Adaptive Sampling-Based Profiling Techniques for Optimizing the Distributed JVM Runtime

King Tin Lam, Yang Luo, Cho-Li Wang
Department of Computer Science
The University of Hong Kong
Hong Kong
{ktlam, clwang, yluo}@cs.hku.hk

***Abstract*—Extending the standard Java virtual machine (JVM) for cluster-awareness is a transparent approach to scaling out multithreaded Java applications. While this clustering solution is gaining momentum in recent years, efficient runtime support for fine-grained object sharing over the distributed JVM remains a challenge. The system efficiency is strongly connected to the global object sharing profile that determines the overall communication cost. Once the sharing or correlation between threads is known, access locality can be optimized by collocating highly correlated threads via dynamic thread migrations. Although correlation tracking techniques have been studied in some page-based software DSM systems, they would entail prohibitively high overheads and low accuracy when ported to fine-grained object-based systems. In this paper, we propose a lightweight sampling-based profiling technique for tracking inter-thread sharing. To preserve locality across migrations, we also propose a stack sampling mechanism for profiling the set of objects which are tightly coupled with a migrant thread. Sampling rates in both techniques can vary adaptively to strike a balance between preciseness and overhead. Such adaptive techniques are particularly useful for applications whose sharing patterns could change dynamically. The profiling results can be exploited for effective thread-to-core placement and dynamic load balancing in a distributed object sharing environment. We present the design and preliminary performance result of our distributed JVM with the profiling implemented. Experimental results show that the profiling is able to obtain over 95% accurate global sharing profiles at a cost of only a few percents of execution time increase for fine- to medium-grained applications.**

Keywords-profiling; sampling; correlation tracking; access locality; thread affinity; thread stack; thread migration; dynamic load balancing; object sharing; distributed Java virtual machine; distributed shared memory systems

I. INTRODUCTION

Software distributed shared memory (DSM) systems have turned over a new leaf by breeding various forms of distributed runtimes having a similar programming model. The key ideas developed by decade-long research efforts in DSM have been realized into production-quality systems that are available in recent years. Examples include GigaSpaces [1], Oracle Coherence [2] and Open Terracotta [3]; all these clustering solutions are reaching out to remote memory for global cache benefits. Researchers' interest in the *partitioned global address space (PGAS)* model, a variant of DSM, is underlined by the multibillion HPCS program [4] led by DARPA. The novel languages like X10 [5] and Chapel [6] basically follow the DSM model but ship with far more new constructs for data locality management. Developers can use the constructs to define how shared data are distributed with locality hints among processors. The deliberate constructs grant users more control and yet somewhat impair the programmability goal of the shared memory model.

With significant speed improvement these years [7, 8], Java has made definite inroads into high-performance computing. Given that 4.5 billion devices and 6.5 million developers worldwide are pivoting on Java [9], a handy parallel paradigm is gluing to idiomatic Java programming and offloading the cluster-wise parallelization task of a threaded Java program transparently onto an underlying *distributed Java virtual machine (DJVM)* [10, 11, 12]. The DJVM software layer handles all aspects of low-level clustering for the application through automatic thread-level parallelization and heap-level virtualization of shared memory across the cluster. Ultimate goal of our DJVM design is to provide a *single-system image (SSI)* while attaining scalability comparable to the message-passing model. This performance target is however difficult to achieve. The memory consistency protocol must be well-designed to eliminate all needless remote communications. Advanced features such as a system profiler should also be in place to get most shared data locality or thread affinity well-managed as if PGAS constructs were added in the application program.

Optimizing locality cluster-wide based on a partial sharing profile would be too inaccurate that thrashing of threads and objects among nodes would be resulted. We need to obtain a complete and precise profile of sharing or *correlation* among threads for global optimization. The profile is however difficult to obtain without high overheads. *Passive correlation tracking* (used in [13, 14]) that relies on remote page faults to activate access logging can only capture partial sharing behavior because access to a validated page by other local threads is missed logging. *Active correlation tracking* [15] was proposed to track the sharing information. The term "active" means the system deliberately fakes invalid page states via memory protection and brings about the so-called "correlation faults" regularly to awake the DSM protocol for access logging. However, when such page-based techniques are applied to fine-grained object-oriented sharing systems,

978-1-4244-6443-2/10/$26.00 ©2010 IEEE

tracking overheads will soar to an intolerable level, for the number of instrumented object state checks is much more than page faults. So we need new techniques to track the sharing profiles in a lightweight manner. The key to reducing the overheads is to use *sampling* to limit the correlation faults on a subset of objects in the heap. However, sampling would worsen accuracy in the correlations tracked and we need to balance between accuracy and profiling cost. Simple sampling strategy at a fixed gap is not a decent solution since the gap varies from one application to another. Without an *adaptive* framework that automatically tunes for a balance, users will find it hard to handcraft an appropriate sampling gap for their applications.

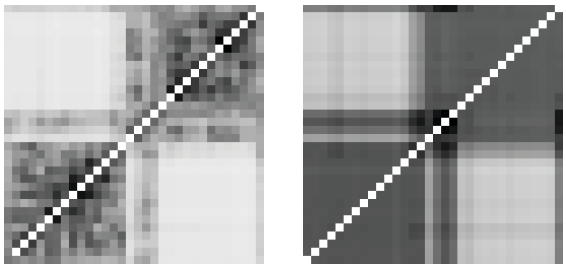
Besides the correlation between thread pairs, the affinity (quantified by access frequency or recency) between a thread and its accessed objects also forms an important system parameter for a runtime with reconfigurable thread placement. Studies in this area usually just consider the *direct* overhead of a thread migration spent on the thread context (stack frames) but ignore the even larger *indirect* overhead spent on remote object faults (analogous to page faults) that follows the migration. Such round-trips can be effectively hidden if the related objects are prefetched along with the migration. While previous work [16, 17] on working-set-based remote memory pre-paging has been done for process migration, determining an effective working set of objects to prefetch for thread migration is much more challenging on similar grounds of huge access tracking overheads.

In this work, we propose a couple of adaptive sampling techniques for tracking thread-thread and thread-object affinity in a lightweight manner. Sampling rates can vary at runtime in both techniques to balance between preciseness and overhead. The contributions of this paper are two-fold. First, we provide a fine-grained active correlation tracking mechanism based on a novel object sampling for improving thread affinity in object-based sharing systems (Section II). Second, we develop a profiling technique based on online stack sampling to monitor the set of objects that are *sticky* to a thread (Section III). The *sticky set* determines the real cost of thread migration covering the predictable remote object faults after the thread migrates. This gives a more accurate cost model for the load balancer to devise profitable thread migrations. Policies or algorithms of making best use of the profiling output for performance gain are indeed a separate hard problem and open to the system engineers (our future work is outlined in Section V).

We implement the proposed methodologies in the *global object space (GOS)* subsystem of the JESSICA2 DJVM [11]. The GOS has been significantly revamped for better implementation of home-based release consistency (HLRC) [18] and barrier synchronization support. We evaluate JESSICA2 in terms of profiling overheads and accuracy (Section IV) on a cluster platform. Experimental results show an average accuracy of 95% obtained at overhead bounded by 10%.

