A Black-box Approach to Understanding Concurrency in DaCapo

Tomas Kalibera Matthew Mole Richard Jones Jan Vitek

University of Kent, Canterbury Purdue University

Abstract

Increasing levels of hardware parallelism are one of the main challenges for programmers and implementers of managed runtimes. Any concurrency or scalability improvements must be evaluated experimentally. However, application benchmarks available today may not reflect the highly concurrent applications we anticipate in the future. They may also behave in ways that VM developers do not expect. We provide a set of platform independent concurrency-related metrics and an in-depth observational study of current state of the art benchmarks, discovering how concurrent they really are, how they scale the work and how they synchronise and communicate via shared memory.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features — Concurrent Programming Structures

Keywords Benchmarks, DaCapo, concurrency, scalability

1. Introduction

In the face of technological and physical limitations preventing further clock speed increases, hardware designers have turned to providing processors with increasing numbers of cores — Intel has 48-core processors, Tilera 64-core processors and Azul ships 54-core × 16 processor systems. All these systems provide shared memory and varying degrees of coherency. To keep delivering ever more powerful applications, programmers must turn their attention to making good use of those cores. This means not only parallelising their algorithms, but also avoiding timing accidents due to non-local memory accesses or cache coherency traffic. Managed language runtimes, or virtual machines (VM), lift some of this burden. High-level concurrency libraries and runtime services, such as just-in-time compilation and garbage collection (GC), make it much easier to write code that runs efficiently on a variety of platforms. To achieve this, VMs are becoming increasingly aware of architectural issues that affect scalability. Garbage collectors may be NUMA-aware when allocating memory, take advantage of multiple cores to speed up memory reclamation and keep thread-local objects together to reduce cache coherency traffic.

If researchers are to develop VMs that meet the challenges set by highly parallel hardware architectures, they need to know what multi-threaded programs really do. VM development is often motivated by the performance of a given system on some suite of well known applications. Such benchmarks influence development. They may "accelerate, retard or misdirect energy and innovation" [6]. The questions for research on multi- and many-core hardware are how new designs and implementations scale with increasing numbers of cores. Modern architectures have complex performance models. When these are coupled with dynamic code generation and memory management techniques that move data on the fly, it becomes very challenging to predict how a particular program will perform and how it will scale. We argue that a necessary starting point is for researchers to understand how the benchmarks they use stress the various components of a platform (e.g. shared memory access, cache coherency, synchronisation, GC).

We aim to provide insights on the suitability of benchmarks, particularly those written in Java, for scalability studies on parallel hardware. We distinguish the terms *concurrency* and *parallelism*. Concurrency is a software engineering tool to model real-world systems that has long been used as a programming model, even on sequential hardware [23]. Concurrent programs can offer better responsiveness [7] by leveraging the scheduler. Concurrency is also a mechanism for achieving improved throughput on parallel hardware. We are interested in the latter. In Java, threads typically communicate via shared memory. The impact on throughput and scalability of communication through shared memory is important for both application programmers and VM designers. Communication may lead to contention, both for software-level constructs such as locks, and hardware artefacts such

OOPSLA'12 October 19–26, 2012, Tuscon, Arizona, USA.
Copyright © 2012 ACM 978-1-4503-1561-6/12/10. . . $10.00

as cache-lines. VM implementors work hard to reduce the costs incurred by communication, but in order to design new optimisations they need benchmark suites that provide representative models of applications 'in the wild', and insights into how these benchmarks use shared memory.

An underlying principle of this study is *platform independence*: to explore how benchmarks may behave on *any* hardware platform and *any* language runtime rather than on just those *currently* at hand. We use a combination of observational techniques based on program instrumentation and on measurements of platform independent metrics. We measure, for instance, the frequency at which different threads access a shared object (application behaviour) rather than measuring hardware performance counters (the consequences of an application's behaviour given a particular scheduling on a particular platform).

We focus on Java-level memory operations that are specified in the source code. These include reads and writes to fields, monitor acquisitions/releases and object allocations. Memory operations may stress concurrency mechanisms in two ways. First, objects may be accessed by multiple threads. Second, these may access different objects that happen to collide in the memory hierarchy, a property known as false sharing. Both can be the source of non-local memory access and/or cache coherency traffic. False sharing may have a significant impact on performance [18] but, since we are interested in the platform independent behaviour of applications, we do not focus on it. The stress induced by shared access may be trivial unless an object is used by multiple threads repeatedly within some time window. The cost of such regular accesses depends on the number of changes of ownership of the object. This can happen through modifications, typically writes and monitor acquisitions. We investigate the extent to which more than one thread accesses a shared object and patterns of these accesses.

Scalability is a core metric for evaluating the performance of parallel systems. To measure scalability, one would run a parallel benchmark several times, on a varying number of cores. One expects a *scalable* benchmark to divide the work equally between the available cores, giving a nearly linear speed-up. We investigate whether benchmarks that are used to study parallel systems are indeed scalable. We ask how many threads in these benchmarks take part in concurrency-stressing memory operations.

We offer a detailed observational study of the DaCapo Java workloads [6, 20]. Our goal is to provide platform-independent insights into (a) how concurrent these workloads really are, (b) how to characterize the patterns of concurrent behaviour exhibited by the individual programs, (c) the extent to which they can be used for testing the scalability of VMs, and (d) the stress they put on memory and particularly shared memory. The source code of our implementations is at `www.cs.kent.ac.uk/projects/gc/dacapo`.

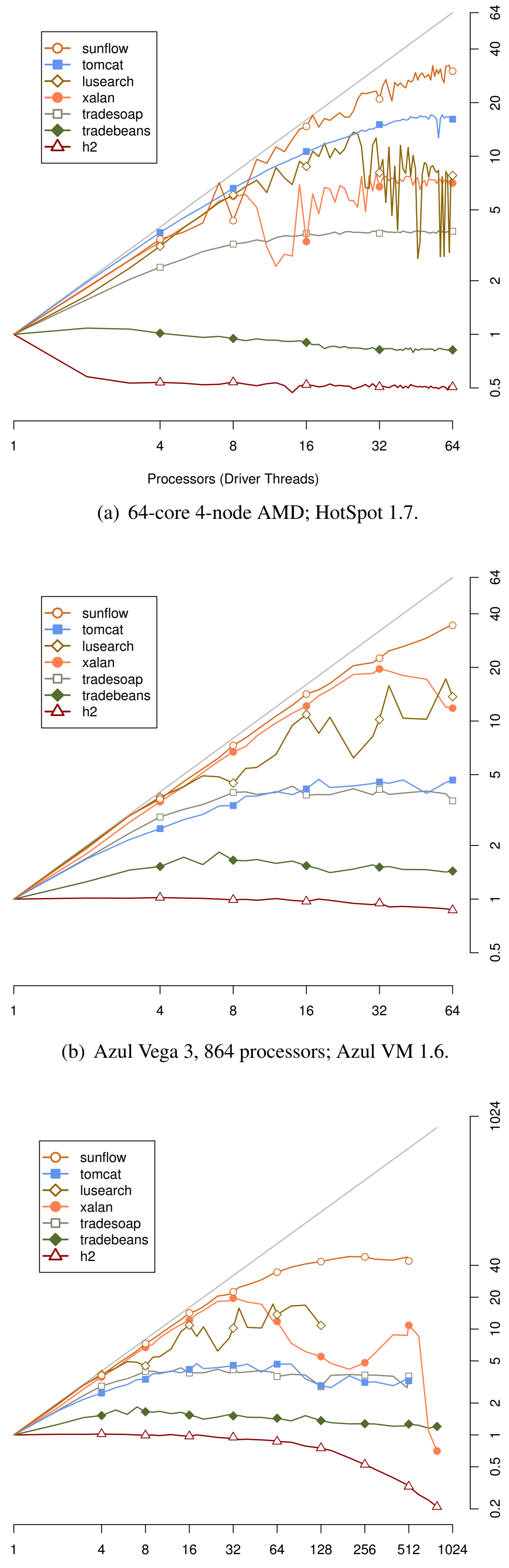
2. Related Work

Dufour et al. proposed a range of dynamic metrics to characterise Java programs in terms of memory use, concurrency, and synchronisation [12]. They measured concurrency by *thread density*, the maximum number of threads ever runnable at the same time, and by the amount of code executed while a given number of threads are runnable. We believe that code is not necessarily the most revealing observable of a concurrent system as, for example, a loop that only operates over local variables does not really pose any interesting challenges to a virtual machine and can be run in parallel with pretty much anything. An arguably better metric is to study memory operations on data that are actually (or potentially) shared. Furthermore, we posit that only the threads that do a substantial amount of memory access are relevant. Consider the case of a benchmark with a large number of threads that poll on network connections. These are always runnable but do nearly no actual work. A better way to get an upper bound on contention might be to measure the number of monitor hand-offs between threads.

Dufour et al. also argue that metrics should be robust (a small change in behaviour leads to a small change in value), discriminating (a large change leads to a large change) and platform independent. We set out to follow their recommendations, while noting that robustness and discrimination are somewhat hard to define and some degree of platform dependence is unavoidable (at least due to scheduling).

Gidra et al. measured the scalability of certain runtime components, such as the various garbage collection algorithms, in the HotSpot Java VM [14]. They found that the cumulative stop-the-world pause times increase with the number of threads as does the total time spent in GC. They claim that most object scanning and copying is done by a GC thread running on a remote node. Although these results suggest an alarming lack of scalability, they need to be confirmed with the NUMA-aware version of the HotSpot GC which was not used in the study. Chen et al. [8] provide an observational scalability analysis of HotSpot. They explain some scalability issues by stalls at the hardware level (cache misses, DTLB misses, pipeline misses and cache-to-cache transfers) and measure the benefits of thread-local allocation buffers and a HotSpot heuristic for determining a good size of the young generation. They find the benefits to be application dependent, but not very VM dependent.

In terms of program instrumentation, we followed in the footsteps of Binder et al. [4]. Their approach differs in that they statically instrument core classes that are loaded before an agent can take control, which allows them to add fields to these classes. We use an agent instead, both to re-instrument classes already loaded and to instrument new classes as they load.



(a) 64-core 4-node AMD; HotSpot 1.7.

(b) Azul Vega 3, 864 processors; Azul VM 1.6.

(c) Azul Vega 3, 864 processors; Azul VM 1.6.

Figure 1: Speedup v. number of driver threads, Dacapo'09.

3. DaCapo

Blackburn et al. [6] introduced the DaCapo benchmarks in 2006 and provided performance measurements and workload characteristics, such as object size distributions, allocation rates and live sizes. The original[1] suite consists of the following benchmarks: antlr$_6$ generates a parser and lexical analyzer, bloat$_6$ performs a number of optimisations and analyses on Java bytecode files, chart$_6$ plots line graphs and renders PDF, eclipse$_6$ runs tests for the Eclipse IDE, fop$_6$ generates a PDF file, hsqldb$_6$ executes a number of transactions, jython$_6$ interprets the pybench Python benchmark, luindex$_6$ indexes a set of documents, lusearch$_6$ searches for keywords, pmd$_6$ analyzes a set of Java classes and xalan$_6$ transforms XML documents into HTML. The DaCapo 2009 suite updated a few of the original benchmarks, eclipse$_9$, fop$_9$, jython$_9$, luindex$_9$, lusearch$_9$, pmd$_9$ and xalan$_9$, as well as introducing new applications: avrora$_9$ is a simulation program, batik$_9$ produces SVG images, h2$_9$ executes a number of transactions, sunflow$_9$ renders a set of images using ray tracing, tomcat$_9$ runs a set of queries against a web server, tradebeans$_9$ and tradesoap$_9$ run the daytrader benchmark.

Multi-threading is not widely used in DaCapo 2006, with only three programs using more than one thread (lusearch$_6$, hsqldb$_6$ and xalan$_6$). DaCapo 2009 supports scaling of seven benchmarks to an arbitrary number of driver threads with identical copies of the code on subproblems. By default, the suites use the number of logical processors in the system. This does not mean, however, that the benchmarks scale well. There are two ways to evaluate scalability — vary either the number of physical cores available to the whole system (operating systems allow this), or the number of driver threads. The former has been used [8] but the latter is more convenient [14]. The implied assumption in both cases is that the amount of work done remains the same.

