Analyzing the Scalability of Managed Language Applications with Speedup Stacks

Jennifer B. Sartor[*] Vrije Universiteit Brussel
Kristof Du Bois[*] Intel
Stijn Eyerman[*] Intel
Lieven Eeckhout Ghent University

***Abstract*—Understanding the reasons why multi-threaded applications do not achieve perfect scaling on modern multicore hardware is challenging. Furthermore, more and more modern programs are written in managed languages, which have extra service threads (e.g., to perform memory management), which may retard scalability and complicate performance analysis. In this paper, we extend speedup stacks, a previously-presented visualization tool to analyze multi-threaded program scalability, to managed applications. Speedup stacks are comprehensive bar graphs that break down an application's execution to explain the main causes of sublinear speedup, i.e., when some threads are not allowing the application to progress, and thus increasing the execution time.**

We not only expand speedup stacks to analyze how the managed language's service threads affect overall scalability, but also implement speedup stacks while running on native hardware. We monitor the application and service threads' scheduling behavior using light-weight OS kernel modules, incurring under 1% overhead running unmodified Java benchmarks. We add two performance delimiters targeting managed applications: garbage collection and main initialization activities. We analyze the scalability limitations of these benchmarks and the impact of using both a stop-the-world and a concurrent garbage collector with speedup stacks. Our visualization tool facilitates the identification of scalability bottlenecks both between application threads and of service threads, pointing developers to whether optimization should be focused on the language runtime or the application. Speedup stacks provide better program understanding for both program and system designers, which can help optimize multicore processor performance.

I. Introduction

Analyzing the performance of multi-threaded applications on today's multicore hardware is challenging. A software developer needs analysis tools to identify the scalability bottlenecks; likewise, computer architects need analysis tools to understand the behavioral characteristics of workloads to design and optimize future computing systems. 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 between threads in multi-threaded applications is complex: some threads perform sequentially for a period of time, others are stalled with no work to do, synchronization behavior makes some threads wait on locks or barriers, and threads can interfere with each other in their use of shared resources, such as the memory subsystem. Many papers have demonstrated the inability of multi-threaded programs to scale well, but studying the root causes of scalability bottlenecks is challenging. Analyzing managed languages is even more challenging, because in addition to the application threads, there are many service threads. Because the application runs on top of a managed runtime environment, additional threads exist to perform dynamic compilation, profiling, and automatic memory management. Teasing apart their effect on the application's performance is also an important part of understanding the scalability of modern multi-threaded programs.

[*]This work was done while at Ghent University.

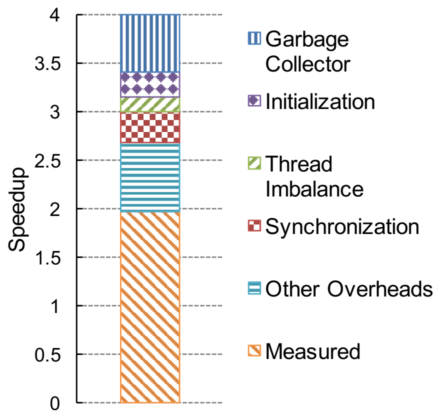

Fig. 1. A speedup stack for the lusearch DaCapo benchmark with 4 application threads and one garbage collection thread running on Jikes RVM.

We build on previous work that introduced the *speedup stack* [1], a visualization tool to quantify components that limit scalability in multi-threaded applications. Speedup stacks give insight to programmers on why their multi-threaded program's speedup, over its single-threaded version, is not actually proportional to the number of cores or threads. It breaks the ideal speedup up into the actual speedup achieved and the contributions of various performance delimiters.

In this paper, we extend the speedup stack to *analyze multi-threaded managed language application scalability on native multicore hardware*. Our performance delimiters include some used in the original speedup stacks: thread imbalance, synchronization, and other overheads (including hardware interference); we also integrate two new components specific to managed applications: automatic memory management (or garbage collection) and the main initialization thread, which performs managed runtime setup and shutdown activities. By integrating new performance delimiters, we can provide a more fine-grained and accurate performance characterization of managed programs, including service threads' impact on scalability. Moreover, while previous work [1] required hardware modifications, in this work we instead use *light-weight OS modules* to compute speedup stacks at negligible overhead on existing multicore hardware.

Our new speedup stacks are a powerful tool to guide optimization of managed applications. Figure 1 shows an example speedup stack for a 4-threaded application, lusearch. The achieved (or measured) speedup of multi-threaded lusearch versus single-threaded execution is only 2 (the bottom box). The components that inhibit scalability the most, i.e., the largest performance delimiters, in relation to an ideal speedup of 4, are garbage collection (GC) and other overheads, which mainly consist of hardware interference (shared cache, bus, main memory, etc.). Other components that limit scalability are synchronization, the initialization of the runtime, and thread imbalance. This visualization tool guides programmers to the component(s) that is/are limiting scalability, whether the bottleneck is in the language runtime or the application, which when fixed will boost application performance. In this example, we would recommend running with more than one collector thread, which would likely reduce the GC component and improve overall performance. Because this benchmark also has a large other-overhead portion, the programmer might look closely at how threads are sharing data.

To illustrate the practical use of speedups stacks, we present an analysis of multi-threaded Java benchmarks with both a stop-the-world and a concurrent garbage collector on Jikes Research Virtual Machine (RVM) [2]. Our analysis reveals that the concurrent collector limits scalability more than a stop-the-world collector using a small heap size, but has better scalability with a larger, less-constrained heap size.

The overarching contribution of this paper is an intuitive, powerful visualization tool for analyzing the scalability of ubiquitous managed language applications running on native modern multicore systems. Speedup stacks point software developers to focus either on improving the parallelization of the language runtime, or on the parallel activities between application threads, to improve overall performance. This tool is important for achieving a better understanding of modern workloads on current multicore machines.