II. THREAD CORRELATION TRACKING

Despite the many advantages such as global cache effect and simpler programming, object sharing over a DSM system does involve additional overheads such as remote locking, object faulting and update propagation, etc. These costs highly depend on the relative locations of threads and shared objects. For home-based protocols, if a thread is placed at the home node of a shared object, its access to the object would be the most efficient (without twin-diff and messaging overheads). Locality is a relative property. Relocating home of one object for locality of one thread may sacrifice locality of other threads accessing it. Load balancing is another concern. Overloading a node by moving to it too many threads causes adverse slowdown, shadowing the locality benefit. Therefore, thread placement must consider global strategies that strive to keep locality optimal for the majority of threads most of the time. Our study would further classify the relative locality or affinity into three types: (1) *thread-thread* or *inter-thread affinity*; (2) *thread-object affinity* and (3) *inter-object affinity*. Inter-thread affinity can be attained by thread migration to collocate a pair of threads sharing large amount of data on the same node. Thread-object affinity measures the access locality by a thread to an object and can be improved either by thread migration or object home migration. Inter-object affinity reflects the correlation between objects under a graph of connectivity and can be dealt with object prefetching and home migration. This paper focuses on (1) and (2) while (3) is studied in another paper [19] in which we introduced *access path analysis* as a profiling technique for the proper scope of object prefetch.



(a) Inherent pattern (b) Induced pattern

Figure 1. False sharing effect on correlation tracking preciseness: clues about precise inter-thread correlations are lost

Effective thread placement is vital to not only distributed systems but also multicore processors in view of their shared cache architecture. Placing threads accessing different data streams to cores in close proximity may cause cache contention and thrashing. On the contrary, placing highly correlated threads to be within intra-core will make their object sharing done speedily over shared L2 cache. With an efficient way to obtain inter-thread correlation, the runtime system can be guided properly for dynamic thread placement.

Employing localized thread placement strategies may not improve the system performance and even cause threads to thrash between nodes due to incomplete sharing profiles. Thus, we need to collect global sharing statistics and deduce a so-called *thread correlation map (TCM)*, a 2D histogram of shared data volume between each pair of threads. Active correlation tracking [15] has been studied in page-based DSM systems for such a purpose, but is not quite useful to fine-grained applications because it can only reveal the *induced* sharing pattern rather than the application's *inherent* pattern after the effect of false-sharing. In Fig. 1, we illus-

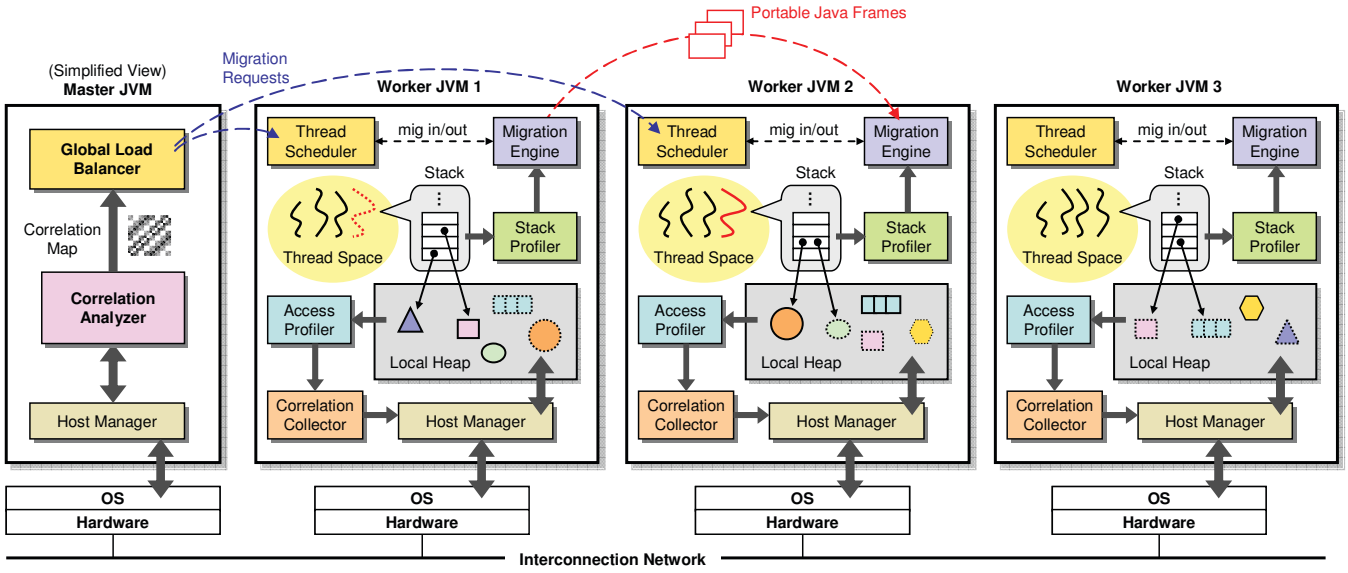


Figure 2. System architecture of JESSICA2 DJVM with profiling

trate this point with two correlation maps showing the inherent and induced sharing patterns of the same program, Barnes-Hut (32 threads, 4K bodies, distance 7.0) in which each thread is responsible for simulating movement of a continuous chunk of bodies in two galaxies. Threads for computing the interaction between bodies within the same galaxy will exhibit much higher data locality than those bodies across galaxies. Indeed, data partitioning algorithms like *costzone* [20] yield even higher locality between adjacent threads. While the program's inherent pattern is shown in (a) obtained by simulation (log inserted at every object access), the induced pattern shown in (b) contains very little hint of locality between threads of the same galaxy due to serious false sharing. Respecting the original application nature calls for a fine-grained version of active correlation tracking.

We present our fine-grained active correlation tracking techniques based on adaptive object sampling in the following subsections.

A. *Fine-Grained Active Correlation Tracking*

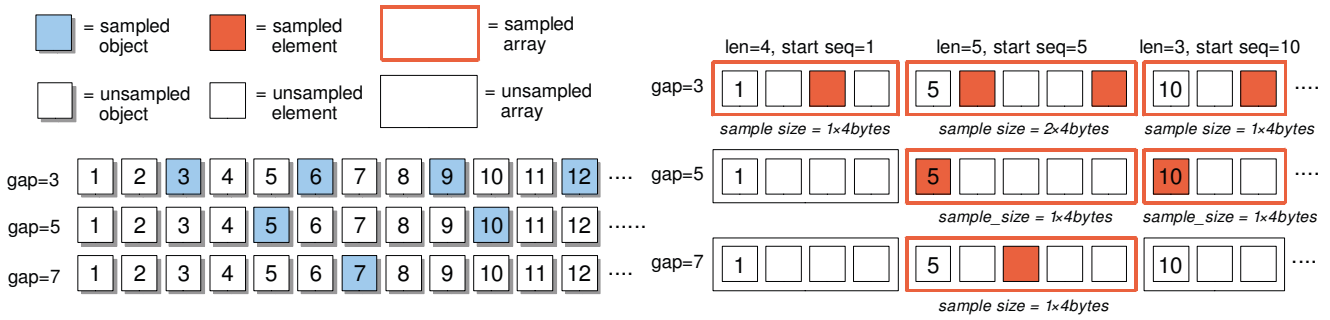
Fig. 2 shows the system architecture of the JESSICA2 DJVM with profiling subsystems introduced. As mentioned, our consistency protocol is home-based. Object home copies (drawn in solid lines) reside in the nodes which are the first to create them. To minimize remote access, shared objects retrieved from home nodes are replicated as cache copies (in dashed lines) in the local heap of the current thread. Cache copies are invalidated at lock time if there are updates made by remote threads happened before the lock. By software checks injected per read/write, access to an invalidated cache of an object will fault in the latest copy from its home.