When run on a 64-core AMD system, sunflow$_9$ keeps improving as cores are added. Figure 1(a) shows that only sunflow$_9$ and tomcat$_9$ are able to get more than 10× speed up. tradesoap$_9$ barely gets more than a 3× improvement. tradebeans$_9$ and h2$_9$ actually report degraded throughput with higher thread counts. For comparison, Figure 1(b) reports the result on an Azul system. Here, scalability is not as good. Only three benchmarks get a speedup better than 5× (sunflow$_9$, xalan$_9$, lusearch$_9$). tomcat$_9$ scales better on the AMD. Absolute throughput numbers are also much higher on AMD. Azul numbers for more than 64 threads are shown in Figure 1(c). Other than sunflow$_9$ which sees small improvements up to 256 threads, performance is either stable or decreases with more cores (h2$_9$ is striking). Of the seven '09 benchmarks that do not support driver-thread scaling, avrora$_9$, eclipse$_9$, pmd$_9$ and luindex$_9$ are multithreaded.

[1] We distinguish versions of DaCapo by a '6' (2006) or '9' (2009) subscript.

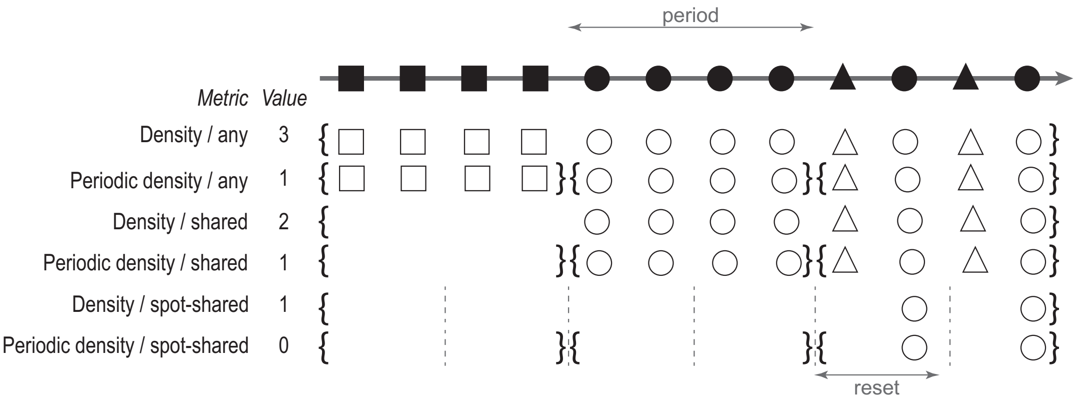


Figure 2: **Concurrency metrics.** Each solid symbol in the timeline is an operation performed by a thread on some object. Different symbols denote operations by different threads. We count the number of threads that perform operations (open symbols) in some intervals. The value of a metric is the median of the cardinalities of these sets of operations performed.

4. Metrics for Concurrency

We devised new metrics to characterize how concurrent applications use shared memory. Communication patterns between threads are key to understanding concurrent multi-threaded applications. In Java, this is typically achieved through shared memory and synchronization.[2] Thus our measurements focus on memory locations that are shared by multiple threads and on the choice of mechanisms for synchronization between threads.

4.1 Characterising Concurrency

Occasional communication adds little overhead so we focus on those threads that contribute *significantly* to the work done by a benchmark *in some interval*, e.g. the entire execution or, of more interest, some shorter interval. If we are interested in which phases of execution threads are active, the metric might be to count how many threads contribute significantly (95% of the operations) in each 100ms *period* and then compute the median. Objects may be shared only transiently and otherwise accessed by only one thread at a time. As widely spaced operations on an object by different threads are unlikely to be interesting, we might rather consider only operations by different threads in some small *spot* interval. To capture this, a metric might reset the status of 'spot-shared' objects to 'local', or unshared, at regular intervals (say, every 10ms), and then count only sharing within these intervals.

We start with an example, computing some of our metrics for the trace in Figure 2, which shows some operations, say 12 writes, performed on a single object by three threads (denoted by ■, ● and ▲). We motivate our choice of metrics as follows.

- **Density / any** counts all the different threads that contribute significantly to each operation on an object throughout the execution. In Figure 2, three threads contribute. However, as we discussed earlier, we want to exclude operations that are not particularly interesting for concurrency.
- **Density / shared** We are interested in the impact of communication between threads. This metric also computes thread density, but only for objects that have become shared. In Figure 2, the object is not shared initially so we count only the operations performed by the ● and ▲ threads.
- **Periodic density / any** This metric captures how threads may be active in different phases. For example, an initialisation phase may construct an object which is then used in later phases. To capture this, we divide execution into periods, compute the thread density in each one and report the median. In the example, we see that 1, 1 and 2 threads act on the object in each period respectively; the periodic density is 1.
- **Periodic density / shared** A single thread might only initialise an object but then hand it on to another thread for use. This pattern is unlikely to stress concurrency mechanisms — indeed, the lifetimes of the two threads may not even overlap. This metric tends to exclude such behaviour by computing the periodic thread density for objects that have become shared. In our example, there are 0, 1 and 2 operations on shared objects in each period respectively, so the median is 1.
- **Density / spot-shared** We are particularly interested in the pressure placed upon concurrency mechanisms by objects whose ownership changes rapidly. Examples might include contended locks, volatiles and other objects used for synchronization. 'Spot-sharing' resets objects' shared status regularly. This metric measures the thread density

[2] Synchronization mechanisms include `synchronized` statements, barriers with `wait`/`notify`, volatile variables, `atomic` classes from the `java.util.concurrent.atomic` package and other high level abstractions such futures or concurrent queues.

of recently (i.e. spot-shared) objects only. For example, in Figure 2, only the ● thread discovers a recently shared object.

- **Periodic density / spot-shared** Finally, we are interested in the concurrency pressure in different phases of the program. Thus, this metric computes the period density of spot-shared objects.

4.2 Traces

Some metrics require precise definition to understand what behaviours we capture. In the definitions below, we assume the presence of a full trace of the program, although our implementation is mostly on-the-fly. Let a log L be a sequence of time-stamped operations: allocations, reads, writes and monitor enters.

$$\begin{aligned} L \equiv\ & \{\mathsf{a}^{\tau}_{t} o : \text{thread } \tau \text{ allocated object } o \text{ at time } t\} \\ & \cup \{\mathsf{r}^{\tau}_{t} o : \text{thread } \tau \text{ read from object } o \text{ at time } t\} \\ & \cup \{\mathsf{w}^{\tau}_{t} o : \text{thread } \tau \text{ wrote to object } o \text{ at time } t\} \\ & \cup \{\mathsf{e}^{\tau}_{t} o : \text{thread } \tau \text{ entered monitor of object } o \text{ at time } t\} \end{aligned}$$

We assume that no two operations can happen at the same time. We use the meta-variable α to range over operations. Filter $L{\downarrow}_T$ selects operations from a time interval T:

$$L{\downarrow}_T \equiv \{\alpha^{\tau}_{t} o \in L : t \in T\}$$

To select only operations of specific type, e.g. reads, we write $L{\downarrow}_{\mathsf{r}}$. To select operations of more than one type from L, say reads and writes, we write $L{\downarrow}_{\mathsf{rw}}$. We write L^* for a set $\{L_1, L_2, \ldots, L_n\}$. Filter on L^* extends to its inner-sets: $L^*{\downarrow}_T = \{L_1{\downarrow}_T, L_2{\downarrow}_T, \ldots, L_n{\downarrow}_T\}$.

4.3 Operations

We can describe numbers of individual operations and their ratios — interesting as metrics of workload intensity and of the balance of operations — such as:

total number of reads	$\lvert L{\downarrow}_{\mathsf{r}} \rvert$
ratio of reads and writes	$\lvert L{\downarrow}_{\mathsf{r}} \rvert \,/\, \lvert L{\downarrow}_{\mathsf{w}} \rvert$

We further split the counts into operations on arrays, object instance fields and static fields. We also compute the rate of the different operations. Operations are counted in instrumented benchmarks, where overheads are large, but rates are computed by dividing them by the execution time of the *un*instrumented benchmark. Execution times obviously depend on the clock speed of the hardware used. To minimise this dependency, we normalise rates. That is, rather than reporting the absolute rate x of an operation in a benchmark, we sometimes give the *normalised rate* $(x - \mu)/\sigma$ where μ is the mean across all benchmarks and σ^2 the variance. This transforms the distribution to one with mean 0 and variance 1.

4.4 Aging

Object behaviour is well known to vary by age. We want to understand the interaction between object age and concurrency related attributes such as sharing, the age at which an object is accessed, and so on. We measure age in the number of timer ticks (with a period of 10ms), which we adjust to an approximate time in uninstrumented execution as follows:

$$approx.time = \text{ticks} \times \text{period} \times \frac{time_{\mathrm{u}}}{time_{\mathrm{i}}}$$

where $time_{\mathrm{u}}$ and $time_{\mathrm{i}}$ are the uninstrumented and instrumented execution times of the benchmark respectively.

4.5 Shared Memory Accesses

Shared memory accesses are one of the key observables for this work. We describe them using the following functions. A shared access is an operation for which the object accessed was previously allocated, written to or synchronized on by another thread.

$$\begin{aligned} ShrR(L) &\equiv \{\mathsf{r}^{\tau}_{t} o \in L \ : \ \exists\, \alpha^{\tau'}_{t'} o \in L{\downarrow}_{[0,t)} \ \wedge\ \tau' \neq \tau\} \\ ShrW(L) &\equiv \{\mathsf{w}^{\tau}_{t} o \in L \ : \ \exists\, \alpha^{\tau'}_{t'} o \in L{\downarrow}_{[0,t)} \ \wedge\ \tau' \neq \tau\} \\ ShrE(L) &\equiv \{\mathsf{e}^{\tau}_{t} o \in L \ : \ \exists\, \alpha^{\tau'}_{t'} o \in L{\downarrow}_{[0,t)} \ \wedge\ \tau' \neq \tau\} \end{aligned}$$

A write or monitor entry is said to be *alternating* if the immediately preceding access (excluding reads) was by a different thread.

$$\begin{aligned} AltW(L) \equiv \{&\mathsf{w}^{\tau}_{t} o \in L \ : \ \exists\, \alpha^{\tau'}_{t'} o \in L{\downarrow}_{[0,t),\mathsf{wea}} \ \wedge \\ & \qquad \tau' \neq \tau \ \wedge\ \nexists\, \alpha^{\tau''}_{t''} o \in L{\downarrow}_{(t',t),\mathsf{wea}}\} \\ AltE(L) \equiv \{&\mathsf{e}^{\tau}_{t} o \in L \ : \ \exists\, \alpha^{\tau'}_{t'} o \in L{\downarrow}_{[0,t)\mathsf{wea}} \ \wedge \\ & \qquad \tau' \neq \tau \ \wedge\ \nexists\, \alpha^{\tau''}_{t''} o \in L{\downarrow}_{(t',t),\mathsf{wea}}\} \end{aligned}$$

We define additional functions to select shared operations on objects that have been accessed by a different thread *recently*. For this we use time windows of a constant length, ϵ. In practice we use periods of 10ms of instrumented execution. As the instrumentation overhead is large, this corresponds to a much shorter interval of uninstrumented execution.

$$\begin{aligned} SpotShrR(L) &\equiv \{\mathsf{r}^{\tau}_{t} o \in L : \exists\, \alpha^{\tau'}_{t'} o \in L{\downarrow}_{[\epsilon\lfloor t/\epsilon \rfloor, t]} \wedge \tau' \neq \tau\} \\ SpotShrW(L) &\equiv \{\mathsf{w}^{\tau}_{t} o \in L : \exists\, \alpha^{\tau'}_{t'} o \in L{\downarrow}_{[\epsilon\lfloor t/\epsilon \rfloor, t]} \wedge \tau' \neq \tau\} \\ SpotShrE(L) &\equiv \{\mathsf{w}^{\tau}_{t} o \in L : \exists\, \alpha^{\tau'}_{t'} o \in L{\downarrow}_{[\epsilon\lfloor t/\epsilon \rfloor, t]} \wedge \tau' \neq \tau\} \end{aligned}$$

We define *spot-shared* operations based on what has already happened in their time-window, $L[\epsilon\lfloor t/\epsilon \rfloor, t]$, which allows efficient implementation with periodic timer. A system recording a full trace could define spot-sharing based on what happened up to ϵ (seconds) before the operation of interest, $L{\downarrow}_{[t-\epsilon,t]}$.

