Garbage Collector Assisted Memory Offloading for Memory-Constrained Devices

DeQing Chen, Alan Messer, Dejan Milojicic, and Sandhya Dwarkadas

*University of Rochester
†Samsung Electronics CTO/SISA, ‡HP Labs at Palo Alto

Technical Report #780

Computer Science Department, University of Rochester

August 2003

Abstract

Our everyday lives are becoming increasingly filled with mobile devices of varying capabilities. The common practice of creating multiple versions of the same application to cope with diverse device resource capabilities increases software development and maintenance costs. In this paper, we discuss an offloading method to mask out the memory constraints on devices running a typical Java virtual machine. The method allows the garbage collector to selectively offload part of the object heap into a nearby wired server. In comparison with traditional virtual memory techniques, the garbage collector can make wiser offloading choices using information about object access patterns at a finer granularity. Our experiments show that our prototype introduces modest overhead in the JVM while allowing applications to execute on devices without enough physical memory. In addition, when running with the Linux virtual memory system under intense memory constraints, the prototype achieves an average improvement of 24% in run-time performance and 53% in energy savings.

Keywords: pervasive computing, Java virtual machines, garbage collection, distributed systems, memory management.

1 Introduction

Many people foresee a pervasive computing environment. Computing services are delivered to the locale by the seamless cooperation of embedded and mobile devices. A common aspect of

*This work started when Chen was an intern at HP Labs. Chen and Dwarkadas were supported in part by NSF grants CCR-9988361, CCR-0219848, ECS-0225413, and EIA-0080124; by DARPA/ITO under AFRL contract F29601-00-K-0182; by the U.S. Department of Energy Office of Inertial Confinement Fusion under Cooperative Agreement No. DE-FC03-92SF19460; and by equipment or financial grants from Compaq, IBM, Intel, and Sun.
this environment is the large amount of diversity in the resource capacities on different devices, including processor speed, memory size, network bandwidth/latency, and power supply. This can be caused by various factors such as design preferences and marketing considerations. However, it complicates the effort to deliver unmodified software to any device.

Maintaining application generality helps reduce development and maintenance costs. Although heterogeneous programming systems such as Java [16] have been used to hide underlying platform incompatibilities, they do not consider the differences in resource capacities, apart from exposing physical characteristics such as a device’s screen size. Instead, special programming paradigms [15] and trimmed-down application suites [17] are designed to cope with the many levels of device capacities. These approaches require programmers to be very aware of the underlying hardware limitations. As a result, it is difficult to develop pervasive software that can be implemented to execute on a range of devices cost-effectively.

In this paper, we evaluate the ability to hide memory size constraints on Java virtual machine (JVM) based mobile devices. In a pervasive computing environment, memory capacity is a highly variable resource, exhibiting an order of magnitude difference from small embedded devices, to mobile and handheld-devices, and up to desktop PC systems. Unlike PC systems, the run-time memory on mobile devices is rarely user upgradeable, because of limited physical space and power supply constraints. On the other hand, with progress in technology and the increased deployment of wireless communication infrastructure [30], a mobile device will become more likely to find accessible memory resources in its immediate neighborhood, especially in environments such as offices, homes, and public libraries. We perceive that there will be a distinct opportunity for memory constrained mobile devices to exploit these resources.

Historically, virtual memory systems have been used to extend system address space with the help of fast backing stores. However, a traditional page-based virtual memory system is not very suitable for mobile devices. First, many processors used for these devices have no support for virtual memory management. Second, permanent stores, such as mechanical disks or flash memory, consume valuable power and space. Third, for Java applications, we have found that typical JVMs run poorly under virtual memory when physical memory is constrained.

To understand the JVM performance with virtual memory, we ran Java applications on a memory-constrained Linux system. We used two benchmark Java applications from the JVM98 benchmark suite, _209_db and _227_mtrt_. We booted the Linux system with different memory configurations to simulate physical memory constraints in the range from 16MB to 24MB. The normalized application execution times under different memory configurations are plotted in Figure 1. The graph shows that system performance deteriorates quickly after the available memory drops below 20MB and 18MB, respectively. The disk exhibited excessive activity because of the poor performance of the virtual memory system, which led to much of the performance degradation.