II. Speedup Stacks: Background

This paper presents speedup stacks that comprehensively visualize what retards performance at both the application and the language-runtime level, or the causes of imperfect scaling for managed applications. We build a visualization of what limits scalability for managed applications running on native hardware, using kernel modules to measure what is causing threads to pause. Before introducing our new speedup stacks, we first give background on previous work that introduced the concept of a speedup stack.

Speedup stacks [1] compare the achieved speedup of a multi-threaded application to the ideal speedup and attribute the gap between them to different possible performance delimiters. The total bar in a speedup stack has height N, which is the number of cores or threads. The actual speedup of a multi-threaded application (over a single-threaded version) was originally marked as the *base* component of the bar. The rest of the bar is broken up into the causes of sublinear speedup, such as interference in the memory subsystem, synchronization overhead, work imbalance, etc. Each of these components represents a performance deficiency, and their relative contributions in the speedup stack provide intuition as to what to optimize for the largest improvement in performance.

T_s is defined as the execution time of the single-threaded program. The execution time of the same program during multi-threaded execution, T_p, will (most likely) be shorter. The total execution time is identical for all threads. Speedup is then defined as the single-threaded execution time divided by the multi-threaded execution time:

$$S = \frac{T_s}{T_p}. \tag{1}$$

The idealized multi-threaded execution time, assuming perfect parallelization, equals T_s/N with N as the number of threads or cores. The idealized time, T_s/N, is often not achieved in practice, hence multi-threaded execution time is typically larger. Or in other words, the single-threaded execution time is usually smaller than the sum of the execution times of all threads, because the threads have some overhead (synchronization, work imbalance, etc.). Formally,

$$T_s = \sum_{i}^{N} \left(T_p - \sum_{j} O_{ij} \right) \tag{2}$$

with O_{ij} the overhead caused by component j for thread i. The original work included positive memory interference in this equation, but we omit that here because it is not relevant to our new speedup stacks (as explained in Section III).

By dividing this equation by T_p, we get speedup:

$$\frac{T_s}{T_p} = S = N - \frac{\sum_{i}^{N} \sum_{j} O_{ij}}{T_p}. \tag{3}$$

This formula immediately leads to a speedup stack by showing the different overhead components j, aggregated over all threads, in a stacked bar. The intuition behind a speedup stack is that it shows the reasons for sublinear scaling and hints towards the expected performance benefit from reducing a specific scaling bottleneck, i.e., the speedup gain if this component is reduced to zero. This can guide programmers to tackle those bottlenecks that have the largest impact on multi-threaded application performance.

The original speedup stacks [1] include the following components: work imbalance, spinning, yielding, and positive and negative last-level cache and memory interference. Constructing speedup stacks previously relied on dedicated hardware support built on top of a per-thread cycle-accounting architecture [3]. Because existing hardware does not provide such support, we cannot compute these speedup stacks on existing hardware. To create the original speedup stacks, they ran the multi-threaded application and then estimated the single-threaded performance from that execution.

III. Speedup Stacks for Managed Applications

We extend the original speedup stacks in this paper to analyze the scalability of multi-threaded managed language applications, whereas previous work had only applied speedup stacks to native applications. We also use a light-weight infrastructure to measure thread behavior on existing hardware (see Section IV). We newly incorporate the managed runtime service threads that affect scalability into the speedup stack, including the sequential parts that do **initialization** and clean-up, for which our Java virtual machine uses a service thread called the MainThread, and the automatic memory manager or **garbage collector** (GC), which interferes with application progress. Thus, if these components of the speedup stack are large, this gives hints to the managed runtime developers to better parallelize the garbage collector or minimize the runtime's initialization and shutdown activities. Additionally, the following performance delimiters are also included in our new speedup stacks: **synchronization activities** between threads, or when the operating system yields the processor to another thread because it is blocked on a barrier or lock; **thread imbalance**, or when certain threads have exited and other threads are still running; and **other overhead** components which can include parallelization overhead and shared hardware resource interference, such as in caches and memory. Below, we build up equations for quantifying these performance deficiencies, or delimiter components, in the bar graph.

Because we measure the performance delimiters of our speedup stacks in system software, we estimate the cache and memory interference component that the original work could precisely measure with hardware. Our tool does not currently take into account busy waiting in spin loops. However, 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 measurable impact in practice.

To build a speedup stack, we first run the single-threaded version of the application to have a baseline. We then profile a multi-threaded execution of the application, computing the different overhead components for each thread. We measure the actions that cause some threads not to run, which will result in less-than-perfect speedup. We denote the actual speedup achieved by the multi-threaded application as the *measured* component in our speedup stack.

In the next sections we discuss the different overhead components that O_{ij} is comprised of, and how they are integrated in Formula 3. We consider garbage collection, the sequential initialization of the runtime, synchronization, thread imbalance, and additional overheads (including hardware interference) as components in our speedup stacks.