The definitions allow metrics for numbers of shared operations of each type and for their ratios. Examples include (more in our results section):

total number of shared reads	$\lvert ShrR(L)\rvert$
total number of spot-shared writes	$\lvert SpotShrW(L)\rvert$
ratio of shared reads and writes	$\lvert ShrR(L)\rvert/\lvert ShrW(L)\rvert$

Before attempting to optimise a system, it is often important to understand how contributions to a metric are distributed. Commonly, a few elements dominate. We define the function $covers$ that tests whether a set of threads Θ 'covers' a significant fraction ρ of operations in a given log L:

$$covers(\Theta, L) \equiv \quad |L{\downarrow}_{\Theta}\ | \geq \rho|L|$$

The $covset$ function returns the smallest subset of threads that cover a significant number of operations in L:

$$covset(L) \equiv \min\{\theta \in 2^{\Theta} : covers(\theta, L)\}$$

This function gives us the most active threads in the set, which is useful in understanding the program. For example, we might report the smallest age for which $\rho = 95\%$ of all accesses are to objects younger than that age. We define the thread density of a set of operations as the size of the covering set: $density_1(L) \equiv |covset(L)|$. We further define the density of multiple sets of operations, so that we can cover each individual type of operation (i.e. reads, writes). The generalized function is:

$$density(L^*) \equiv \left| \bigcup_{L \in L^*} covset(L) \right|$$

Note that $density$ is trivially not equivalent to the density of the union of the operations $density_1(\cup L^*)$. This alternative definition is possible, but our definition has the advantages that it does not give preference to any kind of operation (any element of L^*), and it is easier to implement. We base the following concurrency metrics on $density$:

- thread *density* with *any* operations:
  $density(\{L{\downarrow}_{\mathsf{r}}, L{\downarrow}_{\mathsf{w}}, L{\downarrow}_{\mathsf{e}}, L{\downarrow}_{\mathsf{a}}\})$
- thread *density* with *shared* operations:
  $density(\{ShrR(L), ShrW(L), ShrE(L)\})$
- thread *density* with *alternating modifications*:
  $density(\{AltW(L), AltE(L)\})$
- thread *density* with *spot-shared* operations:
  $density(\{SpotShrR(L), SpotShrW(L), SpotShrE(L)\})$

Shared activity may be concentrated in only a short interval of a program's execution. The level of true concurrency will be lower than the *density* if the activities of some of the threads do not overlap. To better capture this, we consider thread activity in each of a number of small intervals, or *periods* — we use periods of 100ms of instrumented execution. Formally, we define periodic density to focus on time windows of a constant length $\delta > 0$. We summarise densities using medians,

$$pdensity(L^*) \equiv \operatorname*{median}_{k \in \mathbb{N}} \left(density(L^*{\downarrow}_{[k \cdot \delta, (k+1) \cdot \delta)}) \right)$$

Based on $pdensity$, we define the following metrics:

- *periodic* thread *density* with *any* operations:
  $pdensity(\{L{\downarrow}_{\mathsf{r}}, L{\downarrow}_{\mathsf{w}}, L{\downarrow}_{\mathsf{e}}, L{\downarrow}_{\mathsf{e}}\})$
- *periodic* thread *density* with *shared* operations:
  $pdensity(\{ShrR(L), ShrW(L), ShrE(L)\})$
- *periodic* thread *density* with *alternating modifications*:
  $pdensity(\{AltW(L), AltE(L)\})$
- *periodic* thread *density* with *spot-shared* operations:
  $pdensity(\{SpotShrR(L), SpotShrW(L), SpotShrE(L)\})$

Any value returned is an upper-bound on the true concurrency, because our window can always be so large that it regards sequential activity as concurrent. δ and ϵ should be as small as possible, but δ must be much larger than ϵ for spot-metrics to be stable.

4.6 Concurrency Patterns of Shared Accesses

We explore a number of common concurrency patterns further. One such pattern is CREW: concurrent read, exclusive write. Another is unique ownership, where only one thread owns an object at a time and a thread only accesses objects that it owns. A particular example might be that a producer thread creates an object before passing it to consumer threads. A special case is a stationary object, which is created by one thread and then read by other threads but never written again.

An object o is *stationary* in trace L at time t if it is never written after being read, including if it has never been written at all. An object o is *single-writer* in trace L at time t if it has been written to by at most one thread before t:

$$\begin{aligned}
&readonly(o, t, L) \equiv |L{\downarrow}_{[0,t],\mathsf{w}}\ | = 0 \\
&writeonly(o, t, L) \equiv |L{\downarrow}_{[0,t],\mathsf{r}}\ | = 0 \\
&stationary(o, t, L) \equiv writeonly(o, t, L)\ \vee \\
&\qquad \max_{t'} \left(w^{\tau'}_{t'} o \in L{\downarrow}_{[0,t]} \right) < \min_{t'} \left(r^{\tau'}_{t''} o \in L{\downarrow}_{[0,t]} \right) \\
&singlewriter(o, t, L) \equiv |\{\tau' : \exists w^{\tau'}_{t'} o \in L{\downarrow}_{[0,t]}\}| \leq 1
\end{aligned}$$

A thread τ is the *owner* of an object o at time t if no other thread has accessed the object since τ's last access. Access α to an object o by a thread τ is *same-owner* when τ owns o:

$$\begin{aligned} owner(\tau, o, t, L) &\equiv \exists \alpha^{\tau}_{t'} o \in L{\downarrow}_{[0,t]}: \\ &\quad t' < t \ \wedge \left(\forall \alpha^{\tau'}_{t''} o \in L{\downarrow}_{[t',t]} \ \ : \tau = \tau' \right) \\ sameowner(\alpha^{\tau}_{t} o, L) &\equiv owner(\tau, o, t, L) \end{aligned}$$

Based on these definitions, we say that a shared read is 'read-only' if it is to a read-only object, 'stationary' if it is to a stationary object, 'single-writer' if it is to a single-writer object and 'same-owner' if *sameowner* holds. Similar definitions apply to writes. We now define five disjoint sharing patterns that cover all shared accesses, and we use these to characterise individual benchmarks and individual access types (all reads and writes, reads and writes separately, statics, arrays, etc):

- S1, shared *read-only/write-only* accesses:
  read-only reads and write-only writes
- S2, additional shared *stationary* accesses:
  stationary accesses that are not S1
- S3, other shared, *single-writer* accesses:
  single-writer accesses that are not S1 or S2
- S4, other shared, *same owner* accesses:
  same-owner accesses that are not S1, S2 or S3
- S5, all remaining shared accesses:
  all shared accessed not S1-S4

Note that 'read-only' reads are also stationary. While not a property of the definitions, in DaCapo stationary reads are also single-writer. Hence, S1+S2 includes all stationary reads and S1+S2+S3 includes all single-writer reads. S2 is an empty set for writes because all stationary writes are also 'write-only' by definition. Hence S1+S2 includes all stationary accesses even for writes. Not all 'write-only' writes are single-writer. For example, when an array is used to collect results from multiple threads, the collection will be formed by 'write-only' writes.

To obtain the disjoint groups S1-S5, in our implementation we count shared accesses that are read-only/write-only, stationary, single-writer, both single-writer and stationary, same-owner but neither stationary nor single-writer, and all shared accesses (of any pattern).

4.7 Shared-Used and Shared-Reachable

It is often interesting for implementers to know which objects were actually accessed by multiple threads as well as which objects could possibly have been accessed by multiple threads. For this purpose we will define the notions of *shared-used* for an object that *is* used by multiple threads and *shared-reachable* for an object that *could* be accessed by multiple threads based on reachability.

5. Measurement Methodology

We measure behaviour through instrumenting Java programs and the VM with two independent infrastructures, one that depends solely on bytecode instrumentation and the other inside Jikes RVM.

5.1 Bytecode Instrumentation

Java's dynamic instrumentation feature allows instrumenting the bytecode of any class that the VM loads. The logic of our probes is implemented in Java and runs in the same VM, but we isolate its direct impacts from our measurements. We customised btrace [2] for our needs.

Design. We insert probes at instructions of interest, such as field accesses, calls or synchronisation points. A probe might update an object-related metadata structure (e.g. to mark the object as 'shared') stored in a hash table. A probe might also increment a thread-related thread-local counter. We designed our own hash table for the metadata as we require fast look-up, safety in the presence of concurrent access and realistic accounting. Each bucket (a collision set) has a separate self-organising single-linked queue for each thread. When looking for metadata of a given object, a probe locates the correct bucket and then searches the queue of the current thread. It will find an entry there for any thread-local object and also for any shared object that the thread has accessed before. Each entry contains a weak reference to the object it represents and a regular reference to metadata for that object. Periodically, entries of dead objects are detected, processed and removed from the queues.

Whenever an entry is found in the local queue it is moved to the front (self-organisation) for performance reasons. If the entry is not found in the local queue, queues of other threads are searched (again, the search order is self-organising, first searching queues of threads where entries had been found recently). Once found, the entry is copied to the local queue for faster access next time. The only synchronization needed is between a thread moving an entry to the head of the local queue and a remote thread scanning the queue. In the rare case of contention, remote threads spin but local threads give up moving an entry to the front of the queue. A dedicated timer thread samples thread-based counter values and prints them to a file every 100ms (instrumented time). The timer thread is paused during the (stop-the-world) GC, excluding it from our measurements.

Obstacles. To circumvent the 64KB Java method size limit, we optimised btrace to generate fewer instructions. We disabled instrumentation for static initialisers for a few constant arrays in `batik` and `fop`. We disabled the bytecode verifier in order to insert probes after the `new` bytecode but before the constructor's `<init>` method. We modified btrace to instrument libraries before running the application to increase precision of tracing, and so that its own activity can be isolated. We also extended it to keep a cache of field modifiers known at instrumentation time and to pass class references to probes, so that our probes can always determine if an access is to a volatile field. Some applications are sensitive to timeouts, so we increased the initialisation deadlines in the DaCapo harness and patched some JDK 7 classes to ignore timeouts. We fixed bugs in `tomcat`$_9$ and btrace.

Platform and Benchmarks. We used DaCapo 2006-10-MR2 [6] and 9.12-bach with their largest inputs. Trace data is collected on one iteration of the benchmark excluding start-up. Timings are recorded on uninstrumented runs where we strived to get at least 5 repetitions and 10 iterations; unless specified otherwise timings are the mean of these runs. We ran HotSpot 1.7.0_01 for Linux/x86_64, btrace [2] 1.2.1 and ASM [1] 3.3 on a 4.8GHz Intel Core i7, with 4 hardware threads (hyper-threading disabled) and 16GB of RAM. For instrumented runs, we allowed the VM to use 14GB of RAM. Some experiments were run on an Azul Vega 3 with 864 processors with the Azul VM 1.6.

Limitations. Our instrumentation may prevent some optimisations. Some sampling error is inevitable, but sampling is unlikely to be very regular, so errors should be random and easy to spot when we look at multiple results. Our instrumentation may influence thread interleaving and hence some of the sharing and concurrency characteristics we measure.

5.2 Virtual Machine Instrumentation

Although our bytecode instrumentation can measure which objects are accessed by more than one thread, it cannot detect the number of objects reachable from a thread. However, components of the GC do precisely this. The extensible design of Jikes RVM [3] makes it ideal for such experiments.

Design. We customise the compiler to insert probes after memory relevant operations. We ensure that probes are neither instrumented nor call any instrumented code. We apply the probes only to application and not to VM code. We add metadata pertaining to threads to the VM, and that pertaining to objects into their header. We map memory operations into reads and writes and count them per object. For each object we keep two bitmaps, one of all threads that ever accessed it and one of all threads from which it was ever reachable. To identify reachable objects, we run mock per-thread traces through the heap, recording the set of objects each thread can reach at that time.

Obstacles. We extended Jikes RVM to instrument accesses to primitive fields. Probes for static fields require special care: they need class-based metadata which we keep in the VM's internal representation of the classes, but this is only available when the containing class is loaded. Filtering out VM-specific activity is complicated by the VM allocating its objects in the same heap. We adopt the methodology of Jones & Ryder [16] and use Jikes RVM's baseline compiler. We count only operations by application threads, including the finaliser and reference handler threads; we exclude all other VM threads. We force a special GC at shut down to count any remaining live objects.

Platform and Benchmarks. We modified a development version of Jikes RVM (21/7/11), and ran all the benchmarks that Jikes RVM could run on a dual quad-core 2.27GHz Intel Xeon machine with 12GB RAM. We let the benchmarks scale the workload to use all available processors.