To estimate a system-wide sharing profile, the first step is to track every thread's reads and writes on objects, forming an *object access list (OAL)* for each thread. Care must be taken in this step because logging every object access will penalize the common case. By means of the *at-most-once* property of HLRC protocol, no matter how many times a thread accesses an shared object, access log for the object can be done only once per interval across synchronizations.

In JESSICA2, object state check is inlined to every access bytecode operation through the JIT compiler. The state is stored as 2 bits somewhere in the object header. Upon opening a new interval, shared objects (only those accessed in the last interval by the thread) will be reset to false-invalid state to enable tracking on them regardless of their real status (which is now stored in a separate field). When accessing a shared object, access fault will be handled by the GOS service routine to log the access into a per-interval record, cancel its false-invalid state, and maintain object consistency according to its real state. On closing an interval, OALs (i.e. accessed object id and size) will be collected and packed along with the interval context (delimited by start and end bytecode PC) into a jumbo message to be sent to central coordinator (the master JVM in Fig. 2) running the correlation computing daemon. This message is piggybacked on lock or barrier request if they are going to the same destination. If enough intervals are gathered, the daemon will process the OALs, reorganize the per-thread lists to per-object lists of thread ids, and constructs the TCM by accruing bytes of accessed objects in common for each thread pair. Given M objects shared by N threads, OAL reorganization and TCM building take $O(MN)$ and $O(MN^2)$ time respectively. It is clear that computing TCMs for large M can grow into a scalability bottleneck in the system, leading us to think of the sampling approach to reducing M.

B. *Adaptive Object Sampling*

Sampling is a statistical process of selecting a subset of units, i.e. samples, from a population of interest so that by studying the sample we may fairly generalize our results back to the population which in our case is the entire JVM heap. Each object is given a tag marked as "sampled" or "unsampled" upon its creation. A good object sampling scheme should take samples uniformly over the heap. The



(a) Object sampling (b) Array sampling

Figure 3. Sampling at different gaps

simplest sampling scheme is probably to sample objects at the same rate and logs their object sizes which reflect communication costs. However, this simple sampling may not yield precise enough result since different classes of objects may vary a lot in size, access and sharing behavior. Thus, we aim for different sampling rates for different classes. (Note: Different classes may inherit from the same superclass but could show different access or sharing behaviors, so we store the sampling-specific metadata like sampling gap as close to subclasses as possible. We differentiate at class level instead of method level for simplicity though allocation site is a more precise hint of object's behavior in other studies [21].)

1) Class-based Sampling Rate

Choosing an appropriate sampling rate for a specific class needs a careful tradeoff between accuracy and overhead. Without dynamically changing the sampling rate and probing the perceived accuracy, we may never know whether we could still decrease the rate to reduce the overhead further. To allow this to happen, we expand the traditional one-bit sampling tag into a sequence number (half-word for memory efficiency), which is unique among objects within the same class. Sampling rate is then given by a variable parameter known as the sampling gap. An object will be taken as a sample only if its sequence number is divisible by the current sampling gap which is defined at class level.

Each class has a *nominal* sampling gap typically in powers of 2 and we will find a prime number nearest to the nominal to be the *real* sampling gap. For example, 31, 67 and 127 would be chosen as the real sampling gaps for nominal sampling gaps of 32, 64 and 128 respectively. Using prime numbers is necessary in our scheme to avoid non-uniform sampling due to potential cyclic allocation behaviors in some applications. Fig. 3 (a) shows an example. Each box represents an object instance carrying an allocated sequence number. A specific object might be sampled or unsampled under the current sampling gap according to this number.

We adopt the notation nX to denote the sampling rate w.r.t the page size. For example, 8X means "sampling eight objects per memory page". For a class of size s, sampling at rate nX has to set the sampling gap to be $S_P/(s\times n)$, where S_P is the page size (usually 4KB). Sampling rate can vary dynamically from at least 1X to 2X, 4X … until full sampling is reached. The remaining problem is how to determine if the current sampling rate is precise enough or in other words how to measure the accuracy of sampling.

2) Sampling Accuracy

For a system of N threads, a correlation map is an $N\times N$ matrix. Let $A = [a_{ij}]_{N\times N}$ and $B = [b_{ij}]_{N\times N}$ be two correlation maps. We measure accuracy in terms of the difference, i.e. error, between the two matrices. Formulae (1) and (2) below measure the distance between A and B by Euclidean norm and absolute value respectively.

$$E_{EUC} = \frac{\sqrt{\sum_{i=1}^{N}\sum_{j=1}^{N}(a_{ij}-b_{ij})^2}}{\sqrt{\sum_{i=1}^{N}\sum_{j=1}^{N}(b_{ij})^2}} \quad (1)$$

$$E_{ABS} = \frac{\sum_{i=1}^{N}\sum_{j=1}^{N}\left|a_{ij}-b_{ij}\right|}{\sum_{i=1}^{N}\sum_{j=1}^{N}\left|b_{ij}\right|} \quad (2)$$

If B is the result from full sampling, while A is not, we call this *absolute accuracy*. If both A and B are not from full sampling and A samples less frequently than B, we call this *relative accuracy*. While each sampling rate must be evaluated with absolute accuracy, decisions on dynamic sampling rate changes can only be made from limited knowledge of relative accuracy. Therefore, we would study the relation between these two accuracy metrics and evaluate if relative accuracy can yield correct decisions. The basic approach to reaching an optimal sampling rate is to begin with a rough sampling rate, increase it stepwise (by shortening the sampling gap) and compare the distance between the successive correlation matrices. If their distance is small enough (converge to be within some predefined threshold), we stop at the underlying sampling gap. The central coordinator that collects OALs from all threads will decide whether the current sampling rate needs a change. Upon receiving a change notice for a specific class, every thread will iterate through all objects of that class it caches locally, check with their sequence numbers, and sample or desample each of them accordingly to align with the new rate. Resampling does waste some CPU cycles but is useful to prevent those objects sampled at previous rates from accumulating to make tracking overhead ever-increasing. In our benchmarks, it usually takes no more than 0.1% of total CPU time.

3) Sampling of Arrays

The case of array sampling needs special treatment to address non-uniform sampling and correlation bias. First, arrays can vary a lot in their lengths. If we sample arrays like

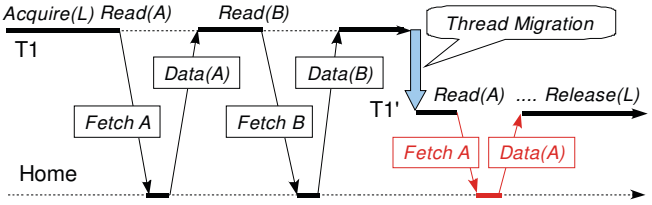


Figure 4. Thread migration's induced cost

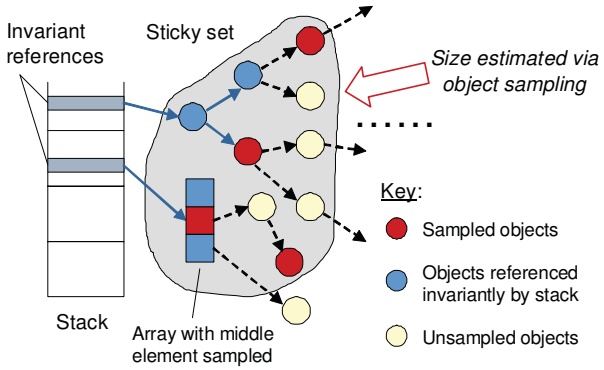


Figure 5. Stack invariant references

the way for ordinary objects, a miss of sampling a large array will leave a large portion of the heap unsampled. Second, if we log the array size to be the sample's size, correlation results obtained via sampling will be largely biased towards large arrays, resulting in skewed correlation. For example, *T*1 and *T*2 share a small array *A* while *T*2 and *T*3 share a large array *B* accessing different element ranges, the correlation between *T*2 and *T*3 will always be overestimated. On the other hand, one could argue that such a bias is appropriate for large arrays since they incur higher communication costs in a common home-based protocol implementation which handle object faults by bringing the whole object from home. However, since array sizes can be larger than a page, allowing the bias would make the correlation result vulnerable to false sharing.

