracting the previous timestamp from it, determines the execution time of the interval that just ended. It adds that time to the running time counter of all threads that were running in the past interval. It also divides that interval's time by the number of active threads, and adds the result to the time share counters of the active threads. Subsequently, the module changes the set of running threads according to the information attached to the call (thread activation or deactivation, and thread ID), and adapts the number of active threads. It also records the current timestamp as the beginning of the next interval. When the OS receives a signal from software, the two counters for each thread are written out, and this information is read by a script that generates the bottle graphs.

There are several advantages of using kernel modules to measure bottle graph components:

1. The program under study does not need to be changed; the tool works with unmodified program binaries.
2. The modules can be loaded dynamically; there is no need to recompile the kernel.
3. We can make use of a nanosecond resolution timer, which enables capturing very short active or inactive periods. We used ktime_get and verified in /proc/timer_list that it has nanoscale resolution.
4. In contrast with sampling, our kernel modules continuously monitor all threads' states accurately, and aggregate metric calculations online without loss of information.
5. The extra overhead is limited, because the calculations are simple and need to be done only on thread creation and scheduling operations. On average, we measure an average 0.68% increase in program execution time com-

pared to disabling the kernel modules, with a maximum of 1.11%, see Table 1 for per-benchmark overhead numbers.

Discussion of design decisions. In the design of our tool and experiments, we have made some methodological decisions, which do not limit the expressiveness of bottle graphs. We keep track of only synchronization events caused by futex, and thus our tool does not take into account busy waiting in spin loops, or interference between threads in shared hardware resources (e.g., in the shared cache and in the memory bus and banks). Threading libraries are designed to avoid long active spinning loops and yield threads if the expected waiting time is more than a few cycles, so we expect this to have no visible impact on the bottle graphs. Interference in hardware resources is a low-level effect that is less related to the characterization of the amount of parallelism in a multi-threaded program. When a thread performs I/O behavior, the OS schedules out that thread. Thus, we choose not to track I/O system calls in our kernel modules because most I/O behavior is already accounted for as inactive. Furthermore, we provide sufficient hardware contexts in our hardware setup, i.e., at least as many as the maximum number of runnable threads. We thus ensure that threads are only scheduled in and out due to synchronization events, and factor out the impact of scheduling due to time sharing a hardware context. If the number of hardware contexts were less than the number of runnable threads, the bottle graph's measured parallelism would be determined more by the machine than by the application characteristics.

In the majority of our results, we read out our cumulative thread statistics, including running time and execution time share, at the end of the program run. We then construct bottle graphs that summarize the behavior of the entire application execution on a per-thread basis. However, our tool can be given a signal at any time to output, and optionally reset, thread counters, so that bottle graphs can be constructed throughout the program run. Thus, bottle graphs can be used to explore phase behavior of threads within a program run (as we do in Section 5), or analyze particular sections of code for scalability bottlenecks. Furthermore, while our OS modules keep track of counters per-thread, in the case of thread pools, the bottle graph scripts could be modified to group separate threads into one component, if desired. Both execution time share and parallelism are defined such that it is mathematically sound to aggregate thread components.

3. Methodology

Now that we have introduced the novel and intuitive concept of bottle graphs, we perform experiments on unmodified applications running on real hardware to demonstrate the usefulness of bottle graphs. While bottle graphs can be used to analyze any multi-threaded program, in this paper we chose to analyze managed language applications. Not only are managed languages widely used in the community, but they also provide the added complexity of runtime service threads that manage memory and do dynamic compilation alongside the application. Thus, we can analyze both application and service thread performance and scalability using our visualization tool.

Benchmark	Suite	Version	ST/MT	Overhead
antlr	DaCapo	2006	ST	0.40%
bloat	DaCapo	2006	ST	0.64%
eclipse	DaCapo	2006	ST	0.70%
fop	DaCapo	2006	ST	0.20%
jython	DaCapo	2009	ST	0.80%
luindex	DaCapo	2009	ST	1.00%
avrora	DaCapo	2009	MT	1.11%
lusearch	DaCapo	2009	MT	0.32%
pmd	DaCapo	2009	MT	0.44%
pseudoJBB	SPEC	2005	MT	0.90%
sunflow	DaCapo	2009	MT	0.03%
xalan	DaCapo	2009	MT	0.91%

Table 1. Evaluated benchmarks and kernel module overhead. ST=single-threaded, MT=multi-threaded.

We evaluate Java benchmarks, see Table 1, from the DaCapo benchmark suite [4] and one from the SPEC 2005 benchmark suite, pseudoJBB, which is modified from its original version to do a fixed amount of work instead of run for a fixed amount of time [19]. There are 6 single-threaded (ST) and 6 multi-threaded (MT) benchmarks.[1] Although bottle graphs are designed for multi-threaded applications, it is interesting to analyze the interaction between a single-threaded application and the Java virtual machine (JVM) threads. We use the default input set, unless mentioned otherwise.

For our experiments, we vary the number of application threads (2, 4 and 8 for the multi-threaded applications) and garbage collector threads (1, 2, 4 and 8). We experiment with different heap sizes (as multiples from the minimum size that each benchmark can run in), but we present results with two times the minimum to keep the heap size fairly small in order to exercise the garbage collector frequently so as to evaluate its time and parallelism components. We run the benchmarks for 15 iterations, and collect bottle graphs for every iteration individually, but present main results for the 13th iteration to show stable behavior.

We performed our experiments on an Intel Xeon E5-2650L server, consisting of 2 sockets, each with 8 cores, running a 64-bit 3.2.37 Linux kernel. Each socket has a 20 MB LLC, shared by the 8 cores. For our setup, we found that the number of concurrent threads rarely exceeds 8, with a maximum of 9 (due to a dynamic compilation thread). Therefore, we only use one socket in our experiments with Hy-

[1] Although eclipse spawns multiple threads, we found that the JavaIndexing thread is the only running thread for most of the time, so we categorize it as a single-threaded application.

perThreading enabled, which leads to 16 available hardware contexts. This setup avoids data traversing socket boundaries, which could have an impact on performance that is hardware related, as demonstrated in [18]. The availability of 16 hardware contexts does not trigger the OS to schedule out threads other than for synchronization or I/O.

We run all of our benchmarks on Jikes Research Virtual Machine version 3.12 [1]. We use the default best-performing garbage collector (GC) on Jikes, the stop-the-world parallel generational Immix collector [3]. In addition to evaluating the Jikes RVM in Sections 4 and 5, we also compare with the OpenJDK JVM version 1.6 [17] in Section 6. We use their throughput-oriented parallel collector (also stop-the-world) in both the young and old generations. It should be noted that OpenJDK's compacting old generation has a different layout than Jikes' old generation, and thus will have a different impact on both the application and collector performance.

Because we use a stop-the-world collector, we can divide the execution of a benchmark into application and collection phases. Application and GC threads never run concurrently; therefore, the application and GC thread components in the bottle graph can be analyzed in isolation. For example, the total height of all application thread boxes is the total application running time, and the same holds for the GC threads. Also, because the collector never runs concurrently with other threads, the parallelism of the GC boxes is the parallelism of the collector itself. However, an interesting avenue for future work is to analyze the scalability of Java applications running with a concurrent collector.

4. Jikes RVM and Benchmark Analysis

By varying the number of application threads, GC threads, heap size, and collecting results over many iterations, we have generated over 2,000 bottle graphs. We describe the main findings from this study in this section, together with a few bottle graphs that show interesting behavior. We refer the interested reader to the additional supporting material for all bottle graphs generated during this study, available at `http://users.elis.ugent.be/~kdubois/bottle_graphs_oopsla2013`.

We first define some terminology that will be used to describe the bottle graphs we gathered for Java. *Application work* is the sum of all active execution times of all application threads (excluding JVM threads), i.e., the total area of all application thread boxes.[2] Likewise, we define *application time* as the sum of all execution time shares of all application threads, i.e., the total height of the application thread boxes. Along the same lines, we define *garbage collection work* as the sum of all GC threads' active execution times, i.e., the total area of all GC thread boxes, and *garbage collection time* as the sum of all GC threads' execution time shares, i.e., the total height of the GC thread boxes.