6. Characterising the DaCapo Workload

The DaCapo suites' use of concurrency is complex. In some cases, it is used primarily as a design tool; in other cases, the goal is performance on parallel hardware. The DaCapo harness allows setting the number of threads that drives the workload, but this does not fully determine how many threads do a substantial amount of work concurrently. Workloads often spawn threads of their own, either directly or indirectly through libraries. Some threads *live* for the entire execution of the benchmark, some only for one iteration and some only for short-term tasks within iterations. Some are *active* throughout whole execution of the benchmark, some only throughout one iteration (e.g. `avrora`$_9$) or some phase of it (e.g. `h2`$_9$), and sometimes tens or hundreds of threads are created each for a single short-term task (e.g. `eclipse`$_9$). Moreover, the number of threads spawned may depend on the hardware — `tomcat`$_9$ spawns poller threads that service network connections depending on the number of logical processors available. On the other hand, even benchmarks that do not spawn any new threads can generate work for the VM's reference handler and finaliser threads. Our goal is provide a black-box view of these applications and thus give developers an understanding of the benchmarks' use of concurrency.

6.1 Operations

We summarise the operations performed by the DaCapo benchmarks in Figure 3. The figure compares statistically normalised rates across benchmarks (higher means that a benchmark performed relatively more operations of that particular kind than other benchmarks). For instance, the graph shows that the `lusearch`$_9$ benchmark allocates fewer but larger objects than average; it acquires fewer monitors but makes more memory accesses. `xalan`$_9$ enters by far the most locks per second of any benchmark (and nearly 4 standard deviations more than the mean over all benchmarks).

We examine memory operations known to be of interest to GC design. Some of the benchmarks in the suite are quite memory intensive. `lusearch`$_9$ has by far the largest allocation rate (4GB/s),[3] followed by `xalan`$_6$ (1.7GB/s) and `sunflow`$_9$ (1.4GB/s). The median allocation rate is 424MB/s. The highest rate of memory accesses (reads+writes) is for `sunflow`$_9$ (5.9G/s), followed by `bloat`$_6$ (3.3G/s) and `xal-`

[3] 1KB of memory is 1024 bytes, but 1K objects is 1000 objects.

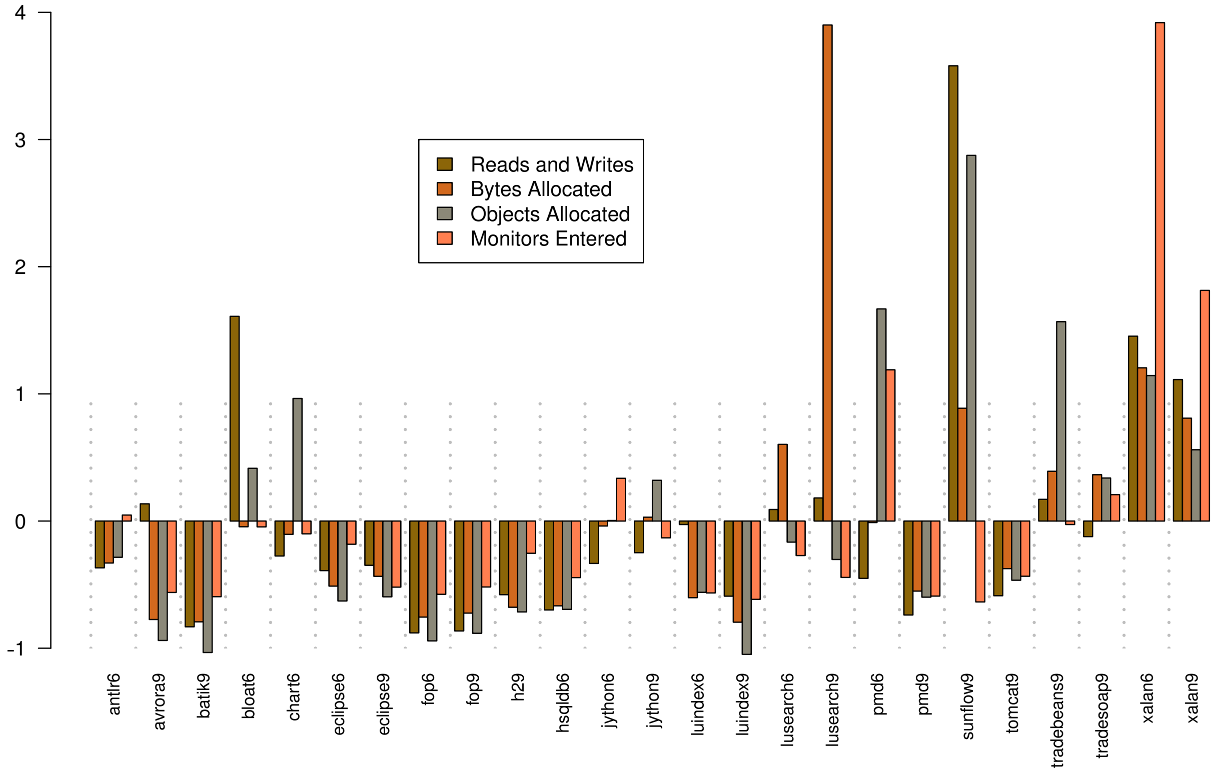


Figure 3: **Operations per second** normalised to have zero mean and unit variance. Intuitively, this shows how each benchmark differs from the DaCapo average in a way consistent across different measures such as bytes allocated or monitors entered. For example, we see that lusearch$_9$ allocates more space than average but enters fewer monitors.

an$_6$ (3.1G/s). The lowest access read+write rate is by fop$_6$ (95M/s) and the median is 806M/s.

All benchmarks do more reads than writes. The geomean read/write ratio is 3, highest with sunflow$_9$ (11×), h2$_9$ (11×) and hsqldb$_6$ (8×), and lowest with jython$_6$ (2×). The distribution is right-skewed (to higher values). Significantly more reads than writes is a well known property of most programs, e.g. exploited by replicating and generational GCs. The read/write ratio is higher for scalars than for arrays. The ratio of scalar/array memory accesses differs greatly across benchmarks. It is highest with avrora$_9$ (13×), followed by sunflow$_9$ (7×) and bloat$_6$ (6×). There are more array accesses than scalar ones for only three benchmarks, batik$_9$ (scalar/array ratio 0.65×), jython$_6$ (0.78×) and tomcat$_9$ (0.94×). Most accesses are non-static. The non-static/static ratio ranges from 7× (jython$_9$) to as much as 480× (sunflow$_9$), with geomean 40× and median 38×. Statics also have a very large read/write ratio (3.7K× geomean, 20× minimum and 436K maximum).

All metric distributions were *heavily right-skewed* towards higher values. This has a serious consequence for performance evaluation. Summaries over these benchmarks are unlikely to be robust and omitting a benchmark for any reason, good or bad, might influence results. Furthermore, results are very sensitive to errors when measuring the 'outlying' benchmarks. They are also sensitive to errors *in* these benchmarks (e.g. the very large allocation rate in lusearch$_9$ is due to a bug in the lucene library [24]).

It is therefore *essential* to interpret benchmark performance results individually. A good strategy for optimisation might be to pick one benchmark with a high overhead to attack and then look at ways to make it faster without drastically slowing others, e.g. by optimising reads and writes in sunflow$_9$, locking in xalan$_6$ or GC in lusearch$_9$.

6.2 Age

Recall that our metric is the smallest age for which 95% of all accesses are to younger objects. For all but 2 of the benchmarks, this age for reads is equal to or larger than that for writes. The geomean age over all benchmarks is about 8× larger for reads than for writes. Thus writes clearly happen early in an object's life: this agrees with other work [5]. We express this by saying that "reads are older than writes". Young accesses do not dominate. In many benchmarks, even very old (several seconds) objects are often accessed. In particular the age of reads is often proportional to execution time.

	Density				Periodic density				
	Any Ops	Alt. Modifs	Spot-shared	Shared	Any Ops	Alt. Modifs	Spot-shared	Shared	# of Threads
avrora$_9$	25	22	25	25	22	22	22	22	30
xalan$_6$	8	8	8	8	8	8	8	8	12
tomcat$_9$	9	15	12	14	9	8	7	8	26
tradesoap$_9$	13	44	32	28	7	13	12	11	267
h2$_9$	5	4	4	5	4	4	4	4	9
tradebeans$_9$	4	4	4	4	4	4	4	4	221
xalan$_9$	4	4	4	4	4	4	4	4	8
pmd$_9$	5	5	5	5	4	4	3	3	8
hsqldb$_6$	201	385	382	393	1	34	15	15	405
lusearch$_6$	63	59	66	64	44	0	57	59	68
lusearch$_9$	4	5	5	5	4	0	4	4	8
sunflow$_9$	9	9	9	9	4	0	4	4	13
antlr$_6$	1	1	1	2	1	0	0	1	4
batik$_9$	3	3	5	6	1	0	0	1	11
bloat$_6$	1	1	1	2	1	0	0	0	4
chart$_6$	1	2	1	3	1	0	0	0	6
eclipse$_6$	2	6	5	5	1	0	0	1	53
eclipse$_9$	28	197	191	72	1	1	0	1	341
fop$_6$	1	1	1	2	1	0	0	0	4
fop$_9$	1	1	1	2	1	0	0	1	4
jython$_6$	1	1	1	2	1	0	0	0	4
jython$_9$	1	1	2	2	1	0	0	1	4
luindex$_6$	1	2	2	2	1	0	0	1	4
luindex$_9$	2	3	3	3	1	0	0	1	5
pmd$_6$	1	1	1	2	1	0	0	1	4

Table 1: **Levels of concurrency.** Numbers of threads that contribute significantly over all execution or periodically (within any 100ms interval). Work is any read, write or monitor enter (Any, Spot-shared, Shared), or alternating writes and monitor enters (Alt.Modifs).

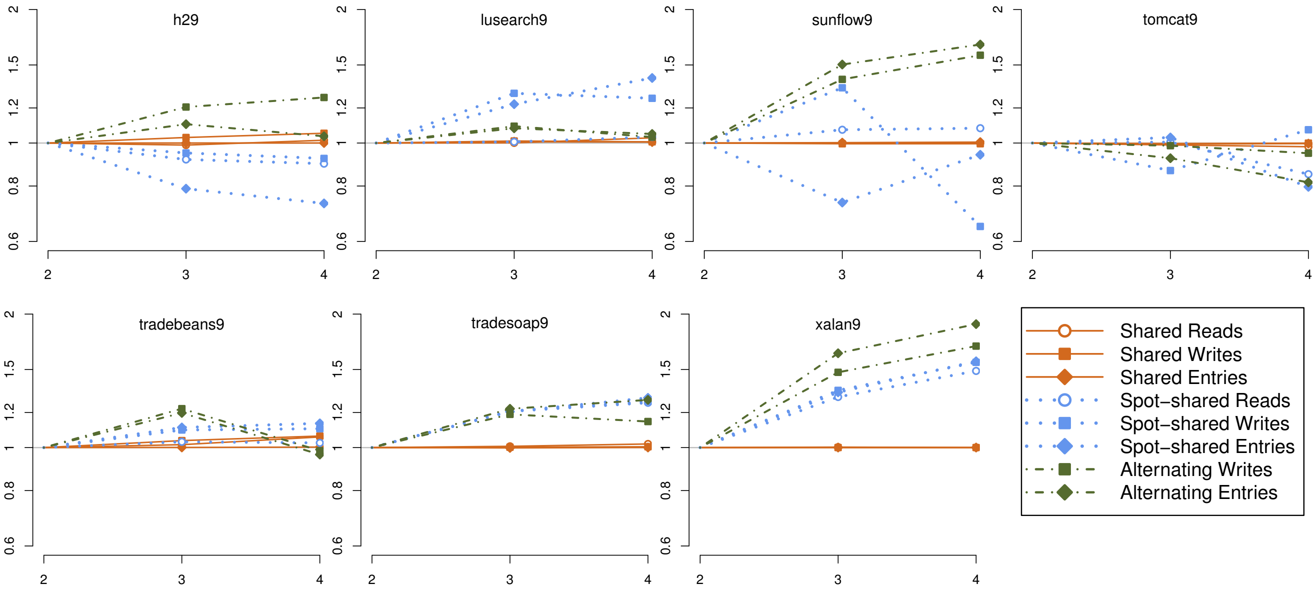


Figure 4: **Increase in shared operations as driver threads are increased from 2 to 4.**

6.3 Levels of Concurrency

Table 1 gives the number of threads that contribute significantly to DaCapo workloads on a 4-core machine.[4] It tells us how parallel and multi-threaded the benchmarks really are, not simply the number of threads that they spawn.[5]

DaCapo'09 "was designed to expose richer behaviour and concurrency on large working sets" [13]. However, our measurements show that only 9 of 14 DaCapo'09 benchmarks (and 2 of 11 DaCapo'06) are meaningfully parallel, i.e. have a periodic density/any ops greater than 1. Only these benchmarks can be expected to scale. Note a large value for periodic density/any ops is an indication of parallel slackness [22].

eclipse$_9$ and hsqldb$_6$ are heavily multi-threaded (large density/any ops) but not parallel. Multi-threaded non-parallel benchmarks, as expected, do not pose a significant challenge to shared memory (periodic density with alt modifs, spot-shared, and shared ops is 1 or 0), and hence are not suitable for evaluating concurrency-related performance optimisations. Indeed, the lack of any challenge to shared memory is good for scalability of parallel benchmarks. sunflow$_9$ and lusearch$_9$ have scarcely any alternating modifications; sunflow$_9$ scales best and lusearch$_9$ second/third best of the DaCapo'09 benchmarks). Note that few alternating modifications implies few contended monitor enters.