We use an amortization scheme to ameliorate these effects by regarding each array as a group of objects no matter the array element type. So every element has its own sequence number. As these numbers are continuous, for each array instance, we only need to save the first element's sequence number and derive the others by adding the array index. Fig. 3 (b) shows an example of sampling arrays of the same class with various lengths. An array is sampled only if at least one of its elements is *logically* sampled. We say "logically" since per-element sampling is needless and we can easily get the number of sampled elements from dividing the array size by current sampling gap. To handle the bias, when a sampled array is accessed, we consider all its sampled elements accessed and log an *amortized* sample's size = sampled elements × element type size for the array when computing the correlation map. The overall scheme would make sampling both statistically uniform and unbiased whatever sampling gap changes.

III. Thread Migration Cost Modeling

As mentioned, the affinity between a thread and its accessed objects is another crucial factor that determines the system performance. Thread migration is a mechanism to improve the data access locality by moving computation to the data. While a thread context is usually cheaper to migrate than the data (object graphs and arrays), the actual migration cost could be much larger than just sending out the thread's stack because of the implicit cost of remote object faults happened thereafter. For a better model of the thread migration cost, we define the *sticky set (SS)* of a migrant thread candidate as a set of objects that will predictably cause remote object faults after the thread migrates. The term "sticky" implies a strong correlation between the thread and the set. So if the thread moves out without the sticky set prefetched along with, it will see successive object misses causing remote roundtrips. Sticky set is a subset of the working set of a thread (which could be too large to send) but is more difficult to determine because capturing access recency alone may not be enough. In Fig. 4, thread *T*1 fetches object *A* and *B* during an interval, each only once. But *A* is accessed frequently while B is accessed only once. If *T*1 migrates to a new node, being *T*1', during the same interval, it will need to fetch *A* again but not *B*. In this case, we can see a thread's local access frequency to objects within an interval does matter although correlation tracking can skip it.

We can observe that objects in a thread's sticky set have the following properties: (1) they have been accessed before thread migration within the same interval; (2) they will still be accessed after thread migration also within the same interval. Only these objects will contribute to the total cost of thread migration for they are fetched twice within a single interval. Normally these objects are constantly accessed throughout the whole interval, so they can be discovered by monitoring object access patterns. It should be noticed that our definition is specific to relaxed memory models like LRC [22] (Lazy Release Consistency) and ScC [23] (Scope Consistency), which have the concept of intervals and the at-most-once property.

A. Profiling Migrant Thread's Sticky Set

We estimate the sticky set by a two-way profiling strategy. First, we make repeated calls of adaptive object sampling within an interval to capture access frequency statistics on sampled objects so as to obtain an approximate size of the sticky set and the class-level composition of the objects in the set, i.e. how many bytes are accumulated in the access log for each class of sampled objects accessed by the thread of interest. We denote this metadata by the term *sticky-set footprint*. Basically, one can design a load balancing policy that weighs the gain from a thread migration against the messaging cost proportional to such a footprint.

However, this process doesn't log precise access frequency and covers only sampled objects, missing those un-

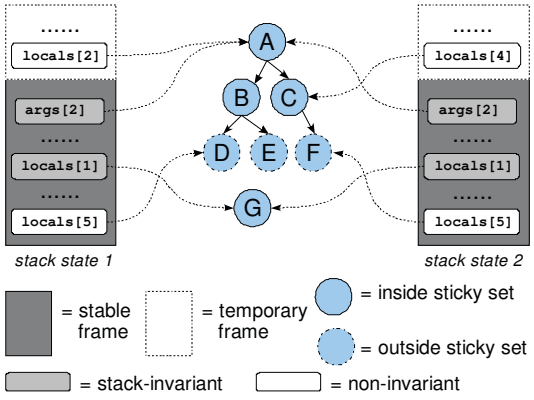


Figure 6. Example of stack-invariants

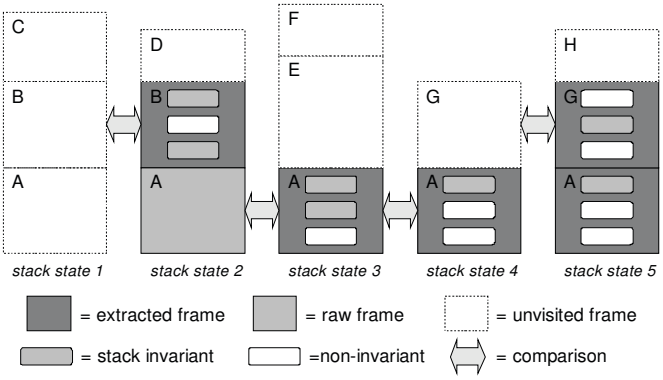


Figure 7. Lazy stack sample comparison

sampled objects that are actually being accessed frequently. So an online stack sampling-based mechanism is employed to back up the accuracy. Stack sampling refers to taking snapshots of the stack frames of a thread periodically. In this case, a sample means a stack image. Through such sampling, we can discover some object references which steadily persist across the taken samples. These *invariant* object references clue us in on the entry points of the underlying sticky set from which we can start prefetching over the object graphs until the bytes prefetched reaches some threshold over the sticky-set footprint estimated by object sampling. This concept is depicted in Fig. 5. This bilateral strategy exploits a mix of heap-sampled access frequency and stack-sampled access recency to derive the thread-object correlation. So although results are tapped from sampling which is limited or speculative, they reconcile or complement one another to give improved accuracy.

The entire SS-profiling process is of three steps:

1) Sticky Set Footprinting:

This step estimates the size of each class of frequently accessed objects to be included in the set by sampling and tracking objects in the heap repeatedly. Compared with correlation tracking that is done at most once for each sample per HLRC interval, repeated tracking over heap objects may impose higher overhead and require tradeoff between accuracy and cost again. Thus, we put a lower bound on object sampling gap and a timer for on and off tracking phases.

2) Mining for Stack-invariants:

This step uses stack sampling to discover some object references that consistently show up on the stack. Since JVM is a stack machine that every bytecode can only access its operands via the current stack frame, we can exploit the following properties to discover hints about a thread's sticky set:

a) If a thread wants to access objects of its sticky set, it must start from a reference on its Java stack.

b) Temporary or transient frames are unlikely to contain key references to a thread's sticky set. In a real-life Java program, many top frames may exist temporarily for a very short time, while the bottom ones could last much longer. References in transient frames will soon be lost after the frames are popped.

c) Stack invariant references, remaining on stack for a long time constantly, are valuable hints about the sticky set. The first reason is that varying references are always obtained by following invariant references, directly or indirectly. Second, such invariant references are often denoting the entry points of some collective data structures like a linked list's head, a tree's root, or a hash table's entry array that are frequently traversed.