We will discuss collector and application performance and their impact on each other in Sections 4.2 and 4.3, but we first show and analyze bottle graphs for all benchmarks in the next section (at the steady-state 13th iteration). In Section 4.4, we also analyze the impact of the optimizing compiler by comparing the first and later iterations.

4.1 Benchmark Overview

Figure 2 shows bottle graphs for all evaluated benchmarks (only the threads that have a visible component in the bottle graph are shown). All graphs have two GC threads, and for the multi-threaded applications, we use four application threads. In general, turquoise boxes represent Jikes' MainThread which calls the application's main method. GC thread boxes are always presented in brown (including the GC controller thread, which explains the third GC box that appears in some graphs), while the dynamic compiler (called *Organizer*) is always presented in dark green. Application thread colors vary per graph. We found that all other JVM threads have negligible impact on execution time, and thus are not visible in the bottle graphs.

These graphs show the intuitiveness and insightfulness of bottle graphs. Single-threaded benchmarks can be easily identified as having a single large component with a parallelism of one. The graph of eclipse clearly shows that it behaves as a single-threaded application, as the JavaIndexing thread dominates performance, although it spawns multiple threads. Apart from the application and GC threads, antlr also has a visible Organizer thread, which is the only other JVM thread that was visible in all of our graphs. This thread has a parallelism of two, meaning that the JVM compiler always runs with one other thread, the MainThread in this case. Because it has a small running time, it does not have much impact on the parallelism of the main thread.

When we look at the multi-threaded benchmarks, we see that for lusearch, pseudoJBB, sunflow and xalan, the application threads have a parallelism of four, meaning that these benchmarks scale well to four threads. PseudoJBB has a rather large sequential component (in the MainThread), compared to the others. PseudoJBB does more initialization before it spawns the application threads, one per warehouse, to perform the work. Avrora is different in that it spawns six threads instead of four. This benchmark simulates a network of microcontrollers, and every microcontroller is simulated by one thread. Therefore, the number of threads spawned by avrora depends on the input, and the default input has six microcontrollers. The bottle graph reveals that the parallelism of avrora's application threads is limited to 2.4, although there are six threads and 16 available hardware contexts. Avrora uses fine-grained synchronization to accurately model the communication between the microcon-

[2] We classify the MainThread as part of the application. The MainThread does some initialization and then calls the main method of the application. For single-threaded applications, all of the work is done in this MainThread. For multi-threaded applications, it does some initialization and then spawns the application threads.

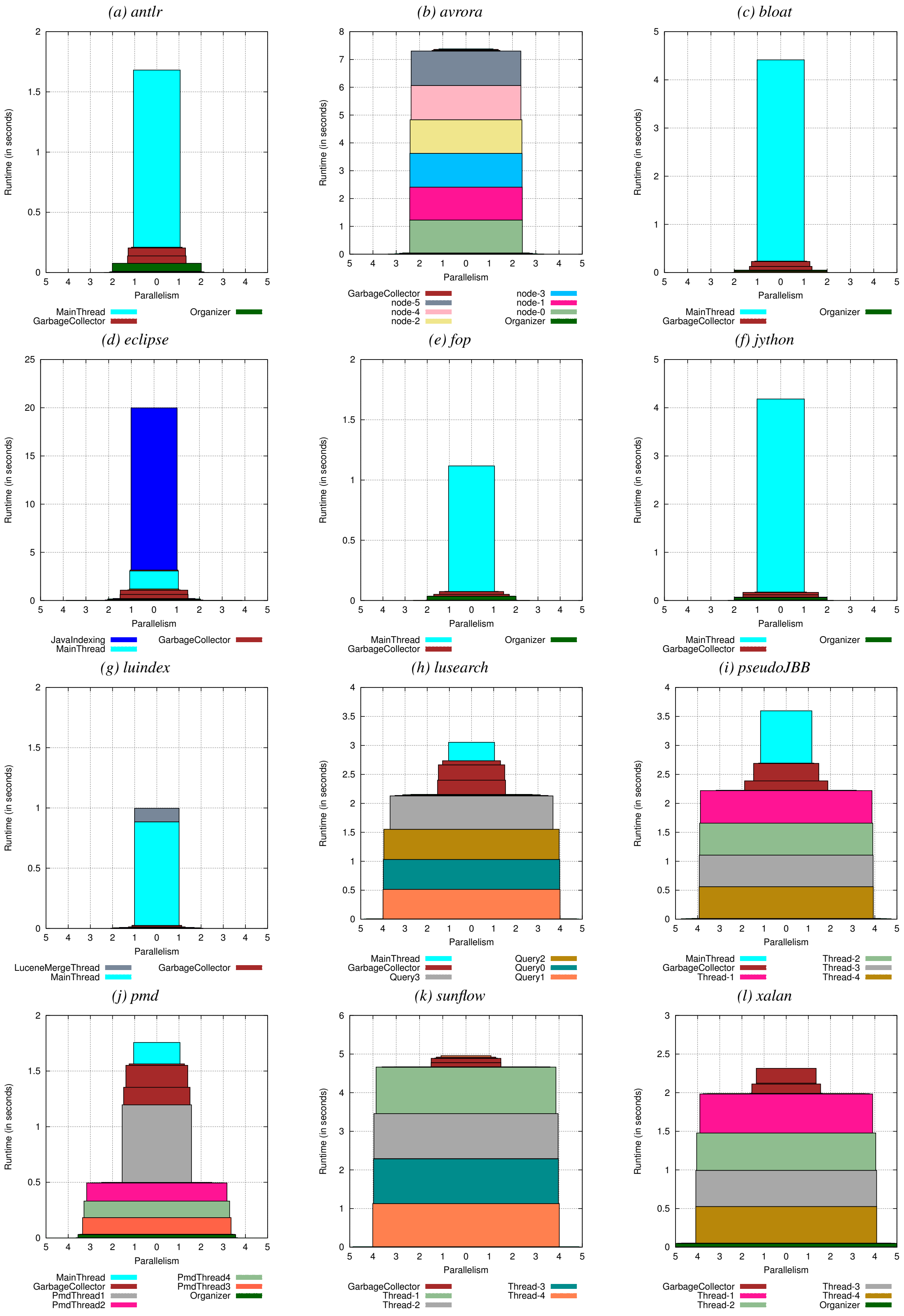


Figure 2. Bottle graphs for all benchmarks for the 13th iteration with 2 GC threads. Multi-threaded applications are run with 4 application threads.

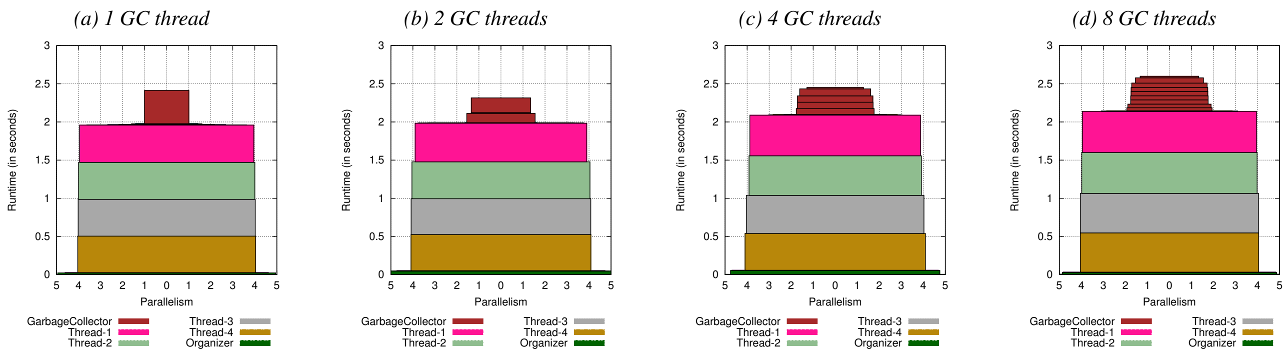


Figure 3. Xalan: scaling of GC threads with 4 application threads.