A. Garbage Collection

A new contribution of this work is that we consider the overhead of the garbage collector in our construction of speedup stacks. Garbage collection is an integral component of many managed languages. Programmers benefit from the fact that the managed runtime environment automatically manages

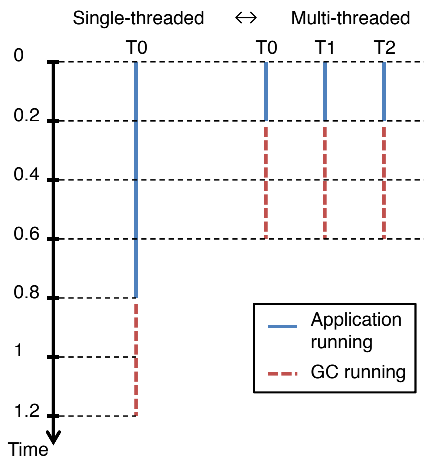

Fig. 2. Illustration of a single-threaded and multi-threaded managed language application execution, with a stop-the-world garbage collector.

and collects memory. However, this benefit comes with a cost; garbage collection does incur some space and time overhead. Previous work estimates that a well-performing stop-the-world generational garbage collector takes on average 10% of an application's execution time [4]. With this collector, the application is stopped while the collector traces the heap and reclaims memory. There are also collectors that reclaim memory while running concurrently with the application; however, they commonly require the application to stop so that the collector can identify a consistent set of roots to trace from (including stack variables, statics and globals), and, after tracing, to finally reclaim memory back to a free list. When GC threads are run concurrently with the application, they are not directly limiting scalability (however, there may be other effects such as hardware interference in shared resources). Thus in our speedup stacks, we only take into account when GC pauses the application because it is then directly affecting the application's ability to make progress.

Figure 2 shows the single-threaded version of a managed language program on the left, with $T_s = 1.2$. Note that overall execution time includes both application and garbage collection phases; the time during which the stop-the-world GC is running (and the application is stopped) is shown with a dashed line. The execution time of running with four application threads, T_p, is 0.6 on the right side of Figure 2. The ideal speedup, T_s/N, in Figure 2's example would be $1.2/4 = 0.3$. Note that the number of GC threads does not have to equal the number of application threads.

In the previous definition of speedup stacks [1], the garbage collection component would have been a part of the yielding component. Application threads have to yield to let GC run, and are thus scheduled out by the OS during that time. This version of speedup stacks more precisely divides up the yield component into GC scalability and a synchronization component that represents only synchronization between application threads, as explained below.

Because the application threads are halted during their execution in order to perform garbage collection, we account for these pauses as an overhead component that can possibly limit speedup. Because the actual speedup of the multi-threaded application over the single-threaded version already takes garbage collection time into account, our overhead component needs to only consider the scalability of garbage collection. We thus compare the amount of time spent on GC in the multi-threaded execution, multiplied by the number of threads, to the time for GC in the single-threaded execution. Integrating garbage collection into Formula 3 leads to:

$$S = N - \frac{N \times T_{GC,MT} - T_{GC,ST}}{T_p} - \sum_{i}^{N} \frac{\sum_j O1_{ij}^r}{T_p} \tag{4}$$

where $T_{GC,ST}$ and $T_{GC,MT}$ is the time needed to do garbage collection for the single-threaded and multi-threaded execution, respectively, and $O1_{ij}^r$ are the remaining overhead components j for thread i. $T_{GC,MT}$ is the same for all threads, i.e., we assume all GC threads are active from the start of a collection to the end. We subtract $T_{GC,ST}$ from the garbage collection overhead because the single-threaded execution also has a garbage collection component, and the speedup is measured over the whole program.

It is clear that if the stop-the-world phase of garbage collection is perfectly scalable (i.e, $T_{GC,MT} = \frac{T_{GC,ST}}{N}$), this overhead component of speedup stacks would reduce to zero. Thus this performance delimiter suggests the effect of limited GC scalability on achieved program speedup.

B. Managed Runtime Initialization

We consider other service threads that stop the managed language application's progress and thus limit its scalability. In Jikes RVM, there is a service thread called the MainThread that initializes the Java virtual machine (JVM), does initial compilation, spawns the application threads, and later performs shutdown activities. Because we explore multi-threaded applications, the MainThread limits scalability because its work is not parallelized. The speedup of a program is limited by the amount of sequential execution in the application [5]. We integrate this component into our speedup formula, which is very similar to the garbage collection component, because application threads are, in essence, stopped while the MainThread is running:

$$\begin{aligned} S = N &- \frac{N \times T_{GC,MT} - T_{GC,ST}}{T_p} \\ &- \frac{N \times T_{seq,MT} - T_{seq,ST}}{T_p} - \sum_{i}^{N} \frac{\sum_j O2_{ij}^r}{T_p} \end{aligned} \tag{5}$$

where $T_{seq,MT}$ is the execution time of the MainThread during multi-threaded execution, and $T_{seq,ST}$ during single-threaded execution.

This initialization component of the speedup stack thus estimates the scalability of the sequential MainThread and its impact on program speedup. Our JVM does not have a parallelized version of these initialization activities; however, if these activities were perfectly parallelizable, this component could be reduced to zero, improving overall program performance and scalability.

C. Synchronization

We now consider synchronization between application threads, or between garbage collection threads, as opposed to interactions between service and application threads. Threads synchronize with each other when working on shared data or because they have to wait on each other. This synchronization leads to an extended execution time — or in other words, time when all threads are not concurrently running — and therefore should be accounted for as an overhead component that limits scalability. These components were included in the original speedup stacks, but we measure them differently because we do not require hardware support. In our implementation, we intercept futex system calls that cause a thread to wait. We thus compute this overhead as the sum of all times a thread is waiting due to synchronization, summed over threads. Integrating this into Formula 5 leads to:

$$\begin{aligned} S = N &- \frac{N \times T_{GC,MT} - T_{GC,ST}}{T_p} \\ &- \frac{N \times T_{seq,MT} - T_{seq,ST}}{T_p} \\ &- \frac{\sum_i^N Sync_i}{T_p} - \sum_{i}^{N} \frac{\sum_j O3_{ij}^r}{T_p} \end{aligned} \tag{6}$$

where $Sync_i$ is the waiting time due to synchronization for thread i. Thus, if all threads' waiting time due to synchronization with other threads would go to zero, this speedup stack component would disappear, thus resulting in a higher achieved speedup.