To see how invariant references of a stack are related to its sticky set, Fig. 6 shows two stack snapshots taken at distinct instants. In each state, the top frames are temporary while the bottom frames are not. The top frames could contain some references to some objects in sticky set but are often popped and replaced very soon. The bottom frames, however, contain both invariant references (`args[2]`, `locals[1]`) constantly pointing to the sticky set and non-invariant references (`locals[5]`) used by some temporary references during the traversal of some data structures. In a word, stack invariants are the likely entry points of a sticky set but some sticky objects, like *B* and *C*, could be prefetched only by following the other front-side sticky objects.

3) Sticky Set Resolution:

SS resolution traces the stack-invariants for selecting objects (regardless of sampled or unsampled) to be the SS candidates to prefetch until the amount of reachable sampled objects hits the estimated SS footprint. Resolution is invoked lazily only when a thread migration event is out. The resolution algorithm has a few special points over the usual connectivity-based object prefetching. First, we may obtain a number of stack-invariant references from stack sampling, acting as multiple starting points for prefetching. If we cannot find enough objects by following a stack-invariant reference, we can switch to the others to continue the tracing. A heuristic here is to always start from topmost stack-invariants because they tend to be more recent than the bottom ones. Second, the sampled objects can serve as some *landmark* objects (red ones in Fig. 5) to avoid prefetching in wrong directions. Since sampled objects are scattered over the underlying SS object graph, during the selection process, we can check, if an adequate number of landmark objects has been met. If not, current prefetching might be in a wrong direction, so we should stop and switch to other paths. For example, if we sample one per 30 objects for a specific class, we will stop current prefetching if we have not seen any landmark for $t \times 30$ objects of that class where t is a tolerance

parameter (>1) to address imperfect sampling uniformity. Third, the resolution is done on per-class basis as we know the sticky set's composition. So we can prefetch each type of sticky objects until the per-class estimated footprint is hit.

B. Adaptive Stack Sampling

The pseudocode in Fig. 8 shows our stack sampling algorithm. The frame content extraction process is summarized here. For a given stack, we start from its top and trace down, finding out all Java frames, as well as each frame's %EBP and %EIP (x86). Then for each Java frame, we find out its corresponding Java method by querying Java's reflection system (line 21) and get its layout (or slots), and then use our stack layout knowledge to extract each slot's content (line 24). For each slot's content, we use the JVM garbage collection interface to check if it is a valid object pointer. After all slots have been checked, we obtain a current sample of that frame. To find out stack invariants, we need to find an old sample of the same frame (line 8) and compare with the new one. Such extraction and comparison can have serious performance impacts on the runtime. So we design the following aggressive optimizations.

1) Timer-based stack sampling:

Execution is split into overhead-free and sampling-enabled phases by using a timer (sampling gap here refers to the time gap between activating stack sampling).

2) Two-phase stack scanning:

To avoid expensive overhead of scanning temporary frames, we add a flag visit to each frame and set it once the frame has been sampled. Our JIT compiler is hacked to ensure a frame's visit flag will always be cleared in every Java method's prologue. In the top-down phase, we start from the top frame of our current stack, trace down until we hit the first visited frame. For the first visited frame, we sample it and compare with its previous sample, which must have been created when its visited flag was set. Since we safely know that all frames below are untouched between these two samples, we do not need to trace down further. In the bottom up phase, we go backward until the top unvisited frame, extract the first sample for each frame, and set its visited flag.

3) Lazy extraction:

Upon the first-time visit of a frame, we just capture it as a raw sample in its native format (line 16) and delay extracting its content until it is visited next time (line 10). If it is not visited for the second time, it will be discarded on the next stack sampling. This avoids extraction cost for almost all temporary frames on the top, because stack sampling gap is normally at least several milliseconds.

4) Sample comparison by probing:

When comparing two captured samples, we always use the old one to probe into the new one (line 12), by comparing each slot remained in the old sample with its corresponding slot in the new one. This helps reduce comparison cost for frequently visited frames because the old sample is usually much smaller as non-reference and non-invariant slots have been discarded in previous samples.

We demonstrate our adaptive stack sampling in Fig. 7. Initially (state 1), all three frames are unvisited, and we just store them in raw form. In the next sample (state 2), frame *C* is gone, with frame *D* on the top. Frame *B* was compared with its last sample to find out invariants, but frame *A* is still in its raw state. In the next sample (state 3) frame *B* and *D* are gone, while frame *E* and *F* are now on the top. Now we visit frame *A* for the second time, so we process the saved raw sample and compare it with the new sample. In the next sample (state 4), frame *E* and *F* are gone, and we continue compare *A*, further removing non-invariants from the sample. In the last sample (state 5), frame *G* survives, so we process the old sample and compare it with the new one, leaving frame *A* untouched.

```
1   SAMPLE-STACK(thread)
2      // top-down phase
3      frame ← TOP-FRAME(thread)
4      while (not VISITED(frame)) do
5         frame ← NEXT-FRAME(frame, thread)
6      end while
7      // process the first visited frame
8      sample_old ← GET-OLD-SAMPLE(thread, frame)
9      if (IS-RAW-SAMPLE(sample_old)) then
10        CONVERT-RAW-SAMPLE(sample_old)   // extract frame content
11     end if
12     COMPARE-BY-PROBING(sample_old, frame)
13     // bottom-up phase
14     while (frame ≠ null) do
15        SET-VISITED(frame)
16        sample_raw ← SAMPLE-FRAME-RAW(frame)
17        ADD-SAMPLE(thread, frame, sample_raw)
18        frame ← PREV-FRAME(frame, thread)
19     end while
20  COMPARE-BY-PROBING(sample_old, frame)
21     method ← GET-METHOD-BY-PC(NATIVE-PC(frame))
22     for each slot in GET-SLOTS(sample_old) do
23        ref_old ← GET-SLOT-FROM-SAMPLE(sample_old)
24        ref_new ← GET-STACK-SLOT(frame, slot)
25        if (ref_old ≠ ref_new) then
26           REMOVE-SLOT-FROM-SAMPLE(sample_old, slot)
27        end if
28     end for
```

Figure 8. Adaptive stack sampling algorithm

IV. Performance Evaluation

We implement the proposed profiling techniques into the JESSICA2 distributed JVM and evaluate the system performance after different effects of profiling are enabled. Our experiments are conducted on the HKU Gideon 300 Cluster [24]. Hardware specification of a node is as follows: Intel Pentium 4 2GHz processor, 512MB DDR RAM, 40GB IDE hard disk and Fast Ethernet network adapter. As a proof of concept, we evaluate the enhanced system on eight nodes. Our benchmark programs are ported from SPLASH-2 [25] to Java and described as follows. Table I summarizes the problem sizes used and the sharing properties of the applications.

TABLE I. Application benchmark characteristics

Benchmark	Problem Size		Sharing	
	Data set	*Rounds*	*Granularity*	*Object size*
SOR	2K × 2K	10	Coarse	each row at least several KB
Barnes-Hut	4K bodies	5	Fine	each body less than 100 bytes
Water-Spatial	512 molecules	5	Medium	each molecule about 512 bytes

1) SOR: an iterative linear algebra kernel executing the red-black successive over-relaxation (SOR) method on a matrix. SOR exhibits a near-neighbor regular sharing pattern with large object granularity (each row is at least a few KB) and modestly intensive computation.