On investigation, we found that this occurred because Java uses a garbage collector to maintain its object heap. To prevent the object heap from becoming full, the JVM must periodically traverse the entire heap to free “dead” objects. The traversal ruins memory access locality and degrades the performance of the virtual memory system. In addition, most Java applications create large numbers of small objects. The virtual memory system’s page-based granularity prevents it from offloading rarely used objects stored in the same page as frequently accessed objects. Clearly, objects with similar lifetimes could be grouped by page but this only removes part of the problem.
Previous research [4] has shown that the performance of a virtual memory system can be improved by exploiting application-specified knowledge about memory access patterns. A JVM has the unique ability to automatically collect and find object access patterns by exploiting the indirect resulting from the use of a byte-code interpreter. We propose extending a JVM to mask memory constraints by offloading live objects from a mobile device to a nearby wired server. This environment, which we call a surrogate, cooperates with the original client device to manage and garbage collect what is now a distributed object heap. We call our approach Distributed Cooperative Garbage Collection (DCGC). Offloaded objects are chosen based on their temporal and spatial locality patterns. The proposed extension can be a stand-alone technology to relieve memory constraints for mobile devices without a virtual memory system. In addition, it can be used as a complementary technique to improve Java application performance on existing virtual-memory-based systems by improving memory locality. We believe our solution can be applied to any memory constrained device. Due to space limitations, we will focus our discussion on JVM-based mobile and hand-held devices.

The rest of the paper is organized as follows. In Section 2, we describe the design of our system. We describe an implementation using HP’s ChaiVM in Section 3, and evaluate its efficiency and sufficiency in Section 4. Lessons learned from this approach are discussed in Section 5. Closely related work is covered in Section 6. Finally, we conclude the paper and outline potential future work in Section 7.

## 2 Design

Figure 2 illustrates the key interactions between a client JVM device and its corresponding surrogate. The JVM on the client device starts a Java application. The JVM monitors the object access patterns while interpreting the bytecodes. When the garbage collector cannot free up enough memory when necessary, it will trigger the offloading system. At this point, the system scans the current
SurrogateClient migrating GC threads Exceeding available memory resources

SurrogateClient migrated object pulling back Accessing a migrated object

Figure 2: Cooperative garbage collector for offloading memory

object heap to offload enough objects by migrating them to the surrogate so that the occupied memory space can be vacated. The system selects offloaded objects according to their locality patterns in the hope that they are least likely to be accessed by the application in the future.

Because the migrated objects belong to the set of live objects, the device JVM cannot completely remove all traces of them. Instead, the device JVM keeps a special handle entry for each remote object. If the application attempts to access a remote object, the device JVM reallocates memory space to pull back the object so that it can be accessed locally. Subsequently, the surrogate will remove the remote object instance at its local storage. The memory allocation for a pulled-back object may trigger a new round of garbage collection or even object offloading.

The surrogate accommodates the migrated objects and cooperates with the JVM on the device side to perform distributed garbage collection over the set of current objects. The surrogate need not execute an entire JVM or be able to interpret Java bytecodes. However, it must be able to traverse migrated objects and identify potential object references inside them.

In comparison with traditional page-based virtual memory systems, the proposed cooperative garbage collector has the potential to be more efficient on a JVM-based memory-constrained device. The JVM is able to collect object access patterns at a finer granularity than page-based systems. It thus can make better offloading decisions and reduce communication traffic with the surrogate. Besides providing space for offloaded objects, the surrogate actively cooperates with the client device in managing these objects. Because the client JVM no longer needs to traverse the migrated objects, it will have a smaller memory footprint and improved locality.

However, the proposed system has a higher software cost in monitoring object accesses than page-based virtual memory systems. The overhead is mostly due to checking object location for each object access attempt. In addition, instead of the overhead for disk activity, a communication cost to migrate and pull objects to and from the surrogate is incurred. The migration and management of remote objects is crucial to the efficiency of the proposed system. In the rest of this section, we discuss our policy for selecting offloaded objects and our trace-based algorithm for distributed object garbage collection.

2.1 Offloading Policy

A desired offloading policy would be able to find objects that are least probable to be used in the future with minimal computation overhead. A policy that chooses offloaded objects randomly
could have very low decision overhead, although many of the offloaded objects might soon be used. Fortunately, with access information on individual objects, we can make wiser choices.