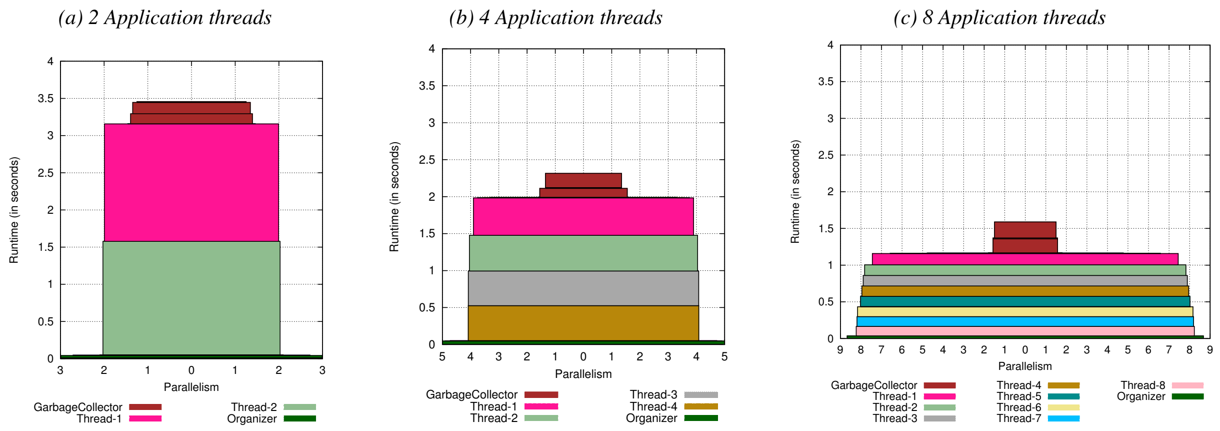


Figure 4. Xalan: scaling of application threads with 2 GC threads.

trollers, which reduces the exploited parallelism. This problem could potentially be solved by simulating close-by microcontrollers in one thread, instead of one thread per microcontroller, which will reduce the synchronization between the threads. Pmd is another interesting case: three of the four threads have a parallelism of more than three, but one thread has much lower parallelism and a much larger share of the execution time (PmdThread1). Pmd has an imbalance problem between its application threads. In Section 5, we analyze bottle graphs at various times within pmd's run to help elaborate on the cause of this parallelism limitation and provide suggestions to improve balance.

Although two GC threads are used in these experiments, the average parallelism of garbage collection is around 1.4. The next section explores varying the number of threads, and elaborates on the impact of the number of GC and application threads on garbage collector performance.

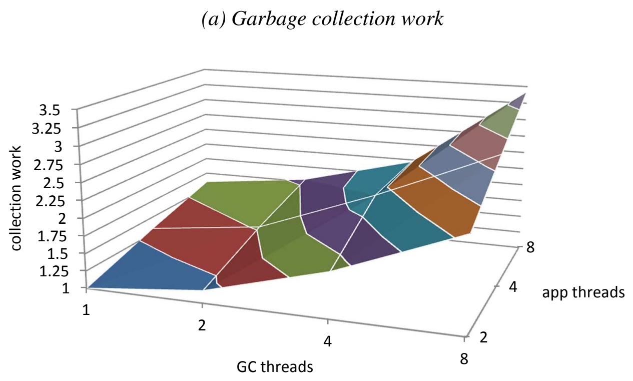
4.2 Garbage Collection Performance Analysis

We now explore what the bottle graphs reveal about garbage collection performance, while varying the numbers of application and collection threads. We analyze in depth the variation in the amount of garbage collection time (sum of GC box heights) and work (sum of GC box areas). Figures 3 and 4 show bottle graphs for xalan with increasing number of GC threads (Figure 3) and increasing number of application threads (Figure 4). We include only the xalan results here because of space constraints; this benchmark has representative behavior with respect to collection. Figure 5 shows the average collection work (a) and time (b) for all multi-threaded benchmarks, as a function of the number of GC threads and the number of application threads.[3] The values are normalized to the configuration with 2 application threads and 1 GC thread. Figure 6 shows the same data for the single-threaded benchmarks, obviously without the dimension of the number of application threads.

We make the following observations:

Collection work increases with an increasing number of collection threads. When the number of GC threads increases, the total collection work increases, i.e., the sum of the running times of all GC threads increases, see Figures 5 (a) and 6. For xalan, total collection work—which equals the total area of all GC thread boxes—increases by 73% from 1 to 8 GC threads (see Figure 3). For all multi-threaded bench-

[3] We exclude the numbers for avrora and pseudoJBB for Figures 5, 7, 8, 9, 15, and 16. For these benchmarks, it is impossible to vary the number of application threads independently from the problem size.



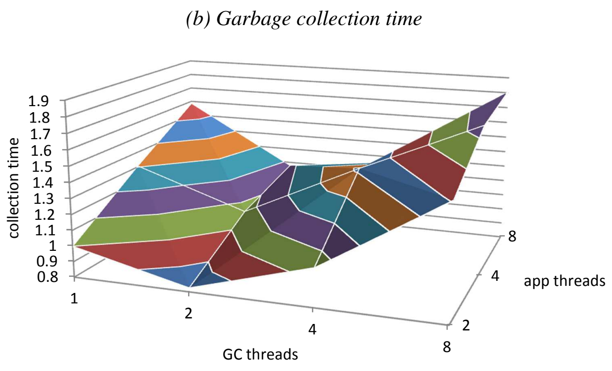


Figure 5. Average garbage collection work (a) and time (b) as a function of application and GC thread count (multi-threaded applications). Numbers are normalized to the 2 application and 1 GC thread configuration. *Collection work increases with increasing GC thread count and increasing application thread count, and 2 GC threads results in minimum collection time for all application thread counts.*

marks, there is an increase of 120% (averaged over all application thread counts), and 198% for the single-threaded benchmarks.

There are a number of reasons for this behavior. First, more threads incur more synchronization overhead, i.e., extra code to perform synchronization due to GC threads accessing shared data. Figure 16 in Section 6 (Jikes line) shows that the number of futex calls significantly increases as the number of GC threads increases. Second, because garbage collection is a very data-intensive process and the last-level cache (LLC) is shared by all cores, more GC threads will lead to more interference misses in the LLC, leading to a larger running time. Figure 7 shows the number of LLC cache misses as a function of the number of GC threads and the number of application threads for all multi-threaded applications, normalized to the 2 application, 1 GC thread configuration. The number of LLC cache misses increases significantly when the number of GC threads increases (more than 2.5 times for 8 GC threads compared to 1 GC thread), which raises the total execution time of the GC threads.

Collection work increases with an increasing number of application threads. There is an increase in the amount of time garbage collection threads are actively running (or their total work), as we go to larger application thread counts. For xalan, collection work increases by 60% if the number of application threads is increased from 2 to 8 (see Figure 4). For all multi-threaded benchmarks, we see an average increase of 47% (over all GC thread-counts), in Figure 5 (a). To explain this behavior, we refer to Figure 8, which shows the average number of collections that occurred during the execution of the multi-threaded benchmarks,[4] again as a function of the number of GC and application threads. When in-

[4] We exclude pmd from this graph, because due to its imbalance, only one thread is running during a significant portion of its execution time. Since this portion increases with the number of application threads, the number of collections decreases, while we see an increase for all other benchmarks.