The number of threads created (last column in Table 1, which includes the finaliser and the reference handler threads) is a poor approximation of concurrency — the number of threads doing a meaningful amount of work (density/any ops) is much smaller, and even smaller is the number of threads doing so concurrently (periodic density/any ops).

6.4 Scaling Workloads

We explore how the DaCapo'09 benchmarks with significant levels of concurrency (h2$_9$, lusearch$_9$, sunflow$_9$, tomcat$_9$, tradebeans$_9$, tradesoap$_9$, xalan$_9$) behave as we drive them with 1, 2, 3 or 4 threads. If developers are to use a benchmark for scalability experiments, they need some intuition for the impact of adding cores/threads. For instance, they may expect a scalable benchmark either to divide the same amount of work among more threads (better performance) or to continue to give the same quantum of work to each thread (more work done). Above all, they need a benchmark to behave predictably.

Unshared operations behave as expected: their numbers are preserved as the number of driver threads increases. Unfortunately, none of the benchmarks scales its shared operations in a predictable and consistent way. lusearch$_9$, xalan$_9$ and tradesoap$_9$ seem to perform the same number of 'shared' operations for 2, 3 and 4 threads, but fewer for 1 thread. This suggests that the degree of sharing in these programs is determined directly or indirectly by the number of driver threads. On the other hand, the level of sharing in the remaining benchmarks (h2$_9$, sunflow$_9$, tomcat$_9$, tradebeans$_9$) seems to be fixed.

Figure 4 compares the change in numbers of operations with 2, 3 and 4 driver threads. The figure shows 'shared' and 'spot-shared' reads, writes and monitor enters, as well as 'alternating' writes and monitor enters. None of the benchmarks scale the numbers of either 'spot-sharing' operations or 'alternating modifications' in a predictable fashion: often, as the number of one operation increases the number of another decreases. As the number of driver threads increases from 2 to 4, the number of alternating reads in xalan$_9$ increases by over 1.5×. xalan$_9$ is memory intensive so this suggests that memory is likely to become bottleneck. The 'shared' operations stay about the same with all benchmarks, which tallies with the intuition that worker threads make the same number of accesses if the problem size remains constant. tomcat$_9$ is the best scaling benchmark. Observe that its 'spot-shared' and 'alternating' operations do not increase with the number of driver threads: both of these kinds of operation are inhibitors to scalability. In contrast, the number of 'alternating writes' increases in h2$_9$, the worst scaling benchmark. Alternating entries also appear to increase with workers in sunflow$_9$, but do not hinder scaling because they are so infrequent (Table 3).

These results show that one cannot easily evaluate how the benefit of a VM optimisation changes with the number of processors, at least not without checking the changes in concurrency stress.

6.5 Shared Memory Accesses

Most DaCapo'06 benchmarks have a single user thread, so it is no surprise that in 10 out of 25 benchmarks less than 0.5% of reads and writes are 'shared'. In each of the remaining 15 benchmarks, the fraction of shared reads tends to be much higher than the fraction of shared writes (Table 2). Shared accesses are therefore even more heavily dominated by reads (28× by geomean) than are thread-local accesses (3×). Array accesses are more shared than scalar accesses in most of these benchmarks (10 out of the remaining 15). Statics are particularly highly shared: in 12 of these benchmarks, at least 95% of their accesses are shared.[6] Sharing is also biased towards statics: although only a geomean of about 2% of all accesses are static, 22% of all shared accesses are to statics.

Counting only 'spot-sharing' (repeated, recent sharing) reduces the fraction of shared accesses by varying degrees. It falls by the most in avrora$_9$ (from 89% to only 6%), followed by h2$_9$ (49% to 6%) and tradesoap$_9$ (36% to 15%). Only 4 benchmarks have more than 10% of accesses spot-shared: sunflow$_9$ (49%), tradebeans$_9$ (35%), trade-

[4] Note that the metrics allow density to be greater with 'alt. modifs' than 'any ops'.

[5] *Threading in DaCapo 9-12*, dacapobench.org

[6] We count sharing of statics at class granularity, so two threads accessing different static fields of the same class would count as sharing.

	Reads and Writes	Reads	Writes	Arrays	Scalars	Statics	Reads and Writes	Reads	Writes	Arrays	Scalars	Statics
	Shared / All						Spot-Shared / All					
avrora$_9$	89	90	83	95	88	99	6	7	1	6	5	97
sunflow$_9$	53	58	0	75	50	100	49	53	0	74	45	97
h2$_9$	49	53	5	54	42	99	6	7	1	5	4	27
tradebeans$_9$	48	57	6	37	45	100	35	41	4	26	29	94
tradesoap$_9$	36	45	8	19	43	100	15	20	2	10	14	81
tomcat$_9$	20	23	13	17	18	100	8	11	3	5	8	78
xalan$_6$	19	21	6	19	12	100	14	16	3	11	8	89
eclipse$_9$	17	20	6	15	14	68	1	1	0	1	1	1
eclipse$_6$	13	17	0	18	2	97	0	0	0	0	0	0
hsqldb$_6$	13	14	13	9	13	70	6	7	1	6	4	61
xalan$_9$	13	14	6	12	10	100	7	8	2	6	5	78
lusearch$_6$	8	10	5	11	6	100	2	3	0	3	0	97
pmd$_9$	6	8	0	6	4	99	2	3	0	2	1	50
jython$_9$	3	4	0	0	0	20	0	0	0	0	0	0
lusearch$_9$	3	4	0	2	0	100	3	3	0	2	0	96

Table 2: Percentage of accesses that were shared or spot-shared, across different access types, scalars and arrays and statics. The figure shows only scalable benchmarks that have at least 0.5% reads and writes shared or spot-shared, using 4 driver threads.

soap$_9$ (15%) and xalan$_6$ (14%); with 'any sharing' only 4 had *less* than 10%. Reads continue to be more likely than writes to be 'spot-shared', and 20% of all spot-shared accesses are still to statics.

The fractions of 'alternating modifications' are very low, the highest being tradebeans$_9$ (1.2% of all writes), followed by h2$_9$ (0.7%) and hsqldb$_6$ (0.6%). Only 7 benchmarks have over 0.2% (tradebeans$_9$, h2$_9$, hsqldb$_6$, tomcat$_9$, tradesoap$_9$, xalan$_9$ and xalan$_6$). In contrast to the measures above, scalars are more shared than arrays in nearly all benchmarks. Static alternating writes vary substantially between benchmarks, from 73% for tradebeans$_9$ to 0.2% for h2$_9$. They account for 5% of all alternating writes, despite the rarity of static writes.

In Figure 5 we further discriminate shared accesses by sharing pattern, for every benchmark with at least 0.5% of reads and writes shared. The topmost graph shows all reads and writes, with the bold black line indicating the percentage of shared reads and writes. Sharing may take several forms, each presenting different stress to the VM. The coloured bars summarise these patterns by increasing level of challenge. The lowest bar (S1) shows the percentage of shared accesses to objects that were only read or only written; the second bar (S2) adds 'stationary' objects (to which there may have been a sequence of writes, then a read but no further writes[7]); and the third bar (S3) adds any objects written by only one thread yet not included in the previous bars. The fourth bar (S4) adds any accesses to objects that were last accessed by the same thread, so ownership[8] does not change. The last bar (S5) accounts for the remaining, and most challenging, shared accesses; most intriguingly there are scarcely any. From the other graphs in the same figure, we see that this kind of *write* sharing is present in some benchmarks (chiefly tomcat$_9$ and pmd$_9$, and for statics tradebeans$_9$), but becomes negligible when reads are included. We can also see that almost all shared accesses by sunflow$_9$, the best scaling benchmark from Figure 1, fall into the first three categories. Nearly all its shared reads are single-writer; its shared writes are to write-only arrays but account for little sharing.

Arrays and statics are very likely to be stationary. Many instance field reads are single-writer, if not stationary, but writes to single-writer objects are less common for scalars than arrays. The incidence of read-only statics is at first surprising. However, the VM and DaCapo harness load and initialise several hundred classes before our Java agent can instrument them. Many of these statics will be used as constants and thus appear as 'read-only' (S1) whereas a better classification might be 'stationary' (S2). Apart from tradebeans$_9$ statics and lusearch$_9$ instance field reads, sharing that changes ownership but is neither stationary nor single writer (S5) is largely concentrated on arrays.

In summary, we see that the workloads include a relatively small number of shared memory accesses, and only to memory shared through simple patterns.

[7] Hence, no stationary write appears in the figures.

[8] We treat any access as possession of ownership.

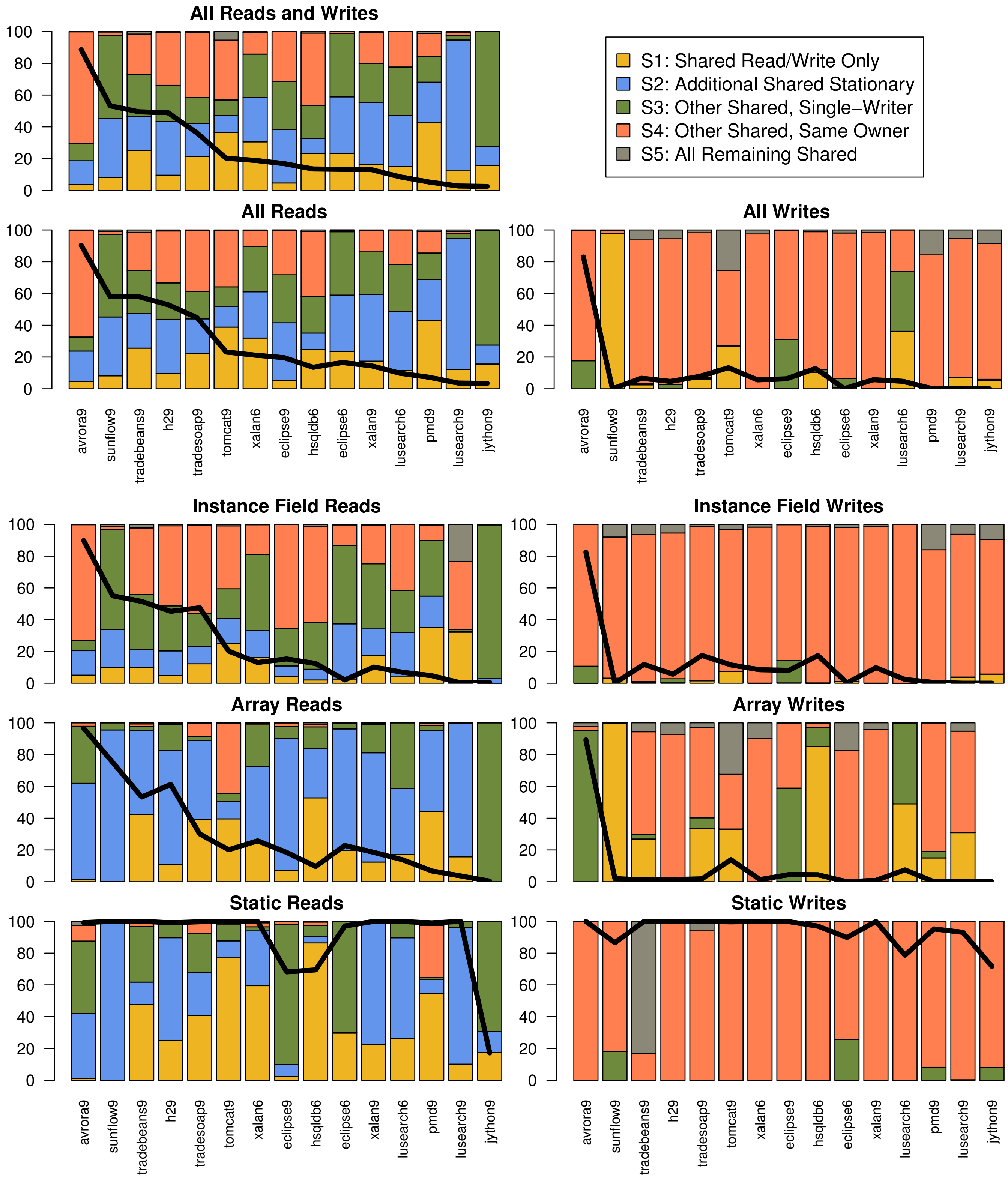


Figure 5: Sharing patterns for different kinds of accesses. The black line denotes accesses of each kind shared in any pattern (numbers from Table 2). The coloured bars show the percentage of shared access that conform to a particular pattern, such as 'read-only or write-only' (S1), 'stationary but not read-only' (S2), 'single-writer but not stationary' (S3) and so on.