D. Thread Imbalance

Thread imbalance happens when one or a few application threads need (substantially) more time to execute than the other threads, which puts a limit on the achieved speedup or scalability of the program. To account for this we measure the waiting time of an application thread inside an exit system call until all application threads finish their execution (see Section IV). Integrating this into the formula leads to:

$$\begin{aligned} S = N &- \frac{N \times T_{GC,MT} - T_{GC,ST}}{T_p} \\ &- \frac{N \times T_{seq,MT} - T_{seq,ST}}{T_p} \\ &- \frac{\sum_i^N Sync_i}{T_p} - \frac{\sum_i^N Exit_i}{T_p} - \sum_{i}^{N} \frac{\sum_j O4_{ij}^r}{T_p} \end{aligned} \tag{7}$$

where $Exit_i$ is the waiting time of thread i after exiting while other threads are still running. This overhead component in the speedup stack represents the proportion of idealized speedup that could be gained if all application threads exit at the same time, or are well-balanced in their work.

E. Remaining Overhead Components

The speedup S in Formula 7 has an additional component $O4_{ij}^r$. The remaining overhead is due to other factors that limit scalability and performance when moving from single-threaded to multi-threaded execution on modern hardware. For example, parallelizing a program typically incurs overhead due to additional instructions being executed. Second, resource sharing on modern multicore processors leads to interference between (all service and application) threads [3], which can manifest with cache coherence overhead, cache misses, and the overhead of going to off-chip memory [1]. We include a final component in our speedup stacks that we call *other overhead*, which estimates $O4_{ij}^r$, accounting for hardware interference and parallelization overheads and showing their impact on speedup. Though we cannot precisely measure this overhead in system software, we present this component as the difference between the measured speedup and the ideal speedup, minus all of the other components. Because we have the advantage of running unmodified managed applications on current hardware, we can quickly and accurately gather statistics on real program behavior. Thus, if this other-overhead component of the speedup stack is large, we recommend that users use performance counters to analyze if the problem is extra instructions, or shared cache or memory traffic. Alternatively, programmers could use tools such as ScaAnalyzer [6] to identify scalability bottlenecks due to contention in caches or memory. Then programmers, runtime engine developers, or architects know what to focus their efforts on to optimize modern multi-threaded applications for improved scalability.

IV. Designing the Application Profiling Tool

We measure the inputs to calculate speedup stacks for applications running on real hardware through operating system (OS) support using light-weight Linux kernel modules. There are several advantages to using kernel modules: the programs require no modifications or re-compilation; the kernel does not need to be re-compiled because the modules are loaded dynamically; we can use a nanosecond-resolution timer; and despite having very limited overhead (on average 0.78% for our benchmarks), our tool continuously monitors all threads' scheduling activities without loss of information.

To measure the values needed to construct speedup stacks, we need to detect/have:

1) The number and IDs of active threads.
2) Events that cause a thread to activate and deactivate.
3) A timer to measure overhead.

The operating system naturally provides what we need. We built a tool that gathers the necessary information to construct speedup stacks 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., communicating the ID of the process that should be monitored) is done using writes and reads in the /proc directory. Kernel modules intercept system calls that perform thread creation and destruction (sys_exit), that schedule threads in and out, and that do synchronization with futex (which implements thread yielding).

Our modules have a counter to accumulate the execution time of (a) the MainThread, (b) garbage collection threads, and (c) application threads. Furthermore, the tool also keeps track of the per-thread waiting time due to futex system calls (synchronization) and exit system calls (thread imbalance) in two additional counters. In our JVM, application threads are stopped to start a stop-the-world garbage collection phase using futex system calls. In order to get the synchronization overhead time without the GC time, we therefore subtract the collector's execution time from the futex waiting time in a post-processing step.

Our tool keeps track of the IDs of active threads and timestamps of interval boundaries, i.e., when any thread is scheduled in or out. Upon detection of an interval boundary, 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 the threads that were running in the past interval. If a thread is halted, it records the corresponding system call (futex or exit) and the current time, and if a thread is woken, the waiting time is added to the corresponding counter. Subsequently, the module changes the set of running threads according to the interval boundary information, and records the current timestamp as the beginning of the next interval. When the OS receives a signal from software, the counters are written out, and this information is read by a script that generates the graphs.

To gather our results, we read out our cumulative thread statistics at the end of the program run, and thus our speedup stacks represent the entire application execution. However, our tool can be given a signal at any time to output, and optionally reset, thread counters. Thus, speedup stacks can be constructed at any time during the program run, and can be used to analyze particular phases or sections of code for scalability bottlenecks.

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 speedup stacks. When a thread performs I/O, the OS schedules that thread out. We choose not to track I/O system calls separately in our kernel modules because most I/O behavior is already accounted for as inactive and we found this component to be very small in our setup. 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. We thus also implicitly support SMT environments. Our modules can be easily updated to also account for I/O overhead and over-subscription overhead (i.e., more threads than hardware contexts) if needed. These overheads can be visualized in a speedup stack, similar to synchronization and imbalance overheads.

While other OS activities besides system calls could affect the execution time of a particular run of an application, such as page faults, the OS would schedule those threads out and thus our system would count that time as inactive. However, the user is encouraged to repeat runs multiple times and use the most consistent (non-outlier) runs for comparison with the corresponding single or multi-threaded executions.

V. Experimental Methodology

We perform experiments on unmodified applications running on real hardware to demonstrate the usefulness of our analysis tool. We analyze both application and service thread performance and scalability using speedup stacks.