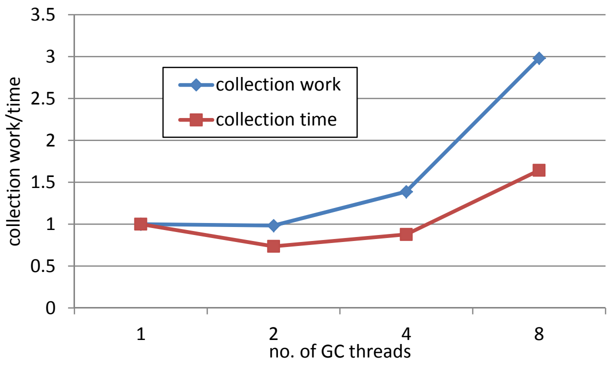


Figure 6. Average collection work and time as a function of GC thread count (single-threaded applications). Numbers are normalized to the 1 GC thread configuration. *Collection work increases with increasing GC thread count and collection time is minimal with 2 GC threads.*

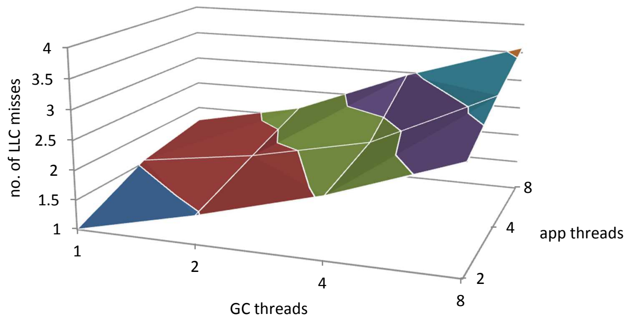


Figure 7. Average number of LLC misses as a function of application and GC thread count (multi-threaded applications). Numbers are normalized to the 2 application and 1 GC thread configuration.

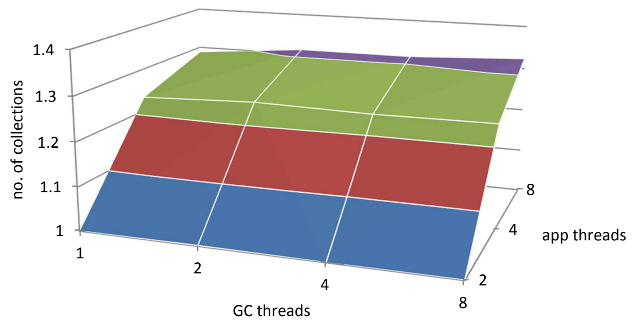


Figure 8. Average number of collections as a function of application and GC thread count (multi-threaded applications). Numbers are normalized to the 2 application and 1 GC thread configuration.

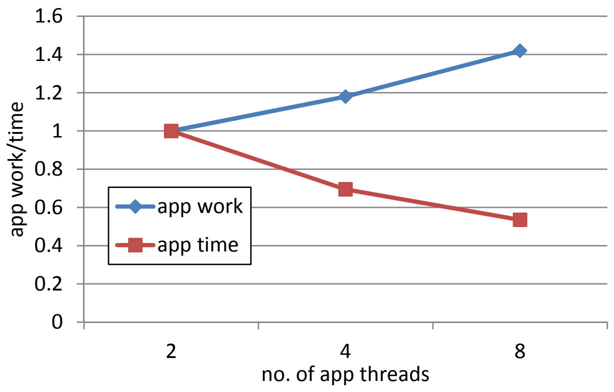


Figure 9. Average application work and time as a function of application thread count (multi-threaded applications). Numbers are normalized to the 2 application thread configuration. *Application time decreases with increasing application thread count, but the decrease is limited because application work increases with more application threads.*

creasing the number of GC threads, the number of collections does not increase. However, the number of application threads has a clear impact on the number of collections: the more application threads, the more collections. The intuition behind this observation is that more threads generate more live data in a fixed amount of time (because Jikes has thread-local allocation where every thread gets its own chunk of memory to allocate into), so the heap fills up faster compared to having fewer application threads. The more collections, the more collection work that needs to be done.

2 GC threads are optimal regardless of the number of application threads. Figures 5 (b) and 6 show that garbage collection time is minimal for two GC threads in Jikes, even for lower and higher application thread counts. Although the amount of work increases when the number of GC threads is increased from 1 to 2, collector parallelism is also increased (from 1 to 1.5), leading to a lower net collection time. When the number of GC threads is further increased to 4 and 8, parallelism increases only slightly (to 1.7 and 1.8, respectively), and does not compensate for the extra amount of work, leading to a net increase in collection time. Although we present results here for only two times the minimum heap size, we found two GC threads to be optimal for other heap sizes as well.

4.3 Application Performance Analysis

We analyze changes in the application time and work as we vary the number of application threads, which seem to suffer less from limited parallelism than the garbage collector. Figure 9 shows the application execution time (excluding garbage collection time) and work as a function of the number of application threads (for the multi-threaded applications), keeping the GC threads at two. Numbers are normalized to those for two application threads.

Application time decreases with an increasing number of application threads, but the decrease is limited because application work increases due to the overheads of parallelism. When the application thread count is increased, application time does decrease, but not proportionally to the number of threads. Compared to two application threads, execution time decreases with a factor of 1.4 for four application threads and 1.9 for 8 threads. Part of the reason this decrease is not higher is the limited parallelism (average application parallelism equals 1.9, 3.1 and 4.7, for 2, 4 and 8 application threads, respectively). However, this does not fully cover the smaller application speedup: from 2 to 8 application threads, parallelism is increased by a factor of 2.5, while execution time is reduced by a factor of only 1.9. This difference is explained by the fact that application work also increases with the number of application threads, see Figure 9. This increase is due to synchronization overhead (the number of futex calls for the application increases from 1.5 to 4.7 calls per ms when the number of application threads increases from 2 to 8) and an increasing number of LLC misses due to more interference (see also Figure 7). Increasing the number of application threads leads to more application work and increased parallelism, resulting in a net reduced execution time, but due to the extra overhead, the execution time reduction is smaller than the thread count increase.

4.4 Compiler Performance Analysis

Lastly, while generating our bottle graphs across many benchmark iterations, we noticed the difference between startup and steady state behavior discussed in detail in Java methodology research [4]. Figure 10 shows the bottle graphs of one single-threaded benchmark, jython, during the first, 9th and 11th iterations. During the first iteration, we see a large overall execution time, and a large (one second) time share for the Organizer thread. This JVM service thread performs dynamic compilation concurrently with the application (it has a parallelism of two), and thus is very active in the first iteration, but is much more minimal in iteration nine.

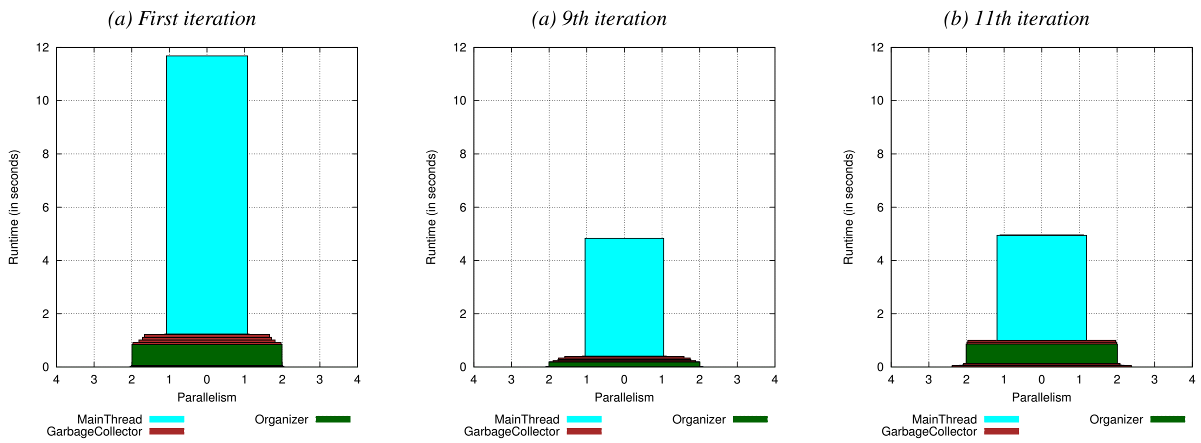


Figure 10. Jython: behavior of Organizer thread over different iterations for 4 GC threads.