2) Barnes-Hut: an N-Body simulation using hierarchical methods. Barnes-Hut shows an irregular sharing pattern with some locality (which cannot be discovered in page-based systems), fine-grained object sharing and moderate compute-intensiveness.

3) Water-Spatial: a molecule dynamics application, simulating interactions between groups of water molecules. Runtime properties include near-neighbor 3D-box sharing patterns with medium granularity, intensive computations and evolving load distribution.

We will present experimental results in terms of overheads and accuracy measures of the two proposed access profiling techniques. Note: All the reported overheads are the result with profiling enabled throughout the entire execution. This is unnecessary for many applications whose sharing behaviors are rather static. Overheads can be much smaller by shutting the profiler after a short profiling phase is over.

A. Correlation Tracking via Adaptive Object Sampling

1) Overhead

There are three types of overheads of correlation tracking: (O1) CPU cost for generating OALs; (O2) network overhead of gathering OALs to a central node; (O3) CPU cost for constructing the TCM from OALs. Benchmarking methodologies for each overhead are as follows. To isolate O1 from other effects, we use a single thread for each application and disable transfer of OALs over the network. To measure O2, first, we measure the volume of OAL traffic and compare it with the volume of object data we have transferred. Second, we compare the total execution time with and without correlation tracking. We use eight nodes, running a single thread each, to avoid uncertainty from per-node multithreading and congestion. Obtaining O3 is trivial as this step is performed on a central server. Each type of overhead is measured at various sampling frequencies, from 1X, 4X, 16X to full sampling. It should be noted that some configurations like 16X might not apply to medium-to-coarse grained applications. All these experiments are performed with stack sampling and thread migration disabled while optimizations of object prefetching and home migration are enabled.

TABLE II. OVERHEAD OF OAL COLLECTION

Benchmark	Execution Time (ms)				
	No Correl. Tracking	*Sampling = On (Collect OALs)*			
		1X	*4X*	*16X*	*Full*
SOR	24250	N/A	N/A	N/A	24360 (0.45%)
Barnes-Hut	53250	52636 (-1.15%)	52742 (-0.96%)	53354 (0.20%)	53844 (1.12%)
Water-Spatial	29461	29507 (0.15%)	29545 (0.28%)	N/A	29717 (0.87%)

Table II and III show all the overhead benchmarking results. First, the extra CPU time spent on collecting OALs at various sampling rates for each application can be found in Table II. It is clear that this overhead is minimal. For the most fine-grained application Barnes-Hut, this overhead is merely around 1% of the total execution time at full sampling. This verifies that our method of setting fake invalid object states across HLRC intervals is much more lightweight compared to page-based DSM systems relying on page faults. The abnormal cases that execution times with sampling enabled get even shorter are reproducible and due to the fact that our implementation has somehow modified the internal memory management system of Kaffe [26], the base JVM of JESSICA2, getting the common case speeded up slightly.

The additional time spent on transferring OALs can be observed on columns 3 to 6 of Table III. With OAL transfer enabled, correlation tracking is of much more noticeable latency but is still tolerable if full sampling is not used. The increase in protocol message volume is shown on columns 8 to 11. Compared with the total GOS traffic (column 7), message volume rise due to OAL traffic is about 2-4% for sampling rate under 16X but soars to 8-22% at full sampling. In particular for SOR, it consists of large arrays that are being tracked although every thread accesses a different portion for most of the time. That is why SOR uses up 20% more bandwidth for transferring OALs than the other two applications with finer object granularity. Since OALs are communicated only at closing of HLRC intervals, such bandwidth consumption is rather bursty. With per-node multithreading as well, this cost does not proportionally reflect on the total execution time increase.

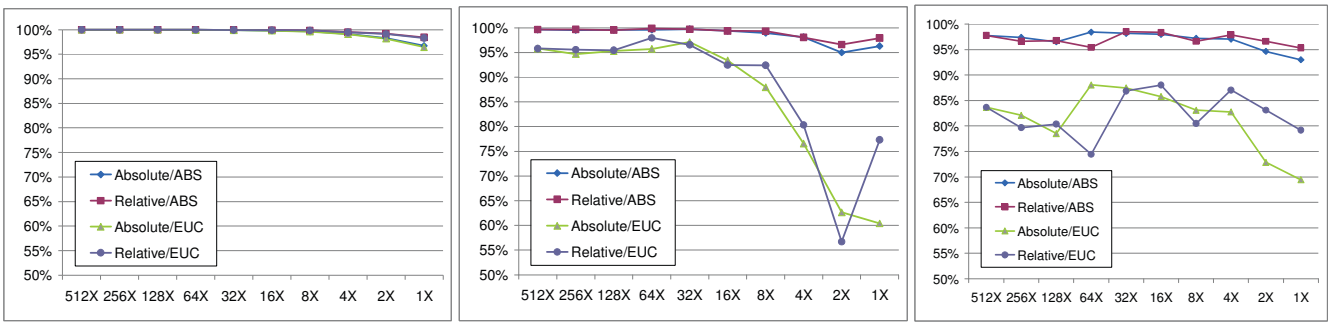
The CPU overhead for computing the TCM from the collected OALs is shown on the rightmost columns in Table III. Clearly, this overhead is among the most severe. Currently we use a dedicated machine to perform this computation, so that total execution time is not affected. For the same dataset size, if the DJVM scales out with more nodes, each iteration will finish sooner making the TCM construction time apparent. Adaptive sampling is useful in this case to lower such overhead by tuning down the sampling rate on demand.

2) Accuracy

For each application, we start from 1024X (i.e. full sampling for page size = 4KB; word size = 4bytes, the smallest possible object size) and halve the maximum rate of each sampled class across every iteration until reaching 1X. We use 16 threads for each application. The result of correlation tracking accuracy is shown in Fig. 9. There are four curves on each figure corresponding to the absolute accuracy and relative accuracy based on Euclidean distance (EUC) and absolute distance (ABS). It is clear that accuracy measured by absolute distance is more stable and consistently outperforms Euclidean distance for all benchmarks. This confirms that absolute distance suggests the maximum deviation between inter-thread communication estimations. So we will use absolute distance exclusively in the remaining experiments. Regarding absolute vs. relative accuracy, Fig. 9 also shows that there is no significant difference between them. We can mostly use relative accuracy as an indicator for ad-

TABLE III. CORRELATION TRACKING OVERHEADS

Benchmark	Execution Time (ms)					GOS Message Volume (KB)	OAL Message Volume (KB)				TCM Computing Time (ms)			
	No Correl. Tracking	*With Correl. Tracking (Collect + Send OALs)*												
		1X	*4X*	*16X*	*Full*		*1X*	*4X*	*16X*	*Full*	*1X*	*4X*	*16X*	*Full*
SOR	3954	N/A	N/A	N/A	4035 (2.04%)	4491	N/A	N/A	N/A	990 (22.05%)	N/A	N/A	N/A	870 (22.00%)
Barnes-Hut	19557	19426 (-0.67%)	19712 (0.79%)	19824 (1.36%)	20805 (6.38%)	60130	140 (0.23%)	525 (0.87%)	2310 (3.84%)	8309 (13.82%)	1568 (8.02%)	1683 (8.61%)	2327 (11.90%)	4609 (23.57%)
Water-Spatial	7942	8186 (3.07%)	8252 (3.90%)	N/A	8340 (5.01%)	31240	828 (2.65%)	879 (2.81%)	N/A	2589 (8.29%)	323 (4.07%)	347 (4.37%)	N/A	749 (9.43%)



(a) SOR (b) Barnes-Hit (c) Water-Spatial

Figure 9. Accuracy of correlation tracking with adaptive object sampling

justing sampling rate. Overall speaking, our result is very positive – almost all sampling rates show at least 95% accuracy – showing our adaptive sampling does not lose the preciseness and would be helpful to making load balancing decision hereafter.