Previous research has shown that object usage patterns in Java applications exhibit strong temporal locality, i.e., objects that are used in the recent past are likely to be used in the near future [6, 8]. In the current design, we use an object’s past access history as the indicator of its temporal locality. A shift counter is used to represent each object’s recent access frequency. To obtain this value, an application’s life cycle is divided into consecutive epochs. An epoch can correspond to a garbage collection cycle. If the object is accessed in the current epoch, the JVM sets the most significant bit of its shift counter. At the end of each epoch, the counter is right-shifted. Objects that have not been accessed for a long time will have their counter reset to 0. Objects that were frequently accessed in the past will have higher counter values.

Besides temporal locality, we have also experimented with spatial locality to help offloading decisions. An object’s spatial locality is measured as its distance from the root object set, which is the set of objects from which a typical garbage collector starts to traverse the entire object heap. It usually contains objects that are pre-registered in the JVM, on the stack, in the current CPU registers, or static objects. Thus, for a deeper object to be accessed, some shallower objects must be accessed first. Intuitively, this means that a deeper object will have less opportunity to be accessed in the future than its shallower ancestor objects.

**Combined migration policy.** The information on temporal locality and spatial locality is combined to select objects for migration. As we have described above, temporal locality is obtained from a per-object history counter. The depth information can be obtained when the garbage collector traverses the object heap. When combined to make offloading decisions, temporal locality is given preference over spatial locality. That is, we always migrate objects with the lowest recent access count. When choosing among objects with equal access history, depth information is used to find deeper objects.

### 2.2 Distributed Garbage Collection

We use a trace-based scheme to manage objects distributed between the client device and the surrogate [23]. As shown in Figure 3, both sides keep a pair of tables recording the cross-site reference information. The incoming tables record objects that are referenced from the other side. The outgoing tables record references in the opposite direction. Objects in the outgoing table on one side appear in the incoming table of the other side. When the client device begins a garbage collection cycle, it adds objects in the incoming table to the root object set. On the surrogate side, the incoming table constitutes its root object set. Each side can conduct garbage collection locally and independently.

During this process, both sides have to exchange their outgoing table information in order to keep their incoming table accurate. Otherwise, dead objects might not be collected. However, information exchange at the end of each GC cycle can cause significant synchronization and communication overhead. For optimization, we take advantage of the fact that the two sides have unbalanced roles, i.e. objects are only mutatated on the client device. In addition, migrated objects are read-only at the surrogate. The outgoing table on the surrogate will not change unless its incoming table is updated by the client device. Thus the communication and synchronization between the client device and the surrogate are only necessary on two occasions: when the client device detects
If object $a$ deletes its references to both objects $d$ and $e$, object $d$ can be subsequently collected. However, objects $b$, $e$, and $f$ are not immediately collectible as they are in a global loop. They will be collected after we migrate $b$.

a change in its outgoing table; and after the surrogate receives an update from the client device and updates its own outgoing table.

### 2.2.1 Collecting Object Loops

When some dead objects are formed into global loops spanning both sides, they cannot be garbage collected even when the reference tables are accurate. An example is shown in Figure 3 where a loop is composed of the objects $b$, $e$, and $f$. After object $a$ deletes its references to objects $d$ and $e$, none of these objects are reachable from the root set if they are not distributed. When distributed, object $b$ is referenced from the client device’s incoming table and would be treated as a root object. Subsequently, the entire loop would be prevented from being collected.

Fortunately, this problem can be resolved with our migration policy. If an object has become collectible, its access history will decrement after each collection cycle and eventually will reach 0. Our migration policy always offloads objects that are only reachable from the incoming table. As a result, dead objects in the loop will migrate to the surrogate and be collected. For example, the object loop in Figure 3 will be collected after object $b$ migrates to the surrogate.

### 2.3 Discussion

Our design is based on several assumptions. First, the wireless communication between a client and a surrogate has reasonable bandwidth and reliability. This has not been true until today for most mobile devices. However, recent progress and the spread of newest wireless communication techniques could make it more realistic. For example, we have seen that 802.11-based designs have gained popularity in recent years [30] through increasing deployment by major mobile carriers in various public venues, making high-bandwidth wireless connection more ubiquitous than before.
Our scheme is more applicable when the mobile device is in a relatively stable environment, like offices, bookstores, or convention centers. In these environments, communication errors are more likely to be short term and can be relieved with orthogonal techniques such as replication.