We evaluate four multi-threaded Java benchmarks from the DaCapo 2009 benchmark suite [7]. Although eclipse spawns multiple threads, we found that only one thread is running for the majority of the execution, so we categorize it as a single-threaded application and exclude it from this study. We also leave out avrora and pseudoJBB from our analysis because it is impossible to change these benchmarks' thread count without changing the input set. Thus, we analyze lusearch, which incurs 1.15% measurement overhead with our tool, pmd (0.53%), sunflow (1.04%) and xalan (0.40%). The average overhead from the kernel module is just 0.78% across our benchmarks because the kernel modules only have to do small calculations when threads are scheduled in or out.

For our experiments, we vary the number of application threads (1, 2, 4 and 8), but set the number of garbage collector threads to two, following recommendations that Jikes performs best with this number [8]. We experiment with different heap sizes (as multiples from the minimum size that each benchmark can run with on the stop-the-world collector). We run the benchmarks for 15 iterations, and present results from the 13th iteration to show stable behavior.

We perform 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 use only one socket in our experiments with HyperThreading enabled, which leads to 16 available hardware contexts. This setup avoids data traversing socket boundaries, which can have a large impact on performance [9]. 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 3.1.2 [2]. We use the default best-performing garbage collector (GC), the stop-the-world parallel generational Immix collector [10]. We also perform experiments with Jikes' concurrent collector. Jikes RVM uses a mark-sweep snapshot-at-the-beginning concurrent GC algorithm. The concurrent collector initiates a new collection cycle with a trigger: after a particular percentage of total memory is allocated, a new concurrent collection cycle is triggered. We use Jikes' default trigger value.

Jikes' concurrent collector requires a small pause of the application to first identify a consistent root set, and later to actually free memory. Between these two actions, the collector threads run concurrently with the application threads in order to trace the object graph. The concurrent collector in Jikes spawns different sets of threads to perform the stop-the-world activities and the concurrent activities. If the application is rapidly allocating while the concurrent GC is running such that the garbage collector cannot free up memory fast enough to accommodate new memory requests, the GC forces the application threads to stop and finishes collection in a stop-the-world mode [11]. As previously explained, the GC scalability component of speedup stacks represents only the stop-the-world portion of the concurrent collector, as the concurrent GC activity does not inherently limit application thread scalability.

VI. Analyzing Scaling Behavior with Stop-the-World Garbage Collection

For understanding how managed applications scale and the relative contributions of various multi-threaded performance deficiencies, we present speedup stacks as explained in Section III. In this section, we perform experiments with the stop-the-world garbage collector using two threads and a heap size of 2× the minimum.

Figure 3 shows speedup stacks for our benchmarks with 2, 4 and 8 application threads as compared to their single-threaded versions. The total height of the bar is equal to the ideal speedup, or the number of application threads. The orange component at the bottom of each stack shows the measured speedup between a single-threaded and multi-threaded execution and the colored boxes on top of it show the various scalability delimiters and their impact on speedup.

From the stacks we can see that most of our applications do not scale very well: no 8-threaded application achieves a speedup much more than 4. Applications sunflow and xalan show comparable measured speedup results, but the reason why their speedup is limited is different. While sunflow mostly suffers from other parallelization overheads, xalan mostly suffers from the limited garbage collector scalability. Pmd is the application that scales the worst because of one thread running longer than the others. Pmd is limited by one large input file [8], that if broken up, could significantly reduce the thread imbalance component. The reason why lusearch does not scale well is a combination of all components in the speedup stack. The GC's limited scalability is the main reason, together with other overheads. For lusearch and pmd, if the language runtime could improve both the parallelization of the garbage collector and of the MainThread's initialization activities, multi-threaded speedup could improve significantly.

To understand the other-overheads component better, we present hardware performance counter data in Figure 4, normalized to the single application thread execution. The performance counters reveal that the total number of instructions stays almost constant for all benchmarks when increasing the number of application threads, meaning that speedup is not limited because of additional instructions, or parallelization overhead. This suggests that the other overhead is mostly due to hardware interference in the memory subsystem.

For sunflow, with the largest other-overheads component, the number of last-level cache (LLC) loads goes up steeply when going to 4 and 8 application threads, which is expected

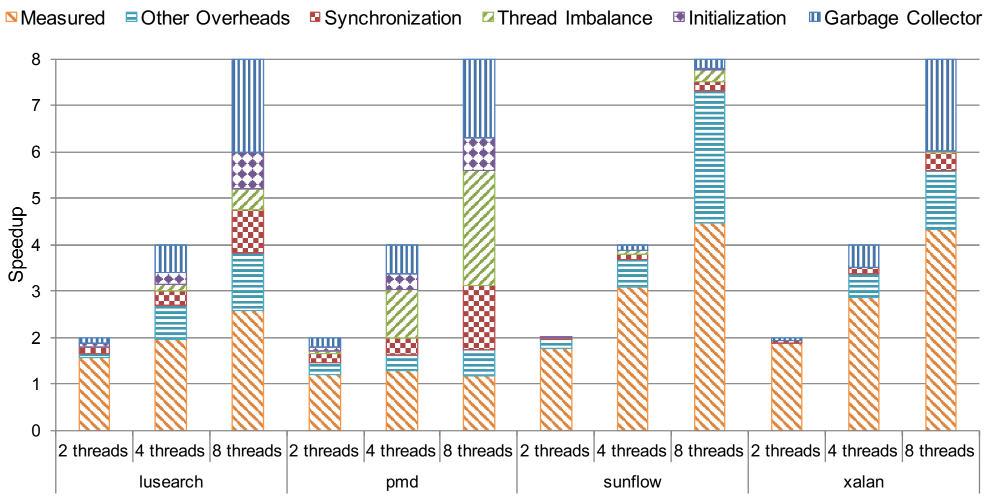

Fig. 3. Speedup stacks for all applications with a stop-the-world garbage collector (2×min heap size).