Iteration nine has a reduced execution time because the Java source code is now optimized and the benchmark is running more at steady-state. However, the bottle graph for iteration 11 shows an increased Organizer thread component. This behavior is specific to this benchmark; other applications see the Organizer box disappear in all higher iterations. Jython is different in that it dynamically interprets python source code into bytecode, and then runs it. Jython actually runs a benchmark within itself, and Jikes continues to optimize the generated bytecode with the optimizing compiler at various iterations, because the compiler is probabilistic and is triggered unpredictably. Thus, bottle graphs are also useful for seeing program and JVM variations between iterations.

5. Solving the Poor Scaling of Pmd

We have analyzed the performance of both Java applications and Jikes RVM's service threads using bottle graphs. As shown in Figure 2, and discussed in Section 4.1, pmd has one thread that has significantly limited parallelism. We now analyze this bottleneck and propose suggestions on how to fix pmd's scalability problem.

Figure 11 shows the bottle graphs for pmd for 2, 4 and 8 application threads, while keeping the collection threads at two and using the default input set. With two application threads, the left graph shows these threads have approximately the same height and width (a parallelism close to two). However, for 4 and 8 threads, pmd clearly has an imbalance issue: there is one thread that has less parallelism and a larger execution time share than the other threads. To understand the cause, we gathered bottle graphs at several time intervals (every 0.5 seconds) within the 13th iteration of the benchmark, running with 2 GC and 8 application threads, shown in Figure 12. We see that after the first 0.5 seconds, although there is some variation, the application threads are still fairly balanced in regards to parallelism. Starting from after the second time interval, one application thread has limited parallelism and a larger share of execution time (PmdThread7). After one second of execution, that same thread continues to run alone while all other application threads have finished their work.

Pmd is a (Java) source code analyzer, and finds unused variables, empty catch blocks, unnecessary object creation, etc. It takes as input a list of source code files to be processed. The default DaCapo input for pmd is a part of the source code files of pmd itself. There is also a large input set, which analyzes all of the pmd source code files. Internally, pmd is parallelized using a work stealing approach. All files are put in a queue, and the threads pick the next unprocessed file from the queue when they finish processing a file. Compared to static work partitioning, work stealing normally improves balance, because one thread can process a large job, while another thread processes many small jobs. Imbalance can only occur when one or a few jobs have such a large process time that there are not enough other small jobs to be run in parallel. This is exactly the case for the default input set of pmd. There are 220 files, with an average size of 3.8 KB. However, there is one file that is 240 KB, which is around 63 times larger than the average. Therefore, the thread that picks that file will always have a larger execution time than the other threads, and there are not enough other files to keep the other threads concurrently busy.

This problem is partly solved when using the large input set, which also has the same big file, but there are 570 files in total, so more files can be processed concurrently with the big file. Figures 13(a)–(c) show the bottle graphs for the large input set with 2, 4 and 8 application threads. The imbalance problem is solved for 2 and 4 application threads, but is still present for 8 threads (although less pronounced compared to the default input set). The more threads there are, the more other jobs are needed to run concurrently with the large job. Figure 13(d) shows the bottle graph for 8 threads and the large input set excluding that one big file, which leads to a balanced execution.

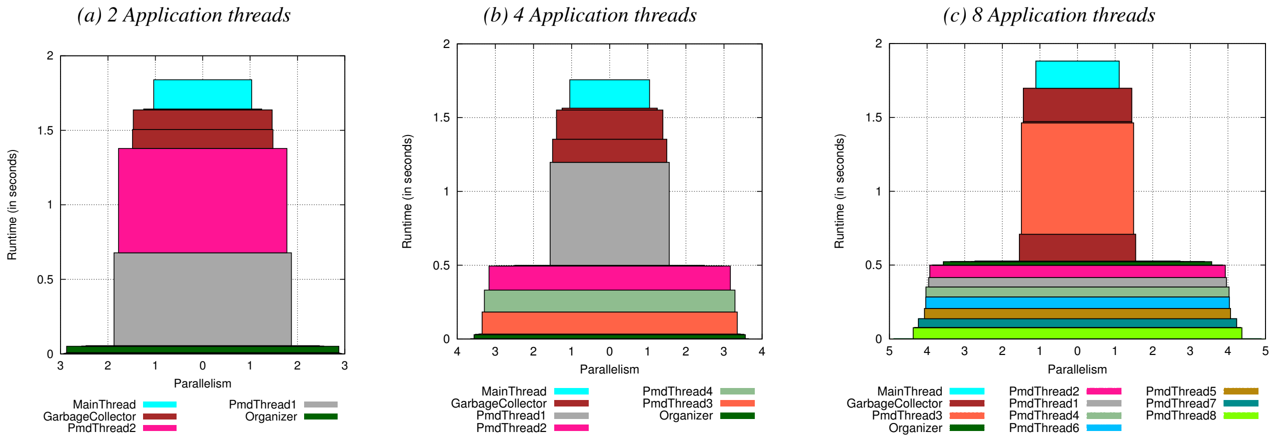


Figure 11. Pmd: scaling of application threads with 2 GC threads (default input set).

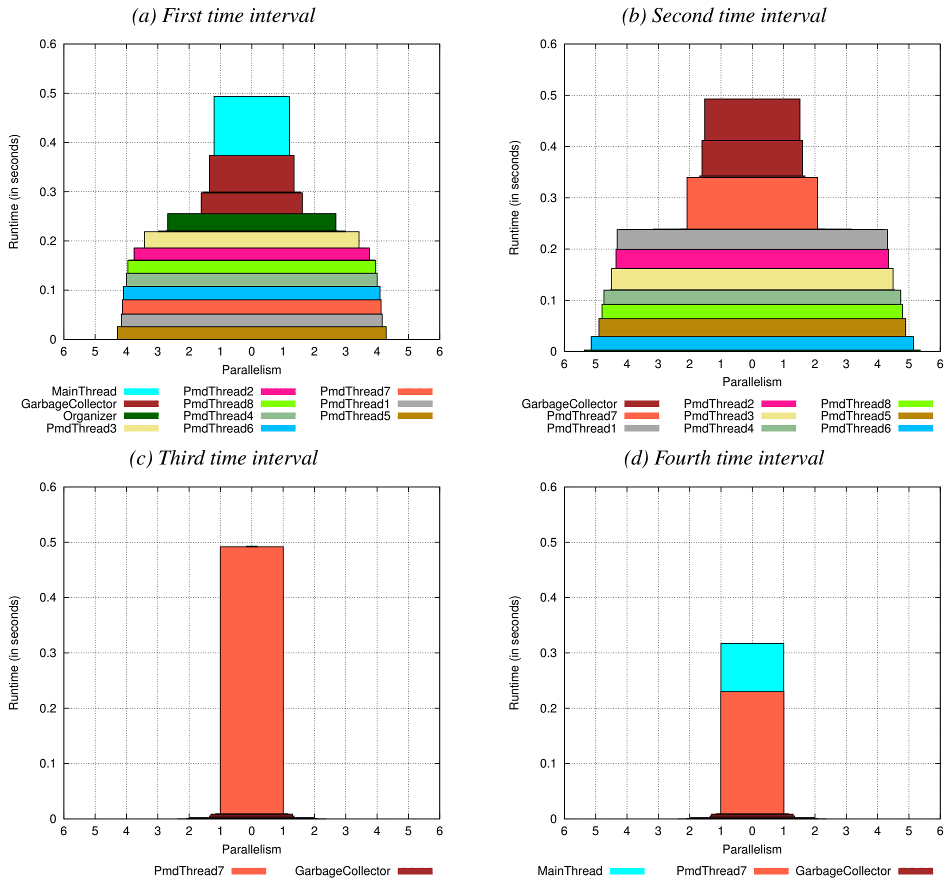


Figure 12. Pmd: bottle graphs taken every 0.5 seconds with 2 GC threads and 8 application threads (default input set).