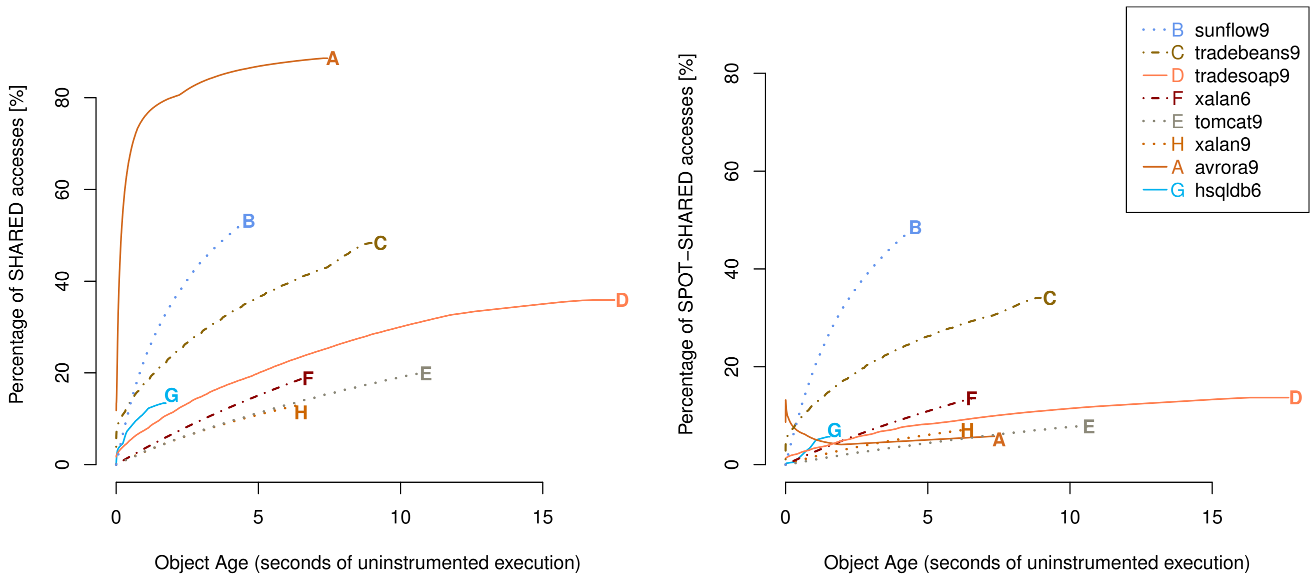


Figure 6: The percentage of accesses up to a given age to objects that are shared or spot-shared. Only benchmarks with at least 5% of accesses spot-shared are shown. Benchmarks are ordered in the keys by spot-sharing. Unusually, many objects in avrora$_9$ are shared at an early age but spot-sharing soon decreases.

6.6 Shared Accesses By Age

It is interesting for GC design to ask whether sharing is related to age. We record the age and locality of accesses, for both 'any' and 'spot' sharing. Figure 6 shows the results for benchmarks with at least 5% of accesses spot-shared. We observe that the older an object accessed, the higher is the chance that it is shared. Young accesses are likely to be to local objects. This is somewhat intuitive, as we would expect that objects need some time to become shared, but this trend continues even for very old objects. This trend is exhibited by all these benchmarks except for avrora$_9$, where the rate of increase in the fraction of accesses to spot-shared objects drops sharply after about 10ms, although the 'any sharing' rate keeps increasing rapidly. This suggests that in avrora$_9$ single threads often make access to objects that were earlier shared. However, for the most part, young objects are very unlikely to be shared, suggesting opportunities for discriminating their handling.

6.7 Lock Operations

Synchronization operations are detailed in Table 3. The first column shows the rate of monitor enters (acquisitions of a monitor). Some single-threaded benchmarks have high rates (e.g. jython$_6$ 13M/s), whereas the best-scaling multithreaded benchmark (sunflow$_9$) makes only 1K enters per second. The next three columns show the percentage of lock enters on objects that are shared, spot-shared or prone to alternating modifications (which includes locking by alternating threads). Single-threaded benchmarks may show shared enters because of threads used by the VM or libraries, such as the finalizer, reference handler, AWT and Java2D threads.

The next three columns (5-7) show the maximum age of most objects locked, i.e. where 95% of locks were to objects of that age or younger. The age is given in seconds of uninstrumented execution (shown in the last column). In many but not all benchmarks, this age is large or even close to the total execution time, suggesting that long-lived objects are subject to locking throughout their lifetime. Unsurprisingly, the number of locks acquired on spot-shared objects seems to have a substantial effect on scalability. The benchmarks that scale best on AMD, sunflow$_9$ and tomcat$_9$, use fewer monitor enters, especially fewer spot-shared enters, than the worst scaling ones, h2$_9$ and tradebeans$_9$ (Figure 1(a)).

6.8 Volatile Accesses

Volatile fields serve as a means of communication among threads, but the benchmarks make comparatively few accesses to volatiles (5% in jython$_9$; 1% in antlr$_6$, tomcat$_9$, hsqldb$_6$, fop$_9$, and jython$_9$; less in others). However, the rates of volatile accesses are still reasonably high (Figure 7). The highest rate is with jython$_9$ (26M/s), the median is 664K/s and the lowest rate is with sunflow$_9$, the most scalable benchmark (4.6K/s only).

Many static writes are volatile: in 12 of the 25 benchmarks, more than half are to volatile fields. Less than a few percent of other types of accesses (static reads, non-static reads, non-static writes) are to volatiles. Because static fields are (potentially) accessible to all threads, synchronisation is necessary and volatile provides a convenient mecha-

	Mon. Enters	Shared	Spot-shared	Alternating	All Enters	Shared	Spot-shared	Native Exec.
	Rate [K/s]	% of Enters			Max Age of Most [s]			Time [s]
antlr$_6$	8912	0	0	0	0.0	1.7	1.7	1.8
avrora$_9$	972	66	65	57	6.9	7.1	7.0	7.4
batik$_9$	538	1	0	0	0.2	2.8	1.2	3.4
bloat$_6$	7699	0	0	0	0.0	0.7	0.9	11.0
chart$_6$	6986	0	0	0	0.0	4.0	4.2	4.8
eclipse$_6$	5918	3	0	0	0.0	11.6	11.7	26.1
eclipse$_9$	1517	27	4	1	22.7	28.6	15.6	39.8
fop$_6$	782	1	0	0	0.2	0.6	0.6	1.0
fop$_9$	1534	0	0	0	0.9	0.9	0.8	1.4
h2$_9$	4991	92	19	1	7.2	7.2	7.3	20.4
hsqldb$_6$	2498	12	7	3	0.8	1.6	1.5	3.2
jython$_6$	12680	0	0	0	11.6	0.9	1.3	13.9
jython$_9$	6575	14	0	0	12.4	16.3	1.5	16.9
luindex$_6$	923	1	0	0	1.6	2.3	2.2	2.3
luindex$_9$	264	3	1	0	0.8	1.1	1.1	1.2
lusearch$_6$	4759	37	0	0	2.3	2.5	0.1	3.0
lusearch$_9$	2514	0	0	0	0.1	2.3	2.1	2.5
pmd$_6$	23813	0	0	0	0.0	4.5	4.8	5.5
pmd$_9$	599	62	27	7	2.1	2.1	2.2	3.3
sunflow$_9$	1	46	15	16	3.4	4.1	4.0	4.7
tomcat$_9$	2635	16	6	3	7.1	9.9	9.5	10.9
tradebeans$_9$	7941	99	44	6	8.5	8.5	8.4	8.9
tradesoap$_9$	11006	23	9	3	11.4	15.2	15.0	17.5
xalan$_6$	59421	22	15	3	5.1	6.3	6.2	6.8
xalan$_9$	31954	11	5	2	3.4	6.1	6.0	6.6

Table 3: Monitor enter operations (locks). The table shows the rate of enters, what percentage of these were shared, spot-shared and alternating, and the maximum ages of most (minimum age for which 95% of locks were to objects of that age or younger). E.g. sunflow$_9$ only acquires about 1,000 monitors per second, 46% of which protected shared objects; its shared objects tend to be older than its unshared ones. Although pmd$_6$ is multi-threaded and has a high rate of entries, nearly all are to local objects.

nism, often used for e.g. lazy initialisation by a single thread. Static fields are commonly used for constants. The evidence from the statistics for static reads and writes supports this hypothesis (Figure 5): we see that the overwhelming majority of static reads are to (at worst) single-writer objects and scarcely any static writes change ownership (except for tradebeans$_9$).

As volatiles are used for similar tasks to locks, we compare the number of volatile accesses and monitor enters. Different benchmarks favour different mechanisms. Although on (geometric) average, enter is used 6× (median 4×) more frequently than volatile access, some benchmarks use more volatile accesses than enters — jython$_9$ (4×), sunflow$_9$ (4×), hsqldb$_6$ (1.2×) — and some use about the same (pmd$_9$, tomcat$_9$, fop$_9$). Volatile access seems as important as locking for optimisation.

We investigated which volatile fields are accessed most often, i.e. account for 95% of volatile accesses in a benchmark. Typically, there were few and these were used by only a few call sites. Over all the DaCapo benchmarks, just 102 fields from 59 classes and 35 packages cover 95% of the volatile accesses of each benchmark. Most of these classes are from the Java I/O, references, reflection, concurrent (including atomics), and collection class libraries (Table 4). Some are from benchmark-specific libraries.

10 out of 25 benchmarks have larger read/write ratios for volatiles than for non-volatiles (the extreme is hsqldb$_6$: 339× vs. 10×).

6.9 Wait and Notify

Interestingly, the wait and notify methods are rarely used. Of all the benchmarks, 15 make fewer than 100 of these calls (less than 13 per second) — rates are shown in Figure 7. Six benchmarks issue fewer than 100 calls per second (tradesoap$_9$, eclipse$_6$, hsqldb$_6$, lusearch$_6$, xalan$_6$ and h2$_9$). The remaining ones are lusearch$_9$ (158/s), xalan$_9$ (361/s), eclipse$_9$ (17K/s) and avrora$_9$ (61K/s). wait is invoked on only 29 classes in the whole suite. The hottest class is

Package (# of classes)	Comment, Selected Classes
java.io (6)	*Stream, RandomAccessFile
java.lang (6)	Class, reflect.Constructor, reflect.Method, ref.ReferenceQueue, System, Thread
java.math (1)	BigDecimal
java.net (1)	URI
java.nio (2)	I/O, locale
java.util.concurrent.atomic (4)	Atomic Boolean, Integer, Long, Reference
java.util.concurrent.locks (2)	AbstractQueueSynchronizer
java.util.concurrent (4)	Concurrent HashMap, LinkedQueue
java.util.zip (3)	ZipFile, *InputStream
java.util (7)	HashMap, HashTable, AbstractMap, logging.Logger, regex.Pattern, locale …
sun (7)	Streams, sockets, fonts, …
org.apache (6)	Benchmark libraries (batik, catalina, fop, lucene, tomcat). I/O, properties, …
org.eclipse.core.internal (3)	OrderedLock, ResourceInfo, ElementTree
org.h2 (3)	Database, Session, MemoryUtils
org.hsqldb (2)	jdbcStatement, Session
org.osgi (1)	ServiceTracker
org.python.core (1)	PyUnicode

Table 4: Classes with volatile fields accessed most in the DaCapo benchmarks. Standard Java libraries are in the upper part, benchmark libraries in the lower part.