B. *Sticky Set Profiling via Stack Sampling*

1) *Overhead*

The profiling cost of a thread's SS footprint consists of two components: (C1) the CPU cost of performing stack sampling for locating the entry references from where to search for sticky-set candidates; (C2) the CPU cost of repetitive sampling over heap objects for SS footprinting. As these two components are independent of each other, their evaluation is separately done by the below methodology. For (C1), we run the applications with a single thread with object sampling and correlation tracking disabled. The stack sampling gap is varied from 4ms to 16ms to observe how the overhead changes accordingly. For (C2), again only a single thread is used with stack sampling and correlation tracking disabled.

Table IV (columns 4 to 7) shows the stack sampling overhead. We can see this overhead is negligible for SOR and Water-Spatial and slightly higher for Barnes-Hut for it has recursive method calls during octree traversal. Lazy frame extraction and comparison performs better than the immediate counterpart in almost all cases except one (Barnes-Hut; 16ms), showing the effectiveness of such a light-weight technique. We will use 16ms sampling gap with lazy extraction throughout the remaining context.

Table IV (columns 8-11) shows the runtime cost of sticky-set footprinting. Full sampling on heap objects in this case is apparently too costly. Slowing down the sampling rate to 4X is seen effective for trimming down the overhead for fine-grained applications (Barnes-Hut and Water-Spatial) but has no effect on SOR. The reason is SOR just contains arrays sized in range of KB that are bigger than the page size, so effectively every of them will be sampled. By the other approach, lowering profiling frequency with a timer is also found effective for reducing overhead. Sampling at 4X with the 100ms time gap makes the cost minimal.

Table IV (last column) shows the CPU overhead of sticky set resolution. We obtain such measure in an ad hoc manner by eagerly carrying out this operation at the end of each HLRC interval and the time difference as shown is indeed reflecting the extra time spent on picking up sticky-set objects during each HLRC interval. Since invoking sticky set resolution is only needed at thread migration time, this cost vanishes across most HLRC intervals and is regarded as part of the overall cost of a thread migration.

2) *Accuracy*

In this experiment, we would assess the impact of sampling frequency on the estimated sticky-set footprint's accuracy. We use 8 threads for each application, profile the footprint via object sampling at 4X, and compute the average difference between accuracies taken at 4X and full sampling. It should be noted that even at full sampling, the footprint is still an estimation, or relative accuracy, only since the absolute accuracy can only be obtained by driving a thread to migrate and inspect the changes in the DSM protocol traffic. This is difficult and unstable as threads might be made to migrate at any time.

Table V. shows the sticky-set footprinting results compiled as a class-level composition suggesting how many

TABLE V. OVERHEAD OF STICKY-SET FOOTPRINT PROFILING

Bench-mark	Data Set Size	Baseline Execution Time	+ Stack Sampling Overhead				+ Sticky-set Footprinting Overhead				+ Sticky-set Resolution Overhead
			Immediate Extraction		*Lazy Extraction*		*Nonstop*		*Timer-based (100ms)*		
			4ms	*16ms*	*4ms*	*16ms*	*4X*	*Full*	*4X*	*Full*	
SOR	1K×1K	6201	6216 (0.24%)	6207 (0.10%)	6211 (0.17%)	6206 (0.08%)	6714 (8.28%)	6707 (8.17%)	6519 (5.13%)	6480 (4.50%)	6639 (1.85%)
Barnes-Hut	4K	93857	94947 (1.16%)	94657 (0.85%)	94697 (0.89%)	95209 (1.44%)	98968 (5.45%)	102190 (8.88%)	93649 (-0.22%)	102334 (9.03%)	97585 (4.20%)
Water-Spatial	512	59105	59232 (0.21%)	59161 (0.09%)	59209 (0.17%)	59124 (0.03%)	59834 (1.23%)	61985 (4.87%)	59501 (0.67%)	60313 (2.04%)	60002 (0.84%)

bytes of shared objects in each class would be sticky to the thread being profiled. SOR achieves a perfect result for the same reason mentioned above that it indeed runs at full sampling. Barnes-Hut and Water-Spatial achieved less perfect results but all classes are consistently over 92% accurate.

TABLE IV. ACCURACY OF STICKY-SET FOOTPRINT

Benchmark	Data Set Size	Class	Average SS Footprint at Full Sampling (bytes)	Average Diff. at 4X Sampling (bytes)	Accuracy
SOR	1K×1K	double[]	2018016	0	100.00%
Barnes-Hut	4K	Body	229376	672	99.71%
		Body[]	47264	3108	93.42%
		Leaf	76804	104	99.86%
		Vect3	130627	9457	92.76%
Water-Spatial	512	double[]	43032	508	98.82%

V. RELATED WORK

Active correlation tracking was first proposed in D-CVM [15, 27] and extended in later work [28]. The system deliberately disables preemptive thread scheduling and sets each page to be invalid for invoking access logging. Due to lack of thread preemption and more page faults, their performance slowdown is much more significant than ours. D-CVM can only make thread migration decisions based on induced correlation map that is less useful for fine-grained programs. Our method on the other hand can detect inherent sharing patterns of fine-grained programs, giving more precise correlation input to global thread scheduling. Second, our sticky set profiling technique can model the thread-object affinity and suggest a right amount of prefetching to save most indirect costs of remote object faults after migration.

Our object sampling mechanism bears some similarity to those studied in single-machine JVM research but with very different goals and resource constraints. In [29], sampling was used to characterize object allocation behavior, predicting object lifetimes. Their profiling result is mainly used to assist pretenuring for improving GC performance. In our case, sampling is used to track and estimate sharing profile; our profiling result is mainly used to direct thread migrations. In our system, space constraint is much tighter since we cannot store or transfer too much sampling result. Thus, we have to start with a wide sampling gap (actually 4KB) and dynamically adjust it only when accuracy is not enough. In contrast, a much smaller sampling gap (256B) was chosen and fixed in [29].

Our stack sampling is similar to [30] which was used for dynamic profiling in IBM's JVM bearing a very different goal from us. In [30], information from dynamic profiling is only used to build a Partial Calling Context Tree (PCCT), which is inquired by the JIT compiler for adaptive optimizations. Such profiling only needs function caller and callee's addresses. On the other hand, in order to locate stack invariant references, we must extract and inspect each thread's frame content, which is more heavyweight and cannot be performed very frequently. It should be noticed that stack machine is only defined conceptually in the JVM specification. Different implementations may vary wildly on implementation details. In our case, Kaffe JVM [26], Java stack is implemented plainly with each Java frame slot corresponding to a unique native frame address, so that we can extract Java stack from native stack readily. For other JVMs like Hotspot [31] or Jikes/RVM [32], native stack layout could be very different from Java frame because of JIT inlining and many other optimizations. However, our techniques are still applicable. For these JVM implementations, stack sampling is doable even more easily by bytecode instrumentation or stack walking callbacks to the JVM Tool Interface (TI) [33].