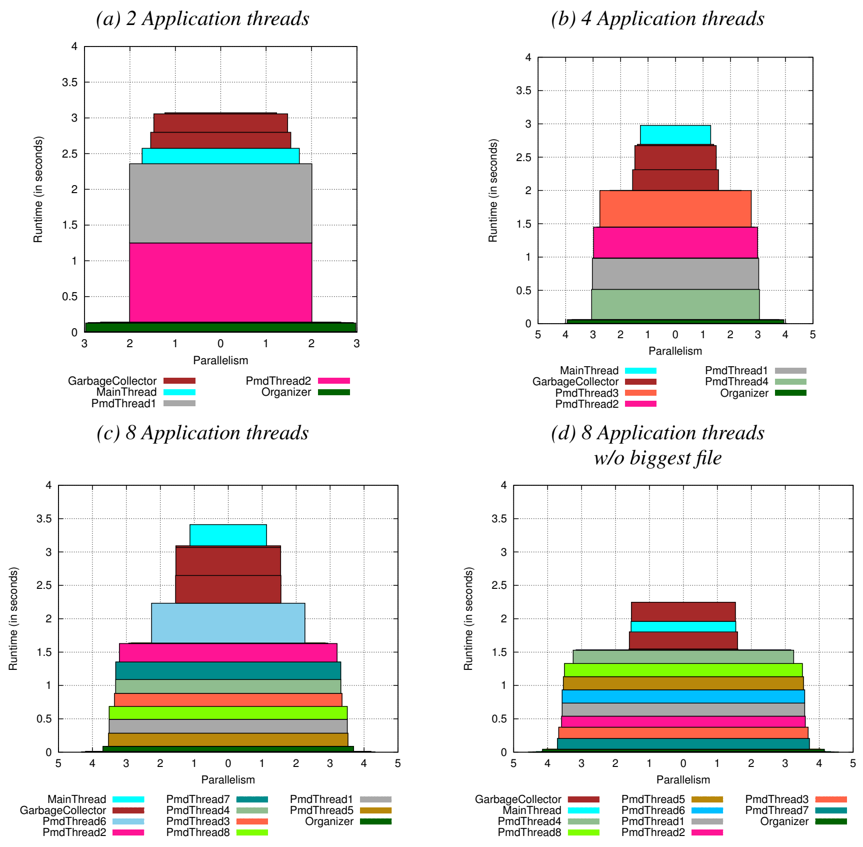


Figure 13. Pmd: scaling of application threads with 2 GC threads (large input set). For the fourth graph, the biggest source file is removed from the input set.

We can conclude that users of pmd should make sure that there is no file that is much larger than the others to prevent an imbalanced, and therefore inefficient, execution. The balance can also be improved by making the scheduler in pmd more intelligent. For example, the files to be processed can be ordered by decreasing file size, such that big files are processed first and not after a bunch of smaller files. In that case, there are more small files left for the other threads, and balance is improved. Another, probably more intrusive, solution is to provide the ability to split a single file across multiple threads.

Apart from imbalance, there is also a problem of limited parallelism in pmd. Figure 13(d) shows that the parallelism of 8 application threads is only 3.5. We looked into the code and found a synchronized map data structure that is shared between threads and guarded by one lock. Reducing the time the lock is held and/or using fine-grained locking should improve parallelism, and therefore performance, for pmd.

6. Comparing Jikes to OpenJDK

In Section 4 we analyzed the performance of the garbage collector and the application with Jikes RVM. We made two novel observations: collector parallelism is limited, leading to an optimal GC thread count of 2, and that the number of GC threads and the number of application threads have an impact on the amount of collection work. We present here a similar analysis on the OpenJDK virtual machine, revealing that OpenJDK's garbage collector scales better than in Jikes, benefiting from up to 8 GC threads. However, we find that collection work still increases with the number of application threads and GC threads.

Figure 14 shows the bottle graphs for pseudoJBB with 4 application threads on OpenJDK, with an increasing GC thread count. We chose pseudoJBB graphs here because they are most illustrative of collector behavior, but other benchmarks have similar behavior. It is immediately clear that GC scales much better for OpenJDK than for Jikes. The average collection parallelism across all multi-threaded benchmarks is 1, 1.9, 3.3 and 4.5, for 1, 2, 4 and 8 GC threads, respectively. This parallelism is substantially larger than the 1.8 parallelism for 8 GC threads on Jikes.

We further investigate the performance of OpenJDK by viewing its collection work and time as a function of GC and application thread count in Figure 15. We present average collection work (a) and time (b) for the multi-threaded benchmarks normalized to the 1 GC and 2 application thread configuration (contrasted with Figure 5). We confirm that OpenJDK scales better than Jikes by seeing that collection

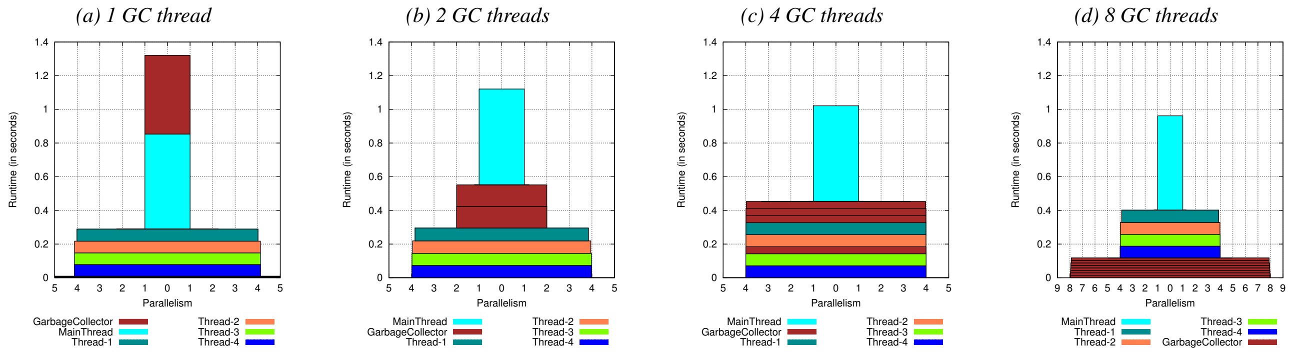


Figure 14. PseudoJBB: scaling of GC threads on OpenJDK, with 4 application threads.

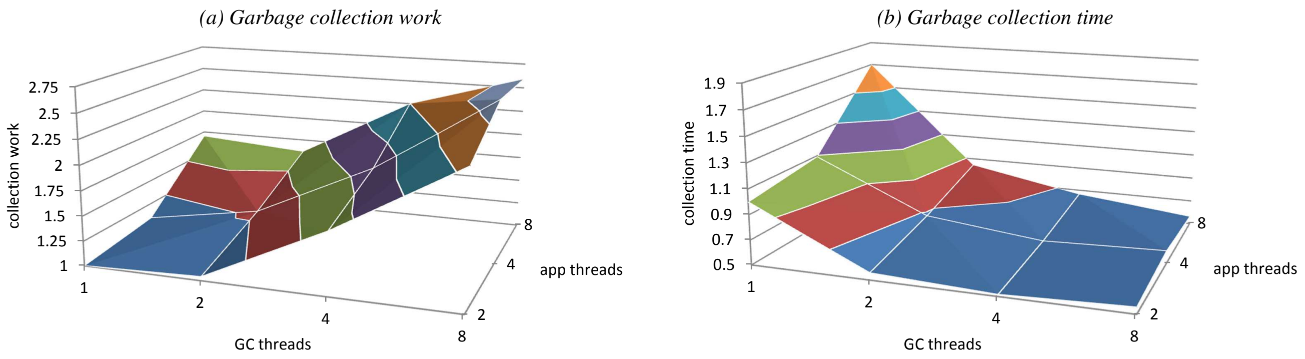


Figure 15. Average OpenJDK garbage collection work (a) and time (b) as a function of application and GC thread count (multi-threaded applications). Numbers are normalized to the 2 application and 1 GC thread configuration. *Garbage collection on OpenJDK scales better than on Jikes, but collection work also increases with increasing GC thread count and increasing application thread count.*

time decreases with an increasing number of GC threads in Figure 15(b). There is less synchronization during garbage collection in OpenJDK compared to Jikes, as evident in Figure 16 which shows the number of futex calls per unit of time as a function of the number of GC threads. Although the number of futex calls also increases with increasing GC thread count, the increase is much smaller for OpenJDK than for Jikes.