Finally, we have assumed the surrogate always has enough memory and computation power to store and process migrated objects. If a client cannot find an appropriate surrogate in the immediate neighborhood, we expect the client can use fixed proxies reachable through the Internet. This is similar to a home-based proxy architecture that has been previously proposed to support mobile devices [25].

3 Prototype Implementation

We implemented a prototype DCGC system based on HP’s ChaiVM implementation. ChaiVM is designed for devices with constrained memory resources, such as HP’s iPAQ handheld devices. It features a slim interpreter with a small memory footprint, and uses an incremental garbage collector to avoid disrupting interactive applications. We are currently porting the prototype to other similar kinds of JVM systems (e.g., J2ME from Sun Microsystems).

The ChaiVM garbage collector (GC) uses a three-color mark-and-sweep-based algorithm. The GC thread runs concurrently with other application threads. The information on total available memory is obtained from a ChaiVM-specific environment variable at startup time (CHAIVM_MEMORY_SIZE). The interpreter monitors the allocation of new objects. When the available memory reaches a low-watermark threshold, the GC thread is activated to collect dead objects.

We modified ChaiVM to handle object offloading. In the original ChaiVM, objects are indirectly addressed through their handles. Each time an object is accessed, its handle has to be dereferenced to obtain the object’s memory address. We instrument this dereference action to allow migrated (that is, remote) objects to be checked, and to collect object access patterns.

We create a new object status table to store additional object information. Four bits are used to record an object’s access history and four bits are used to record an object’s depth. One bit is used to flag whether the object is being referenced by an object on the surrogate (i.e., an INCOMING reference), one bit to flag whether the object is referencing a remote object (i.e., an OUTGOING reference), and one bit to indicate whether the object is reachable from the local root set (see Section 2.2).

The prototype communicates with a stand-alone surrogate via a private TCP channel. The surrogate implements the garbage collector functionality described in Section 2.2.

4 Experiments

For our experiments, we used a commodity PC to emulate a typical memory constrained handheld device. This is justified because when memory becomes constrained the execution time is dominated by virtual memory overhead, which is fairly constant across machine types. In addition, we normalize all our execution times to make results comparable. All experiments are run on a PC with a 700MHz Intel Pentium III processor and 128MB of memory, running Redhat Linux 6.2.
Table 1: Benchmark Applications

<table>
<thead>
<tr>
<th>Application</th>
<th>Description</th>
<th>Peak Mem. Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>201_compress</td>
<td>Modified LZW method (10% config.)</td>
<td>9.7MB</td>
</tr>
<tr>
<td>227_mtrt</td>
<td>Multi-threaded raytracing (100% config.)</td>
<td>10.7MB</td>
</tr>
<tr>
<td>209_db</td>
<td>Database benchmark (100% config.)</td>
<td>11.9MB</td>
</tr>
<tr>
<td>ThesJava</td>
<td>A demo thesaurus dictionary; looking up 40 words from Shakespeare's <em>MacBeth</em></td>
<td>6.8 MB</td>
</tr>
<tr>
<td>PJ</td>
<td>A toolkit for processing PDF files; converting <em>Henry VI</em> from ASCII to PDF format</td>
<td>10.2MB</td>
</tr>
</tbody>
</table>

We experimented with different memory configurations, including different ChaiVM heap sizes and different physical memory sizes. We changed the ChaiVM heap sizes by setting the environment variable `CHAIVM_MEMORY_SIZE`. The physical memory size seen by the virtual memory system was set by changing the `mem=xxxM` kernel boot parameter.

We ran the surrogate on another Linux box with a 2GHz Pentium IV processor and 512MB of memory. The client and surrogate machines were connected with an 11.2Mbps IEEE 802.11b wireless Ethernet. The client used a DLink DWL650 PCMCIA card for the wireless network connection.

4.1 Memory-Intensive Applications

We tested a variety of applications. First, we chose three applications — 201_compress, 209_db, and 227_mtrt — from the JVM’98 benchmark suite. The JVM’98 benchmark applications can be configured to run at different “configurable speeds” by having the benchmarks use different amounts of input data or having them run different numbers of loops. In our experiments, the benchmark 201_compress is run at its 10% configuration speed, and 227_mtrt and 209_db are run at their 100% configuration speed. In addition, we used the ThesJava [34] (a Java thesaurus dictionary) and PJ [12] (a PDF file viewer) applications, which were obtained from the Internet. More information regarding these applications is provided in Table 1. In the table, we define the peak memory size as the smallest memory heap size configuration in which the application can successfully run to completion without migrating any objects.