VI. CONCLUSION AND FUTURE WORK

This paper has introduced new methods to estimate inter-thread and thread-object correlations in distributed object sharing systems. By means of sampling, we can profile accurate correlations at low cost. These methods are useful for devising better load balancing policies to optimize thread placement and hence reduce communication costs of object sharing. Our future work is to formulate an advanced load balancing policy that utilizes the correlation maps and sticky sets gathered to complement the insufficient policy based on system load monitoring alone. Our active correlation tracking mechanism still needs to be enhanced for taking home effect into account for proper thread migration decisions in some tricky cases that objects shared by a pair of threads are homed at neither node of the threads. In terms of profiling efficiency, it is desirable to have distributed algorithms for deducing correlation maps in a more scalable way.

ACKNOWLEDGMENT

This research was supported by Hong Kong RGC grant HKU7176/06E and China 863 grant 2006AA01A111.

REFERENCES

[1] GigaSpaces eXtreme Application Platform (XAP) – a commercial JavaSpaces implementation. http://www.gigaspaces.com.

[2] Oracle Coherence – an in-memory distributed data grid solution. http://www.oracle.com/technology/ products/coherence/index.html.

[3] A. Zilka. Open Terracotta – JVM clustering, scalability and reliability for Java. http://www.terracotta.org.

[4] The DARPA High Productivity Computing Systems (HPCS) Project. http://www.highproductivity.org.

[5] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*. p.519-538, San Diego, CA, USA, Oct 16-20, 2005.

[6] B. Chamberlain, D. Callahan, and H. Zima. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, Aug 2007.

[7] J.P.Lewis and U. Neumann. Performance of Java versus C++. Computer Graphics and Immersive Technology Lab University of Southern California, Jan. 2003 (updated 2004). http://www.idiom.com/~zilla/Computer/javaCbenchmark.html.

[8] J. M. Bull, L. A. Smith, L. Pottage, and R. Freeman. Benchmarking Java against C and Fortran for scientific applications. In *Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, p.97-105, Palo Alto, California, United States, Jun 2001.

[9] Sun Microsystems. Learn about Java Technology. (accessed Sep 29, 2008). http://java.com/en/about/.

[10] Y. Aridor, M. Factor, and A. Teperman. cJVM: a single system image of a JVM on a cluster. In *Proceedings of International Conference on Parallel Processing (ICPP'99)*, p.4-11, Sept 21-24, 1999.

[11] W. Zhu, C. L. Wang, and F. C. M. Lau. JESSICA2: A distributed Java virtual machine with transparent thread migration support. In *Proceedings of the 4th IEEE International Conference on Cluster Computing (CLUSTER'02)*, p.381-388, Chicago, USA, Sep. 2002.

[12] M. Factor, A. Schuster, and K. Shagin. JavaSplit: A runtime for execution of monolithic Java programs on heterogeneous collections of commodity workstations. In *Proceedings of the 5th IEEE International Conference on Cluster Computing (CLUSTER'03)*, p.110-117, Hong Kong, China, Dec 2003.

[13] A. Itzkovitz, A. Schuster, and L. Shalev. Thread migration and its applications in distributed shared memory systems. *Journal of Systems and Software*, 42(1):71–87, Jul 1997.

[14] Y. Sudo, S. Suzuki, and S. Shibayama. Distributed thread scheduling methods for reducing page thrashing. In *Proceedings of the 6th IEEE International Symposium on High Performance Distributed Computing (HPDC'97)*, p.356, Portland, OR, USA, Aug 5-8, 1997.

[15] K. Thitikamol, and P. J. Keleher. Active correlation tracking. In *Proceedings of 19th IEEE International Conference on Distributed Computing Systems (ICDCS'99)*, p.324-331, Austin, TX, 1999

[16] A. Barak and A. Litman. MOS: A multicomputer distributed operating system. *Software: Practice and Experience*, 15(8):725-737, Aug 1985.

[17] R. Ho, C.L. Wang, and F.C.M. Lau. Lightweight process migration and memory prefetching on openMosix. In *Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS'08)*, pp.1-12, Miami, FL, Apr 14-18, 2008.

[18] Y. Zhou, L. Iftode, and K. Li. Performance evaluation of two home-based lazy release consistency protocols for shared virtual memory systems. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI'96)*, p.75-88, Seattle, Washington, United States, Oct 29-Nov 01, 1996.

[19] Y. Luo, K. T. Lam, C. L. Wang. Path-analytic distributed object prefetching. In *Proceedings of the 10th International Symposium on Pervasive Systems, Algorithms and Networks (ISPAN'09)*, p.98-103, Kaohsiung, Taiwan, Dec 14-16, 2009.

[20] J. Pal Singh, C. Holt, T. Totsuka, A. Gupta, and J. L. Hennessy. Load balancing and data locality in adaptive hierarchical nbody methods: Barnes-hut, fast multipole, and rasiosity. *Journal of Parallel and Distributed Computing*, 27(2):118–141, Jun 1995.

[21] S. M. Blackburn, S. Singhai, M. Hertz, K. S. McKinley, and J. E. B. Moss. Pretenuring for Java. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'01)*, p.342-352, Tampa Bay, FL, USA, Oct 14-18, 2001.

[22] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th International Symposium on Computer Architecture (ISCA'92)*, p.13-21, Queensland, Australia, May 19-21, 1992.

[23] L. Iftode, J. P. Singh, and K. Li. Scope consistency: a bridge between release consistency and entry consistency. In *Proceedings of the 8th ACM Symposium on Parallel Algorithms and Architectures* (SPAA'96), p.277-287, Padua, Italy, Jun 24-26, 1996.

[24] The HKU Gideon 300 Cluster. http://www.srg.cs.hku. hk/gideon/.

[25] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture (ISCA'95)*, p.24-36, Jun 22-24, 1995.

[26] T. Wilkinson. Kaffe: a clean room implementation of the Java virtual machine. http://www.kaffe.org.

[27] K. Thitikamol and P. Keleher. Thread migration and communication minimization in DSM systems. In *Proc. IEEE, Special Issue on Distributed Shared Memory Systems*, 87(3):487–497, Mar 1999.

[28] T-Y Liang, C-K Shieh, and J-Q Li. Selecting threads for workload migration in software distributed shared memory systems. *Parallel Computing*, 28(6):893-913, 2002.

[29] M. Jump, S. M. Blackburn, and K. S. McKinley. Dynamic object sampling for pretenuring. In *Proceedings of the 4th International Symposium on Memory management (ISMM'04)*, Vancouver, BC, Canada, Oct 24-25, 2004

[30] J. Whaley. A portable sampling-based profiler for Java virtual machines. In *Proceedings of the ACM 2000 Conference on Java Grande*, p.78-87, San Francisco, California, USA, Jun 03-04, 2000.

[31] M. Paleczny, C. Vick, and C. Click. The Java HotSpot™ server compiler. In *Proceedings of the Java Virtual Machine Research and Technology Symposium* – Vol. 1, p.1-12, Monterey, California, USA, Apr 23-24, 2001.

[32] B. Alpern et al. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, Feb 2000.

[33] Sun Microsystems. The JVM tool interface (JVM TI), version 1.0. http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/jvmti.html.