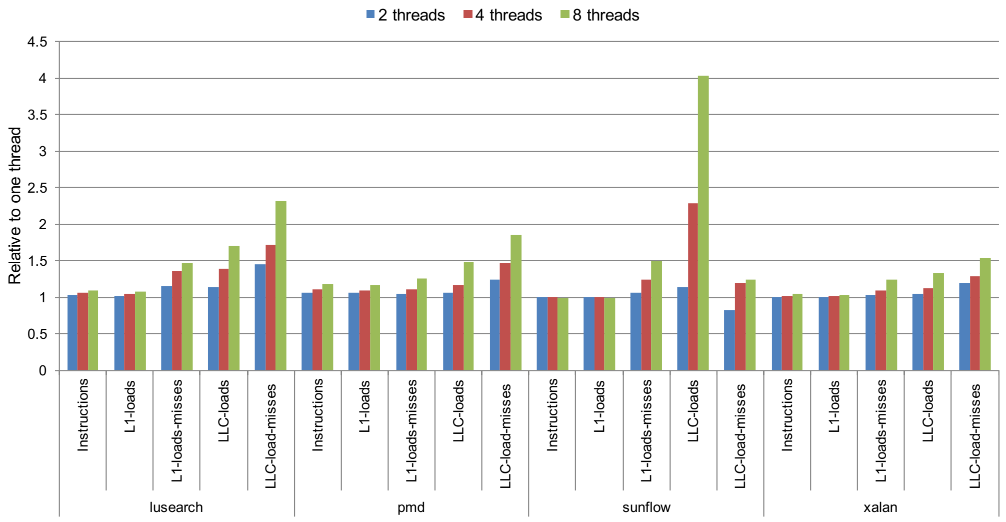

Fig. 4. Data from hardware performance counters running with a stop-the-world garbage collector (2×min heap size), normalized to one application thread.

as the GC and cache coherence activity increase. However, the number of LLC load misses does not increase significantly, and 2 application threads experience fewer misses than one application thread. This suggests that sunflow has a lot of shared data between threads. For other benchmarks, other-overhead translates into a combination of an increased number of LLC loads and LLC load misses, particularly for lusearch. Lusearch allocates a lot of memory at a high rate [9], and thus the extra GC activity could contribute to this increased LLC traffic and its large synchronization component.

VII. Analyzing Scaling Behavior with Concurrent Garbage Collection

In this section, we explore the scalability of multi-threaded Java applications running with a concurrent collector using speedup stacks. As previously explained in Section III-A, the GC component of speedup stacks only measures the limited scalability of the stop-the-world phases of garbage collection, because they directly inhibit the progress of the application.

Figure 5 shows speedup stacks for the same applications as in Figure 3, but now running with a concurrent garbage collector. In this experiment we use the same heap sizes for the applications as in the previous section. The speedup stacks reveal that for all benchmarks the impact of GC on speedup has become larger compared to using a stop-the-world garbage collector. In fact, for three benchmarks, the measured speedup is reduced when going from 4 to 8 application threads. The application that suffers the most from the garbage collector's limited scalability is lusearch, because of its high allocation rate [9]. Because of the excessive allocation, the concurrent collector is not able to free up memory fast enough [11], and transitions to stop-the-world mode, which retards the application's scalability. We conclude that the concurrent garbage collector in Jikes RVM does not scale well, especially with

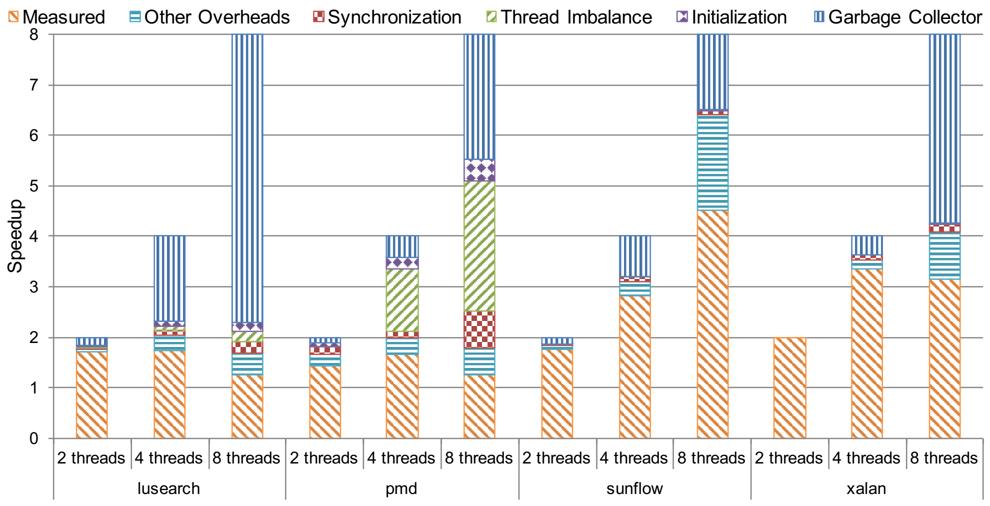

Fig. 5. Speedup stacks for all applications with a concurrent garbage collector (2×min heap size).