4.2 Varying Degrees of Memory Constraints

We simulated different device types with different amounts of memory by varying the amount of memory used for the ChaiVM heap. We define the relative memory constraint as the degree to which a Java application has insufficient memory. This represents the minimum amount of memory that the DCGC has to offload (by migrating objects) to allow the application to run successfully.

\[
C_{\text{relative\ constraint}} = (\text{Mem}_{\text{peak}} - \text{Mem}_{\text{avail.}})/\text{Mem}_{\text{peak}}
\]

Overall performance. Figure 4 shows the application performance on ChaiVM employing the proposed offloading scheme (DCGC). The relative memory constraints range from 5% to 50%. Each application’s execution time is normalized with respect to its execution time on an unmodified
version of ChaiVM with sufficient available memory. Thus, a point above 1 on the y-axis indicates a performance degradation and a point below 1 indicates a performance speedup. Figure 5 shows the percentage of migrated objects that were pulled back to the original device in these experiments. The missing points for some applications are due to their not running to completion because of excessive thrashing.

![Normalized application performance under different relative memory constraints.](image1)

Figure 4: Normalized application performance under different relative memory constraints.

![Pull-back rates under different relative memory constraints.](image2)

Figure 5: Pull-back rates under different relative memory constraints.

The results vary for different applications. _227.mtrt_ has a stable 20% performance degradation when the memory constraint ranges from 5% to 35%. However, it shows a performance speedup after the memory constraint exceeds 35%. In Figure 5, we can see that for _227.mtrt_ the DCGC efficiently selects migrated objects and fewer than 5% of the migrated objects are pulled back. This results in low communication overhead. With fewer objects to traverse, the garbage
Table 2: Average size (bytes) of migrated objects with memory constraints at 20%

<table>
<thead>
<tr>
<th></th>
<th>ThesJava</th>
<th>_209_db</th>
<th>_201_compress</th>
<th>_227_mtrt</th>
<th>PJ</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>68</td>
<td>598</td>
<td>3771</td>
<td>68</td>
<td>1416</td>
</tr>
</tbody>
</table>

collector on the client has less work, resulting in the overall speedup.

_201_compress_ shows mixed results. While our DCGC system is highly efficient in choosing objects to migrate, it cannot run to completion in a reasonable amount of time for this application when the memory constraints exceed 20%. A closer look at the application shows that it uses two big arrays, each approximately 3.2MB in size, or approximately 1/3 of the peak memory requirement. Thus, when the memory constraint exceeds 25%, the DCGC has to migrate one of two large arrays. However, the array will soon be pulled back because of being reaccessed. The two arrays begin thrashing between the original device and the surrogate, making the application almost impossible to complete.

PJ slows down from 50% to 100% for memory constraints from 10% to 40%. PJ’s average migrated object size is the largest among the five applications tested. Although the percentage of pulled-back objects keeps growing in this range, the number of migrated objects is small with the average size of each object being larger as shown in Table 2, resulting in a smaller number of messages and hence lower communication cost. However, after memory constraints increase above 40%, the application begins to thrash, resulting once again in it being impossible to complete.

DCGC demonstrates a lower migration accuracy for both _209_db_ and _ThesJava_, with over 75% of migrated objects pulled back. Consequently, the overhead of DCGC increases rapidly as memory constraints increase. We believe that this is caused by the nature of the applications. They are both query/search applications and it is very likely that most objects are touched during a search. Thus, whatever policy is used to migrate objects, many of them will soon be pulled back.

**Migration policy.** Our prototype employs two pieces of object locality information to select a migrating object, object history representing temporal locality and object depth representing spatial locality. To further study their effectiveness in predicting an object’s future behavior, we compare the pull-back rates of three different policies in Figure 6. A random migration policy is used as a baseline policy. It selects objects randomly from the current live object set. The “Access” policy only uses an object’s past access history information. The “Access-Depth”, which is the policy implemented in our prototype, employs information on both object access history and object depth (Section 2.1).