RippleSynchronizer in avrora$_9$, which is the target of about 6M calls; it implements global simulation time for a parallel discrete event simulator. Simulator threads use it to wait for data from other threads that are not keeping up (and notify the synchroniser of their progress). The next hottest class is ReadManager in eclipse$_9$ (about 33K calls), which synchronises worker threads that read files in parallel. Apart from avrora$_9$ and eclipse$_9$, the number of these calls is small enough that performance should not be an issue.

6.10 Concurrent APIs

The java.util.concurrent APIs are widely used in jython$_9$ (over 175M total or 7.4M/s); five other benchmarks have between 100–349K calls/s (h2$_9$, tradebeans$_9$, hsqldb$_6$, tomcat$_9$); another five do 10–16K/s (luindex$_6$, antlr$_6$, eclipse$_9$, pmd$_9$, xalan$_6$). The rates are shown in Figure 7. The remaining benchmarks issue fewer than 10K calls per second. avrora$_9$ and sunflow$_9$ have the lowest number of calls (less than 200) of all benchmarks. Only jython$_9$, pmd$_9$, tomcat$_9$, tradebeans$_9$ and tradesoap$_9$ call the concurrency APIs directly; other benchmarks do so only through other Java libraries. Overall, the suites used atomics (integer, boolean, long, reference, reference field updater), concurrent hash maps and linked queues, copy-on-write sets and array lists, linked blocking queues, re-entrant read-write locks, semaphores, thread pools and future tasks.

In summary, what can we learn from DaCapo's usage of different concurrency mechanisms, such as monitor enters, volatile accesses, concurrent API and wait/notify calls? Several benchmarks are significant outliers. jython$_9$ makes heavy use of all these mechanisms except wait/notify (particularly concurrent hash maps, which in turn use volatiles), even though it is essentially single-threaded (Table 1). The behaviour of some benchmarks has changed significantly between releases. xalan$_9$ has by far the highest monitor enter rate of all benchmarks and also has a very high volatile access rate; xalan$_6$ also has very high rates but lower than xalan$_9$. pmd$_6$ uses locks heavily but does not use volatiles that much. jython$_6$ behaves very differently to jython$_9$, with a much lower rate of concurrent calls and volatiles, but more monitor enters. avrora$_9$ makes much use of wait/notify calls (and hence monitors), but has a very low rate of concurrent calls and quite a low rate of volatile accesses. sunflow$_9$ stands out in that it has the lowest enters rate and makes limited use of the other mechanisms compared to other benchmarks; it is the most scalable benchmark (Figure 1(a)) in the suite.

6.11 Shared-Used and Shared-Reachable Objects

VMs have an interest in which objects are, or could potentially be, accessed by more than one thread. There are a number of advantages to segregating objects into thread-local heaplets that can be collected in isolation, without stopping other threads. This approach accords well with transactional workloads where there is almost no trace left after a transaction commits. It makes it easier to keep thread-local objects on the local node in a NUMA system, helping to address the problems of the 'allocation wall' [26] or the 'memory wall'.[9] It can also reduce coherency traffic due to false sharing.

Static or dynamic analyses (or a combination of both) can be used to identify 'shared' objects, trading precision for cost of analysis and object management. The benefits depend on precision. An escape analysis may determine an object to be 'shared' if it is potentially accessible by more than

[9] www.azulsystems.com/presentations/qconsf-state-of-the-art-in-java-gc

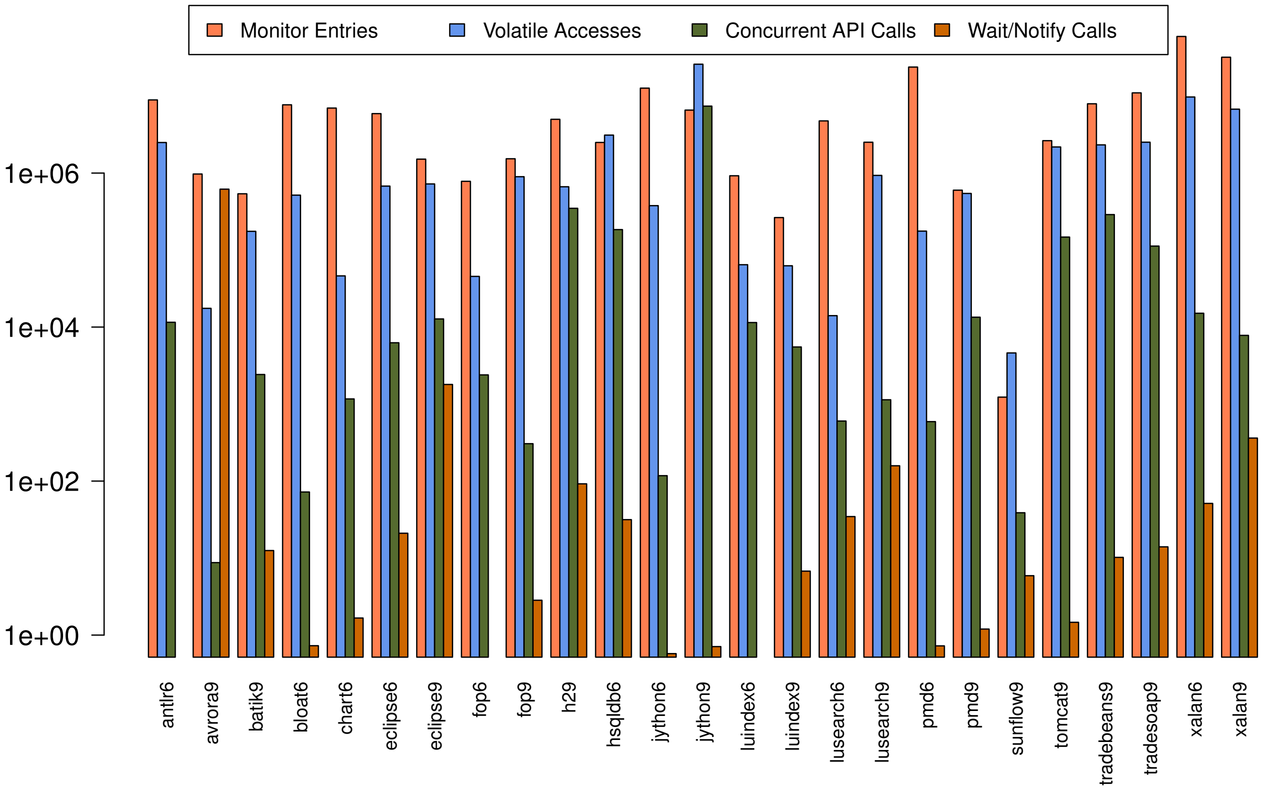


Figure 7: Rates of monitor entries, volatile accesses, concurrent API calls and wait+notify calls (operations per second).

one thread [21]. Alternatively, in languages like ML, only objects annotated as mutable need be considered 'shared' as immutable objects can be freely copied [9, 10]. Static analyses overestimate the number of objects actually shared. Most are conservative in the face of (or ignore) dynamic class loading; Jones & King [15] use an optimistic, and hence more precise, analysis that falls back at run-time to more conservative assumptions should a dynamically loaded class invalidate parts of a prior analysis.

Runtime analysis delivers more precise results but requires read or write barriers to detect escape [17]. The most conservative approach is to have a write barrier mark as 'shared' the entire transitive closure of an object at the point it becomes reachable from another 'shared' object [11] — we call these objects *shared-reachable*. Precise techniques identify as *shared-used* only those objects that are actually accessed by more than one thread, but permit references from shared to local objects. Here, a read barrier is needed to detect sharing. Whilst read barriers are generally considered expensive [25], Haskell combines the barrier cheaply with its closure entry mechanism [19].

We compared the incidence of shared-used and shared-reachable objects in Jikes RVM. Note that even benchmarks with a single user thread also employ a finaliser thread and so will report a non-zero number of objects that are shared-reachable, e.g. from statics. Shared-use was detected by our probes on the fly. To compute shared-reachability, we ran a GC after every 10MB of allocation to give a lower bound of the set of shared-reachable objects.

We measured the proportion of all objects allocated by a benchmark that became shared-used or shared-reachable (Figure 8), by number and by volume. Reachability is an even poorer approximation than we expected of how many objects will be accessed by more than one thread. Although only a negligible fraction of space is actually shared in these benchmarks (7% in avrora$_9$, but below 1% in others, and we know that in avrora$_9$ about 70% of shared accesses are made without ownership changes, see Figure 5), the shared-reachable volume is large: over 10% in 7 of 14 benchmarks, over 20% in 4 and as much as 61% in hsqldb$_6$. In 11 of the 14 benchmarks, the fraction of shared-reachable objects is larger by volume than by number, so we can speculate that shared-reachable objects tend to be the bigger ones. Surprisingly, this is not true for shared-used objects: 5 of the 14 benchmarks have a higher fraction of shared-used objects by number than by volume.

We also compared the fraction of reads and writes to objects that were shared-used or shared-reachable (Figure 9). There are many more shared-reachable objects than are actually shared; the difference is highest with xalan$_9$ (19 percentage points for reads and 17 for writes). We tracked which objects were reachable from static fields at every GC. hsql-

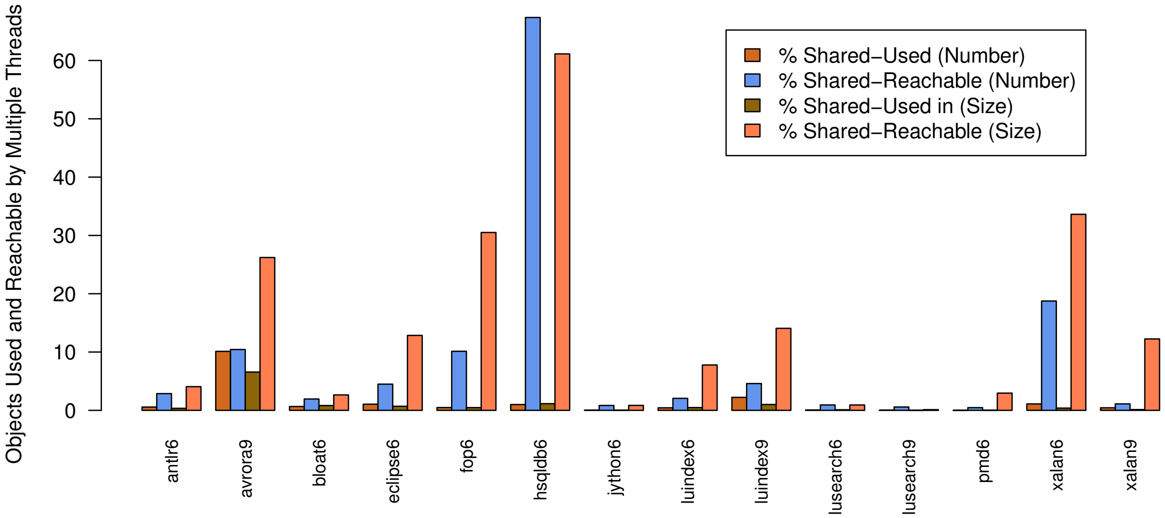


Figure 8: Percentage of all objects ever used by (shared-used) or reachable from multiple threads (shared-reachable).