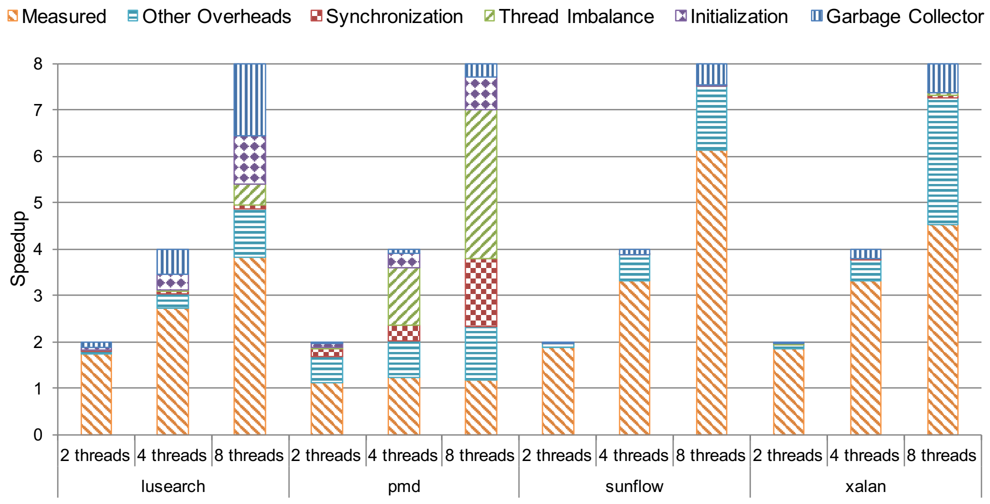

Fig. 6. Speedup stacks for all applications with a concurrent garbage collector (10×min heap size).

small heap sizes, and if fixed by developers, could improve application scalability significantly.

To explore the scalability of the concurrent garbage collector when the heap size is not constrained, we also ran our applications with a larger heap size (10× the minimum heap size used with the stop-the-world collector). The speedup stacks are shown in Figure 6. We see that the impact of GC's scalability on speedup is significantly reduced and the measured speedup is improved, compared to Figure 5 (except for pmd that suffers from a large thread imbalance). For sunflow and xalan, the main speedup delimiter now is other overhead, as expected because of the many threads concurrently running and interfering with each other, while for lusearch it is a combination of different components.

For comparison, we also performed an experiment using the stop-the-world collector and the larger heap size, and found no noticeable difference between the generated speedup stacks for the larger and smaller heap sizes. However, the measured component of the speedup stacks is slightly higher when using the concurrent (versus stop-the-world) collector at the larger heap size, because the application threads are not stopped from making progress as much because of GC. Furthermore, all benchmarks (except pmd) seem to have a reduced synchronization component in the speedup stacks with a concurrent collector, probably due to the stop-the-world collector running more frequently, and thus issuing more futex operations. Also, pmd and xalan have much reduced garbage collector scalability components when using Jikes' concurrent GC, while sunflow has a reduced other-overheads component.

For explaining the other-overhead component, particularly the hardware interference, we measured hardware performance counters when running with a concurrent collector and a large heap (omitted due to space constraints). Lusearch suffers from an increased number of LLC loads as the application thread count increases, and LLC load misses increase from 2 to 4 application threads, but go down for 8 threads. This suggests that with a larger number of application threads, there is more data sharing (between application and GC threads) in the LLC. This was not the case when using a stop-the-world collector (see Figure 4), which can disrupt the LLC and incur more LLC load misses for the application, especially at high thread counts.

Sunflow has an increasing number of L1 cache loads, misses and LLC loads, which are the main causes of the hardware interference. This behavior is different than when sunflow runs with a stop-the-world garbage collector, which does not have an increasing number of L1 loads as the application thread count increases, but has a larger increase in LLC loads. With the concurrent GC, the increase in L1 loads are due to the garbage collector accessing the L1 cache more often at the same time as the application. However, the application does not suffer much because the data is kept in the upper levels of cache. Xalan and pmd show similar behavior with a concurrent collector as with a stop-the-world collector: an increasing number of L1 load misses that result in more LLC load accesses and LLC load misses.

We have shown that speedup stacks facilitate the visualization of performance and scalability bottlenecks in multi-threaded managed language applications. They reveal the impact of the limited scalability of the garbage collector, initialization activities, synchronization activities between application threads, imbalance of application threads, and the effect of other overheads – particularly hardware interference in the memory subsystem – which can also be explained by performance counter data. Thus, the speedup stack directly reveals whether time should be spent on improving the parallelization of the managed runtime, looking at the application threads' interactions, or trying to minimize shared memory system interference.

VIII. Related Work

We now describe related work in performance visualization and Java parallelism analysis.

A. *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], Rogue Wave/Acumem ThreadSpotter[1] (which targets memory problems) and PGPROF from the Portland Group [14] 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. These tools provide fairly detailed analysis at a fine granularity in small functions and individual lines of code. They do not automatically analyze managed language service threads separately from application threads, and they do not give a broader view on the scalability of the application or what to focus on to improve it.

Recent work focused on minimizing parallel overhead by enabling the analysis of very small code regions, such as critical sections [15], [16]. Other related work [17] 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 [18]. All of these approaches strive at providing fine-grained performance insight. However, none of these approaches provide a simple and intuitive visualization and understanding of gross performance scalability bottlenecks in managed multi-threaded applications, as our work does and which is needed by software developers to guide optimization.

Recent work presented criticality stacks [19], and then bottle graphs [8], which display per-thread contributions to total program performance and parallelism. This work points out thread imbalances, but it does not suggest how much total application performance could be improved by removing a particular scalability bottleneck.

IBM WAIT[2] [20] is a visualization tool for diagnosing performance and scalability bottlenecks in Java programs, particularly server workloads. It uses a light-weight profiler that regularly samples information about each thread. WAIT 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 speedup stacks because we use OS modules. WAIT also collects a snapshot of information only at specific program points, with increasing overhead with finer-granularity sampling. In contrast, speedup stacks contain more information with lower overhead. Our OS modules are continually monitoring every thread status change, and aggregating our metrics at all times.