From Figure 6, we can see that locality information helps improve migration accuracy. However, the degree of improvement depends on the characteristics of individual applications. For example, _209_db_ and _ThesJava_ both do searches and possess little locality in their access patterns. As a result, we can not do much better than a random policy for both of them. Noticeably, the random policy is very effective for _201_compress_. This is because _201_compress_ uses a very small part of the allocated objects for computation (basically, the two large arrays mentioned above). A randomly selected object has a very small chance of being accessed later. Therefore, the extra improvement from using locality information is negligible.

The figure also demonstrates the added benefit from using object depth information. It is effective in four of the five benchmarks. Recall that depth information is used to differentiate objects with equal recent access histories. _209_db_ uses a **Vector** (internally represented as an array) to
store database entries. Each entry holds references to its values. Since the benchmark periodically searches and sorts the entire array, all entry objects and their value objects (strings) have similar access histories. Using depth information only makes the system offload the value objects always ahead of the entry objects. This actually performs worse than offloading entry objects and their values together.

Ideally, we would like to compare against an offline optimal policy, one with future knowledge of object references. This requires logging every object access and then replaying it in the ChaiVM. As we have observed in the above experiments and previous research [6], Java applications usually create many small and short-lived objects. Furthermore, to make the optimal result deterministic, we must also record the time for which the garbage collector threads are activated. Logging all of these activities is difficult and impractical in our current simulator, but is an important part of our future work. However, we believe the current result is sufficient to demonstrate the efficiency and usefulness of exploiting object localities.

4.3 Using DCGC to Complement the Linux Virtual Memory System

In the previous experiments, we used 128MB of physical memory in order to remove the need for virtual memory. In Figure 1 (Section 1), we evaluated the effect of constraining physical memory on two of our benchmarks. Table 3 lists the memory size (and corresponding performance) at which each of our five applications begins to incur excessive paging activity due to insufficient physical memory. At these memory sizes, the applications incur intense paging activity; and if we further reduce the memory size, they would not run to completion in reasonable time. In the table, the running times are normalized with respect to those measured when the memory size is 24MB.

We use the memory sizes from Table 3 to boot Linux and repeat the experiments described in Section 4.1. Figure 7 shows the performance results. The y-axis shows the execution time normalized to when the applications run at the same memory size on the original ChaiVM. A value below 1 indicates a performance speedup relative to the original ChaiVM on the virtual memory system. The x-axis shows the relative memory constraints.
In these experiments, there are two main activities that influence an application’s performance. The first is the excess paging performed by the virtual memory system. It is caused by insufficient physical memory. The second activity is DCGC’s effort at migrating objects to the surrogate. The resulting smaller object heap size can improve the performance of the virtual memory system by reducing paging activities.

Table 3: Memory configurations at which excessive paging activity is incurred and the corresponding normalized performance (relative to performance with 24MB memory).

<table>
<thead>
<tr>
<th>Memory</th>
<th>11MB</th>
<th>18MB</th>
<th>12MB</th>
<th>16MB</th>
<th>12MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td>10.71</td>
<td>12.85</td>
<td>1.45</td>
<td>11.92</td>
<td>1.71</td>
</tr>
</tbody>
</table>

Figure 7: Normalized application performance at application thrashing points with DCGC limiting the size of the object heap.

In Figure 7, we can clearly see the combined influence of the two activities. When the memory constraint is 0% and thus no objects are migrated, the applications exhibit worse performance than on the original ChaiVM. This extra overhead is caused by DCGC’s own object monitoring and additional memory requirement for maintaining object access patterns. However, with higher relative memory constraints, DCGC will begin to migrate objects to the surrogate. As a result, the virtual memory system has a smaller object heap to deal with and avoids paging activity. Thus, despite the increased overhead due to the use of DCGC for object migration, the performance of the applications actually begins to improve since the performance of the virtual memory system improves. If DCGC increases memory constraints so that not all available physical memory is used, the applications eventually begin to thrash between the client and the surrogate since objects are repeatedly pulled back. Hence, application performance deteriorates again. Ideally, DCGC should set its memory constraint to match available physical memory and thus avoid virtual memory paging activity.

Table 4 lists those lowest points on the curves shown in Figure 7 for each application and its corresponding normalized performance. At these points, the applications can harvest the most perfor-
Performance benefits from the DCGC. We can see that three of the five applications we tested – `db`, `mtrt`, and `ThesJava` – can achieve a performance speedup on the DCGC implementation under these configurations. `compress` and `PJ` have better locality under the virtual memory system and do not gain much from DCGC. On average, our prototype implementation shows 24% improvement in performance relative to the performance of the original ChaiVM running with limited physical memory.