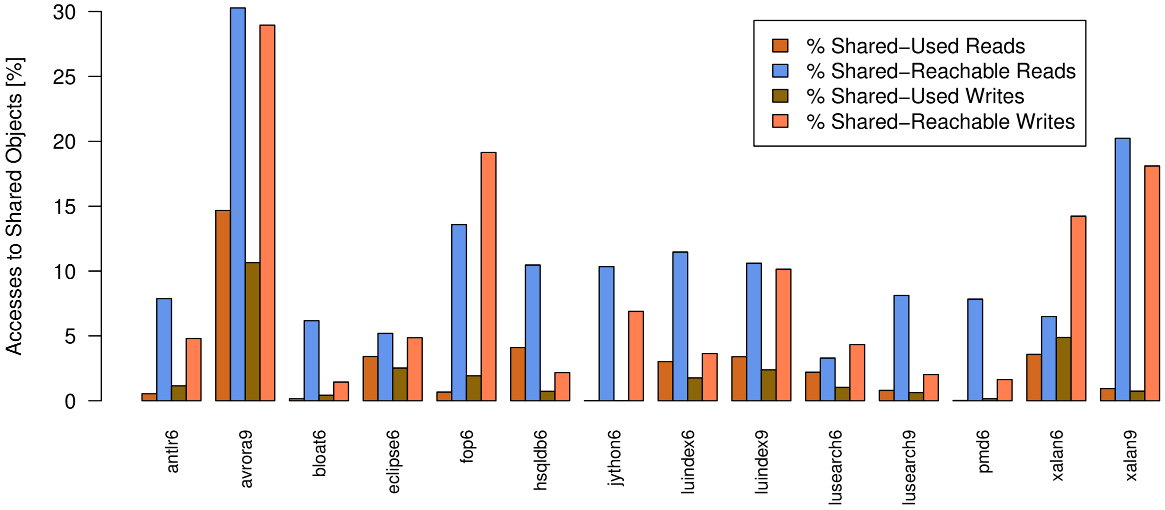


Figure 9: Percentage of reads from and writes to objects that were ever *used* by multiple threads (shared-used) or objects that were ever potentially *reachable* from multiple threads (shared-reachable).

db$_6$ has the largest number and volume of shared-reachable objects. It turns out that 97% of objects that were shared-reachable were also reachable from at least one static field at some point. Only five types accounted for a significant fraction (more than 95%) of these objects.

6.12 Unused Objects

A noticeable number of 'write-only' objects are initialised by the benchmarks but never read nor synchronised, some read-only objects are never initialised nor synchronised, some 'locked-only' objects are used only for synchronisation — a common metaphor — and some objects are neither read, written nor synchronised. Our tool only captures accesses from Java, but we manually verified selected allocation sites in the benchmarks and found that surprisingly many of the objects are not used by native code, either.

Figure 10 shows a particularly large proportion of write-only objects in jython$_6$, jython$_9$ and chart$_6$. In jython, write-only objects are mostly backing arrays of string buffers allocated eagerly by the lexical analyser but not used later. These buffers are created from a single-character string, but the class libraries pre-allocate a 17-char backing array for each. In chart$_6$, the vast majority of write-only objects are backing arrays for lines of text input: string comparisons on two lines ignore the arrays if their lengths differ. chart$_6$ also creates a significant number of notification objects, but frequently there are no listeners to be notified so these objects are never read. pmd$_6$ pre-initialises many objects to repre-

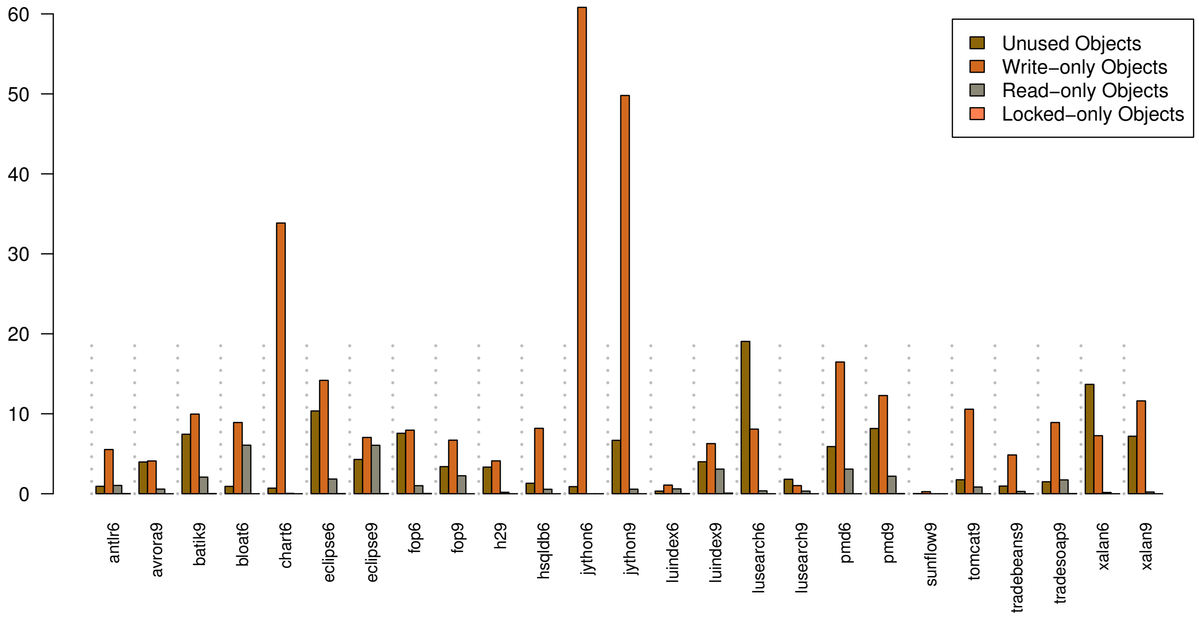


Figure 10: Volume of objects that are only read, only written, only synchronized or unused by Java. While some of these objects are read or written by native code, it turns out that many, particularly write-only, objects are not read by native code either. In $\texttt{chart}_6$ write-only objects are mostly strings representing lines of input that are skipped during processing. In $\texttt{jython}_6$ they are mostly compiler objects eagerly allocated and initialised but not needed later.

sent the contexts of XPath predicates, but these are often not needed and hence write-only. In general, write-only objects arise through eager allocation and/or preparation of data that is not always needed for later computation.

Objects are often read-only because only their default value is needed. $\texttt{chart}_6$ also creates string prefixes as part of a decimal number formatter but often no characters are added so the prefix's backing array is never written. Eager allocation also leads to objects not being used subsequently. For example, the python compiler in jython pre-allocates hashmaps for metadata that are not always needed. Many big integers representing zero produce unused single-element int arrays. In jython, the DaCapo results-validation reads a long text file, which includes many empty lines leading to the creation of many zero-length byte arrays.

We also found instances of objects that were read-only or unused in Java, but were written or read from the native code. For instance, I/O buffers used when reading a file appeared 'read-only'. Buffers used for class loading appeared 'unused' as they were only read by the VM's native code.

7. Conclusion

We provide a number of platform independent metrics of concurrency, shared memory use and scalability of Java applications. Based on tracing, these metrics allow black-box studies of large applications, giving the developers an understanding of what applications really do. Our approach takes into account application libraries and standard Java libraries, but abstracts from any hardware platform or VM implementation. We believe that it will help application developers gain insights into their programs by understanding scalability bottlenecks independent of the platform used. Similarly VM developers need confidence that the benchmarks they use to measure new optimisations do stress the optimisation they are trying to test. For this they need not only good application benchmarks but also to understand their behaviour. For example, to test locking optimisations, they need applications which use locks in a non-trivial manner. We provide metrics to examine how the programs use locking and other concurrency mechanisms such as volatile accesses, synchronisation through atomic memory operations, or frequent accesses to shared memory by different threads.

We implement our metrics in two tools, one which uses bytecode instrumentation and one which modifies a Java VM. We provide an observational study of what the DaCapo'09 and DaCapo'06 benchmarks really do in terms of concurrency and shared memory use. We believe our results will help developers to choose the right benchmarks for their purposes. We measure how many threads in these benchmarks actually contribute significantly to the work of the benchmark. While many benchmarks are multi-threaded, we find that their threads do not communicate much or communicate only in limited ways. Moreover, communication often does not change predictably as we vary the number of cores deployed. Although limited synchronisation is good for scalability, these limitations have to be kept in mind when benchmarks are used as test applications by VM implementors attempting to optimise the concurrency support they provide.

We hope that our findings for large Java applications will inform VM development, particularly memory management and concurrent GCs for many-core systems with shared memory. The DaCapo benchmarks do not scale beyond 20 or so cores on the platforms we have looked at and some even degrade. There are more reads than writes, and many more static reads than static writes. Reads tend to be to older objects than writes. Young objects do not dominate memory accesses. Shared accesses are dominated by reads. Array accesses tend to be more shared than scalar ones, but scalars tend to have more frequent changes of ownership. Static accesses are massively shared. Volatile accesses are quite rare compared to all accesses, but only somewhat less frequent than locks. Nearly half of all static accesses tend however to be volatile. The chance that an access is shared increases with age. There are many accesses to objects reachable from multiple threads, but few are accessed by multiple threads: reachability is a gross over-estimate of sharing. A non-trivial volume of memory is reachable from statics, which immediately contributes to this over-estimate in nearly single-threaded workloads. Furthermore, a non-trivial volume of memory is never used.

Finally, we thank Doug Lea, Eliot Moss, Andreas Sewe, Pavel Parizek, Thomas Shilling and the anonymous reviewers for their thoughtful comments and suggestions. We are grateful for the support of the EPSRC through grant EP/H026975/1 and NSF 1048398, 1019518, 1019607.

References

[1] ASM project. `http://asm.ow2.org`, 2011.

[2] BTrace. `http://kenai.com/projects/btrace`, 2011.

[3] B. Alpern, C.R. Attanasio et al. Implementing Jalapeño in Java. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 1999.

[4] W. Binder, J. Hulaas and P. Moret. Reengineering standard Java runtime systems through dynamic bytecode instrumentation. In *Source Code Analysis and Manipulation (SCAM)*, 2007.

[5] S.M. Blackburn and K.S. McKinley. Ulterior reference counting: Fast garbage collection without a long wait. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2003.

[6] S.M. Blackburn, R. Garner et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2006.

[7] A. Burns and A.J. WellingS. *Real-Time Systems and Programming Languages: ADA 95, Real-Time Java and Real-Time POSIX*. Addison-Wesley, 3rd edition, 2001.

[8] 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), 2011.

[9] D. Doligez and G. Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *Symposium on Principles of Programming Languages (POPL)*, 1994.

[10] D. Doligez and X. Leroy. A concurrent generational garbage collector for a multi-threaded implementation of ML. In *Symposium on Principles of Programming Languages (POPL)*, 1993.

[11] T. Domani, E.K. Kolodner et al. Thread-local heaps for Java. In *International Symposium on Memory Management (ISMM)*, 2002.

[12] B. Dufour, K. Driesen et al. Dynamic metrics for Java. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2003.

[13] H. Esmaeilzadeh, T. Cao et al. Looking back on the language and hardware revolutions: Measured power, performance and scaling. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[14] L. Gidra, G. Thomas et al. Assessing the scalability of garbage collectors on many cores. In *Programming Languages and Operating Systems (PLOS)*, 2011.

[15] R.E. Jones and A.C. King. A fast analysis for thread-local garbage collection with dynamic class loading. In *Source Code Analysis and Manipulation (SCAM)*, 2005.

[16] R.E. Jones and C. Ryder. A study of Java object demographics. In *International Symposium on Memory Management (ISMM)*, 2008.

[17] R.E. Jones, A.L. Hosking, and J.E.B. Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall, 2011.

[18] T. Liu and E.D. Berger. Sheriff: precise detection and automatic mitigation of false sharing. In *Object Oriented Programming Systems, Languages and Applications (OOPSLA)*, 2011.

[19] S. Marlow and S.L. Peyton Jones. Multicore garbage collection with local heaps. In *International Symposium on Memory Management (ISMM)*, 2011.

[20] K. Shiv, K. Chow et al. SPEC jvm2008 performance characterization. In *SPEC Benchmark Workshop on Computer Performance Evaluation and Benchmarking*, 2009.

[21] B. Steensgaard. Thread-specific heaps for multi-threaded programs. In *International Symposium on Memory Management (ISMM)*, 2000.

[22] L. Valiant. A bridging model for parallel computation. *CACM*, 33(8):103–111, 1990.

[23] P.H. Welch and J.B. Pedersen. Santa Claus: Formal analysis of a process-oriented solution. *ACM Trans. Comput. Syst.*, 32 (4), 2010.

[24] X. Yang, S.M. Blackburn et al. Why nothing matters: the impact of zeroing. In *Object Oriented Programming Systems Languages and applications (OOPSLA)*, 2011.

[25] X. Yang, S.M. Blackburn et al. Barriers reconsidered, friendlier still! In *International Symposium on Memory Management (ISMM)*, 2012.

[26] Y. Zhao, J. Shi et al. Allocation wall: A limiting factor of Java applications on emerging multi-core platforms. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2009.