For OpenJDK, we found 4 GC threads to be optimal with either 2 or 4 applications threads, and 8 GC threads optimal for 8 application threads. Garbage collection on OpenJDK scales better than for Jikes RVM, but we observe that collector time slightly increases or does not decrease much between 4 and 8 GC threads, which suggests that garbage collection scaling on OpenJDK saturates at 4 to 8 GC threads. This is in line with the findings in [10], where the authors observe a decrease in collection time when the number of GC threads is increased from 1 to 6, but an increase when the number of GC threads is increased to 12 and more.

Our two other observations for Jikes, namely that collection work increases with GC thread count and application thread count, still hold for OpenJDK. Figure 15(a) for OpenJDK looks very similar to Figure 5(a) for Jikes. While we have observed in both JVMs a saturation point in the utility of increasing the parallelism of garbage collection threads, we still find that inter-thread synchronization has a significant impact on garbage collection performance.

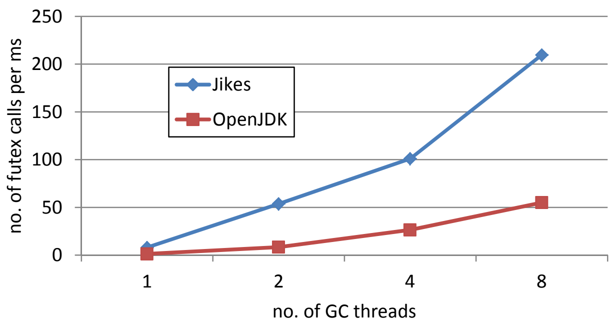


Figure 16. Average number of futex calls per ms during garbage collection as a function of GC thread count, with 4 application threads (multi-threaded applications).

7. Related Work

We now describe related work in performance visualization and Java parallelism analysis. We detail one particularly related visualization tool called WAIT in Section 7.1.1.

7.1 Performance Visualization

Software developers heavily rely on tools for guiding where to optimize code. Commercial offerings, such as Intel VTune Amplifier XE [12], Sun Studio Performance Analyzer [13], and PGPROF from the Portland Group [20] use hardware performance counters and sampling to derive where time is spent, and point the software developer to places in the source code to focus optimization. Additional features offered include suggestions for potential tuning opportunities, collecting lock and wait behavior, visualization of running and waiting threads over time, etc. The key feature of these tools is that they provide fairly detailed analysis at fine granularity in small functions and individual lines of code. Recent work focused on minimizing the overhead even further, enabling the analysis of very small code regions, such as critical sections [6]. Other related work [16] proposes a simple and intuitive representation, called Parallel Block Vectors (PBV), which map serial and parallel phases to static code. Other research proposes the Kremlin tool, which analyzes sequential programs to recommend sections of code that would get the most speedup from parallelization [9]. All of these approaches strive at providing fine-grained performance insight. Unfortunately, none of these approaches provide a simple and intuitive visualization and understanding of gross performance scalability bottlenecks in multi-threaded applications, as bottle graphs do and which is needed by software developers to guide optimization.

Some recent work in performance visualization focused on capturing and visualizing gross performance scalability trends in multi-threaded applications running on multicore hardware, but do not guide the programmer on where to focus optimization. Speedup stacks [8] present an analysis of the causes of why an application does not achieve perfect scalability, comparing achieved speedup of a multi-threaded program versus ideal speedup. Speedup stacks measure the impact of synchronization and interference in shared hardware resources, and attribute the gap between achieved and ideal speedup to the different possible performance delimiters. By doing so, speedup stacks easily point to scalability limiters, but present no data on which thread could be the cause, and do not suggest how to overcome these scalability limitations. Bottle graphs, on the other hand, provide detailed information on per-thread synchronization events, showing where optimization effort should be targeted, namely at the narrowest and tallest thread(s) in the graph. In our most recent work, we presented criticality stacks [7], which display per-thread contributions to total program performance. Criticality stacks focus on synchronization only, and do not incorporate parallelism as bottle

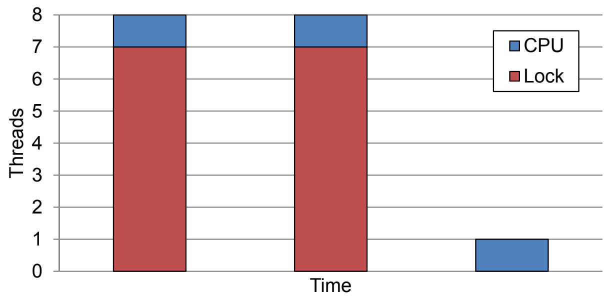


Figure 17. Output of WAIT for pmd running on OpenJDK with 8 application and 2 GC threads using a 1 second sampling rate. The graph shows 3 samples that monitor application thread status: *CPU* threads are active, while *Lock* threads are inactive.

graphs do. Moreover, this work requires hardware modifications and, while it points out thread imbalances, it does not suggest how much gain could be achieved by making a particular thread better able to co-execute with other threads.

7.1.1 Comparison to IBM WAIT

IBM WAIT[5] [2] is a performance visualization tool for diagnosing performance and scalability bottlenecks in Java programs, particularly server workloads. It uses a lightweight profiler that collects samples of information about each thread at regular points in time (configurable through a sampling rate parameter). WAIT records information on each thread's status (active, waiting or idle), locks (held or waiting for), and where in the code the thread is executing. This data is used to construct a graph that visualizes the threads' status over time (x-axis), each bar showing a sample point. The bar's height is the total number of threads for that sample, and the bar is color-coded by thread status. Figure 17 shows the output of WAIT for pmd running on OpenJDK with 8 application threads and 2 GC threads during the 13th iteration using a 1 second sampling rate, where *CPU* denotes active threads, and *Lock* denotes inactive threads. Information about code position and locks can be retrieved by clicking on the bars in the timeline.

While WAIT is a powerful analysis tool for Java programs, it has some limitations. First, it can be applied only to Java application threads, not to parallel programs written in other languages or to Java virtual machine service threads, both of which can be analyzed easily with our bottle graphs because we use OS modules. Second, WAIT is sampling-based, and thus collects a snapshot of information only at specific program points, with increasing overhead with finer-granularity sampling. To demonstrate this, Figure 17 shows results for pmd using at the lowest default sampling period value of 1 second. WAIT only collects 3 samples, which suggest that there is only one active (*CPU*) thread during the execution. We lowered the sampling rate to 50 milliseconds by

[5] `https://wait.ibm.com/`

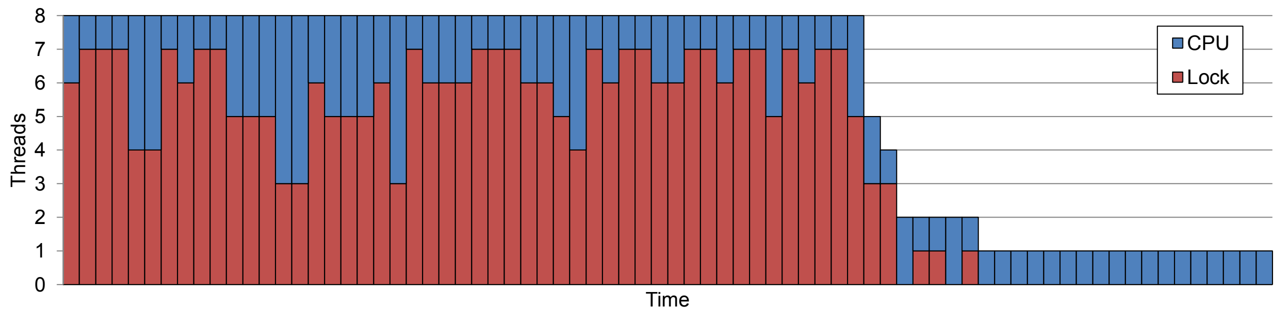


Figure 18. Output of WAIT for pmd running on OpenJDK with 8 application and 2 GC threads using a 50 millisecond sampling rate. The graph shows one bar per sample that monitors application thread status: *CPU* threads are active, while *Lock* threads are inactive.

modifying WAIT's scripts, and produced the more detailed graph in Figure 18 which reveals that the number of active threads varies over time. However, the overhead of using the 1 second versus 50 millisecond sampling period jumps from 0.87% to 16.62%, an order of magnitude larger than for our tool.