B. *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. [21] analyze the 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 [22]. They did follow-on work to optimize scalability at large thread-counts for the parallel stop-the-world garbage collector in OpenJDK [23]. Similarly, Chen et al. [24] 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 did not, however, quantify the speedup lost because of the parallel

[1]http://docs.roguewave.com/threadspotter/2011.1/manual/

[2]https://wait.ibm.com/

collector, or analyze how it interacts with the application, as we do in this paper.

IX. Conclusion

This paper extends speedup stacks to visualize the scalability bottlenecks of managed language workloads on native multicore hardware. Speedup stacks have always provided a comprehensive breakdown of the causes of limited scalability in programs, revealing the causes of sublinear scaling. Our new speedup stacks show the relative contributions of not only previously-proposed components: synchronization, thread imbalance, and hardware interference (within other overhead), but also two new performance delimiters specific to managed languages: the garbage collector, and the managed runtime initialization and shut-down activities. Our speedup stacks enable users to immediately see whether the parallelization of the language runtime service threads needs to improve, or if the application code itself needs to be re-written to increase multi-threaded speedup.

To construct speedup stacks on real hardware, we propose a light-weight and low-overhead approach for measuring scheduling behavior using OS kernel modules with no changes required to software or hardware. We present the speedup stacks of several multi-threaded Java benchmarks to explore not only the application's scalability, but also the performance of the JVM's service threads. We reveal several insights that we have gathered about Jikes RVM's garbage collectors; in particular, we find that the concurrent collector scales worse than the stop-the-world collector at small heap sizes, but at larger heap sizes, the opposite is true.

Overall, we demonstrate that speedup stacks are particularly effective at visualizing the performance bottlenecks of multi-threaded managed language applications. Speedup stacks offer programmers and computer architects a comprehensive understanding of modern managed applications running on current multicore hardware, and guide them to the scalability bottlenecks that, when fixed, can optimize performance.

Acknowledgements

We thank the anonymous reviewers for their valuable feedback and suggestions. This work is supported in part by the FWO projects G.0434.16N and G.0144.17N, and ERC grant agreement no. 259295.

References

[1] 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)*, Apr. 2012, pp. 145–155.

[2] 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)*, Nov. 1999, pp. 314–324.

[3] K. Du Bois, S. Eyerman, and L. Eeckhout, "Per-thread cycle accounting in multicore processors," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 4, pp. 1–22, Jan. 2013.

[4] T. Cao, S. M. Blackburn, T. Gao, and K. S. McKinley, "The yin and yang of power and performance for asymmetric hardware and managed software," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2012, pp. 225–236.

[5] G. M. Amdahl, "Validity of the single-processor approach to achieving large-scale computing capabilities," in *Proceedings of the American Federation of Information Processing Societies Conference (AFIPS)*, Sep. 1967, pp. 483–485.

[6] X. Liu and B. Wu, "ScaAnalyzer: A tool to identify memory scalability bottlenecks in parallel programs," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015, pp. 47:1–47:12.

[7] 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 *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, Oct. 2006, pp. 169–190.

[8] K. Du Bois, J. B. Sartor, S. Eyerman, and L. Eeckhout, "Bottle graphs: Visualizing scalability bottlenecks in multi-threaded applications," in *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, Oct. 2013, pp. 355–372.

[9] 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)*, Oct. 2012, pp. 281–296.

[10] S. M. Blackburn and K. S. McKinley, "Immix: A mark-region garbage collector with space efficiency, fast collection, and mutator performance," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Jun. 2008, pp. 22–32.

[11] S. Akram, J. B. Sartor, K. Van Craeynest, W. Heirman, and L. Eeckhout, "Boosting the priority of garbage: Scheduling collection on heterogeneous multicore processors," *ACM Transactions on Architecture and Code Optimization*, vol. 13, no. 1, pp. 4:1–4:25, Mar. 2016.

[12] Intel, "Intel VTune^{TM} Amplifier XE 2013," http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/, 2013.

[13] M. Itzkowitz and Y. Maruyama, "HPC Profiling with the Sun Studio^{TM} Performance Tools," in *Tools for High Performance Computing*. Springer, 2010, pp. 67–93.

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

[15] A. Bhattacharjee and M. Martonosi, "Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, Jun. 2009, pp. 290–301.

[16] J. Demme and S. Sethumadhavan, "Rapid identication of architectural bottlenecks via precise event counting," in *International Symposium on Computer Architecture (ISCA)*, Jun. 2011, pp. 353–364.

[17] M. Kambadur, K. Tang, and M. A. Kim, "Harmony: Collection and Analysis of Parallel Block Vectors," in *International Symposium on Computer Architecture (ISCA)*, Jun. 2012, pp. 452–463.

[18] 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)*, Jun. 2011, pp. 458–469.

[19] 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)*, Jun. 2013, pp. 511–522.

[20] 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)*, Oct. 2010, pp. 739–753.

[21] 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)*, Oct. 2012, pp. 335–354.

[22] L. Gidra, G. Thomas, J. Sopena, and M. Shapiro, "Assessing the Scalability of Garbage Collectors on Many Cores," *ACM SIGOPS: Operating Systems Review*, vol. 45, no. 3, Dec. 2011.

[23] ——, "A study of the scalability of stop-the-world garbage collectors on multicore," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar. 2013, pp. 229–240.

[24] K. Y. Chen, J. M. Chang, and T. W. Hou, "Multithreading in Java: Performance and scalability on multicore systems," *IEEE Transactions on Computers*, vol. 60, no. 11, pp. 1521–1534, Nov. 2011.