In contrast, bottle graphs contain much more information for lower overhead. Our OS modules are continually monitoring *every* thread status change, and *aggregating our execution time share and parallelism metrics* at all times in a multi-threaded program run, on a *per-thread basis*. For roughly the same overhead, we contrast Figure 12 showing bottle graphs at various times in a run of pmd with Figure 17 showing WAIT's three thread samples. WAIT's visual representation makes it hard to know that one of pmd's threads is a bottleneck throughout the run. In the end, pmd's parallel imbalance due to input imbalance is difficult to detect without analyzing the source code and input. In conclusion, bottle graphs' visualization of scalability per-thread facilitates grouping by category (such as for thread pools or garbage collection threads) in order to analyze the group's execution time share, parallelism, or work to pinpoint parallelism imbalances.

7.2 Java Parallelism Analysis

Analyzing Java performance and parallelism has become an active area of research recently. Most of these studies use custom-built analyzers to measure specific characteristics of interest. For example, Kalibera et al. [15] analyze concurrency, memory sharing and synchronization behavior of the DaCapo benchmark suite. They provide concurrency metrics and analyze the applications in-depth, focusing on inherent application characteristics. They do not provide a visual analysis tool to measure and quantify performance and scalability, and reveal bottlenecks on real hardware as we do.

Researchers recently analyzed the scalability problems of the garbage collector in the OpenJDK JVM [10]. They also confirm that the total collection times increase with the number of collector threads, without providing a visualization tool. They did follow-on work to optimize scalability at large thread-counts for the parallel stop-the-world garbage collection in OpenJDK [11]. Similarly, Chen et al. [5] analyzed scalability issues in the OpenJDK JVM, and provided explanations at the hardware level by measuring cache misses, DTLB misses, pipeline misses, and cache-to-cache transfers. They also explored the sizing of the young generation and measured the benefits of thread-local allocation. They did not, however, vary the number of collection threads or explore the scalability limitations of the parallel collector itself, or how it interacts with the application, as we do in this paper.

8. Conclusions

We have presented bottle graphs, an intuitive and useful tool for visualizing multi-threaded application performance, analyzing scalability bottlenecks, and targeting optimization. Bottle graphs represent each thread as a box, showing its execution time share (height), parallelism (width), and total running time (area). The total height of the bottle graph when these boxes are stacked on top of each other is the total application execution time. Because we place threads with the largest parallelism on the bottom, the neck of the bottle graph points to threads that offer the most potential for optimization, i.e. those with limited parallelism and large execution time share.

We have built light-weight OS modules to calculate the components of bottle graphs for unmodified parallel programs running on real hardware, with minimal overhead. Then we have used bottle graphs to analyze a set of 12 Java benchmarks, revealing scalability limitations in several well-known applications and suggesting optimizations. We have also used bottle graphs to analyze Jikes' JVM service threads, revealing very limited parallelism when increasing the number of garbage collection threads beyond two due to extra synchronization activity. We have compared this to OpenJDK's garbage collector which scales much better for our thread counts. Bottle graphs are a powerful visualization tool that is necessary for tackling multi-threaded application bottlenecks in modern multicore hardware.

Acknowledgments

We thank the anonymous reviewers for their constructive and insightful feedback. Stijn Eyerman is supported through a postdoctoral fellowship by the Research Foundation – Flanders (FWO). Additional support is provided by the FWO project G.0179.10N, the UGent-BOF project 01Z04109 and the European Research Council under the European Community's Seventh Framework Programme (FP7/2007-2013) / ERC Grant agreement no. 259295.

A. Appendix

See `http://users.elis.ugent.be/~kdubois/bottle_graphs_oopsla2013/` for more bottle graphs. This website also contains the OS modules tool to measure the data to construct bottle graphs.

References

[1] B. Alpern, C. R. Attanasio, A. Cocchi, D. Lieber, S. Smith, T. Ngo, J. J. Barton, S. F. Hummel, J. C. Sheperd, and M. Mergen. Implementing Jalapeño in Java. In *Proceedings of the Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 314–324, Nov. 1999.

[2] E. Altman, M. Arnold, S. Fink, and N. Mitchell. Performance analysis of idle programs. In *Proceedings of the Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 739–753, Oct. 2010.

[3] S. M. Blackburn and K. S. McKinley. Immix: A mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *Proceedings of the Annual ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 22–32, June 2008.

[4] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. L. Hosking, M. Jump, H. B. Lee, J. Moss, A. Phansalkar, D. Stefanovic, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 169–190, Oct. 2006.

[5] K.-Y. Chen, J. M. Chang, and T.-W. Hou. Multithreading in Java: Performance and scalability on multicore systems. *IEEE Transactions on Computers*, 60(11):1521–1534, Nov. 2011.

[6] J. Demme and S. Sethumadhavan. Rapid identication of architectural bottlenecks via precise event counting. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, pages 353–364, June 2011.

[7] K. Du Bois, S. Eyerman, J. B. Sartor, and L. Eeckhout. Criticality stacks: Identifying critical threads in parallel programs using synchronization behavior. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, pages 511–522, June 2013.

[8] S. Eyerman, K. Du Bois, and L. Eeckhout. Speedup stacks: Identifying scaling bottlenecks in multi-threaded applications. In *Proceedings of the International Symposium on Performance Analysis of Software and Systems (ISPASS)*, pages 145–155, Apr. 2012.

[9] S. Garcia, D. Jeon, C. M. Louie, and M. B. Taylor. Kremlin: Rethinking and rebooting gprof for the multicore age. In *Proceedings of the Annual ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 458–469, June 2011.

[10] L. Gidra, G. Thomas, J. Sopena, and M. Shapiro. Assessing the scalability of garbage collectors on many cores. *ACM SIGOPS: Operating Systems Review*, 45(3), Dec. 2011.

[11] L. Gidra, G. Thomas, J. Sopena, and M. Shapiro. A study of the scalability of stop-the-world garbage collectors on multicore. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 229–240, Mar. 2013.

[12] Intel. Intel VTune^{TM} Amplifier XE 2011. http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/.

[13] M. Itzkowitz and Y. Maruyama. HPC profiling with the Sun Studio^{TM} performance tools. In *Tools for High Performance Computing 2009*, pages 67–93. Springer, 2010.

[14] L. K. John. More on finding a single number to indicate overall performance of a benchmark suite. *ACM SIGARCH Computer Architecture News*, 32(4):1–14, Sept. 2004.

[15] T. Kalibera, M. Mole, R. Jones, and J. Vitek. A black-box approach to understanding concurrency in DaCapo. In *Proceedings of the Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 335–354, Oct. 2012.

[16] M. Kambadur, K. Tang, and M. A. Kim. Harmony: Collection and analysis of parallel block vectors. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, pages 452–463, June 2012.

[17] OpenJDK. *OpenJDK (Implementation of the Java SE 6 Specification), Version 1.6*. Oracle, 2006. URL `http://openjdk.java.net/projects/jdk6/`.

[18] J. B. Sartor and L. Eeckhout. Exploring multi-threaded Java application performance on multicore hardware. In *Proceedings of the Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 281–296, Oct. 2012.

[19] SPEC. *SPECjbb2005 (Java Server Benchmark), Release 1.07*. Standard Performance Evaluation Corporation, 2006. URL `http://www.spec.org/jbb2005`.

[20] STMicroelectronics. PGProf: parallel profiling for scientists and engineers. http://www.pgroup.com/products/pgprof.htm, 2011.