Energy consumption. To investigate the implication of our memory offloading scheme on the device energy consumption, we ran the same experiments described above on battery power. Before each execution we recharge the battery to full capacity. We use Linux’s `apm` command to find the battery status after the execution. The `apm` status gives an estimate of the percentage of the residual battery power. We used it to compute the amount of energy consumed. We list the results in Table 5 at the same configuration points as those in Table 4. The results are normalized to the energy consumption when the applications are running on the original ChaiVM with physical memory configuration provided in Table 3.

<table>
<thead>
<tr>
<th>Apps.</th>
<th>ThesJava</th>
<th>db</th>
<th>compress</th>
<th>mtrt</th>
<th>PJ</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCGC Config.</td>
<td>30%</td>
<td>20%</td>
<td>10%</td>
<td>40%</td>
<td>20%</td>
</tr>
<tr>
<td>Speedup</td>
<td>0.91</td>
<td>0.17</td>
<td>1.24</td>
<td>0.1</td>
<td>1.38</td>
</tr>
</tbody>
</table>

Table 4: Maximum speedups from DCGC

<table>
<thead>
<tr>
<th>Applications</th>
<th>ThesJava</th>
<th>db</th>
<th>compress</th>
<th>mtrt</th>
<th>PJ</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCGC Config.</td>
<td>30%</td>
<td>20%</td>
<td>10%</td>
<td>40%</td>
<td>20%</td>
</tr>
<tr>
<td>Energy Cons.</td>
<td>0.2</td>
<td>0.2</td>
<td>0.67</td>
<td>0.06</td>
<td>1.2</td>
</tr>
</tbody>
</table>

Table 5: Maximum energy savings from DCGC

 Interestingly, we find there is an even greater savings in energy consumed when using DCGC, although our replacement policy has been designed to optimize performance. For example, even though the application `compress` does not show improved performance, it can save one third of its energy consumption. We believe this is because when objects are offloaded, the virtual memory system can reduce paging activity and correspondingly reduce hard disk energy consumption. Consequently, in our experiments, cooperative garbage collection with a remote surrogate proves to be more energy efficient than paging to a local disk. On average, we save 53% of the energy consumed without the use of DCGC.

It must be emphasized that the above results are preliminary. However, they provide promising evidence that offloading can have both performance and energy benefits. Further refinement of our measuring methodology is required to verify these results. In addition, the energy savings we have obtained relate directly to the power efficiencies of the disk and the wireless communication components in our experimental platform.

\(^1\) `db` and `mtrt` cannot complete their execution with a fully-charged battery. We conservatively assume they just need 100% of the energy of the fully-charged battery.
4.4 Object Monitoring Overhead

We have also measured the overhead of monitoring object access patterns and checking object locations. The available memory size was set high enough to avoid any need for object migration. Table 6 shows the monitoring overhead when running the SPEC JVM'98 benchmarks. The extra overhead is mostly introduced by monitoring object access patterns and checking object locations. The overhead for most of the benchmarks is approximately 10%, with a few reaching 20%. Although the overhead is non-negligible, we believe it can be greatly optimized. As future work, the object monitoring can be adaptively disabled when physical memory is sufficient to support application execution. Furthermore, we can sample object accesses instead of monitoring every one of them [1] to reduce the overhead for collecting access patterns.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Normalized Monitoring Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>compress</td>
<td>7.21%</td>
</tr>
<tr>
<td>jess</td>
<td>11.1%</td>
</tr>
<tr>
<td>raytrace</td>
<td>9.80%</td>
</tr>
<tr>
<td>db</td>
<td>20.2%</td>
</tr>
<tr>
<td>javac</td>
<td>10.2%</td>
</tr>
<tr>
<td>mpeg</td>
<td>8.57%</td>
</tr>
<tr>
<td>mtrt</td>
<td>8.67%</td>
</tr>
<tr>
<td>jack</td>
<td>9.32%</td>
</tr>
</tbody>
</table>

Table 6: Normalized monitoring overhead

5 Lessons Learned

Locality information can help predict future object access. We use two pieces of locality information to select offloading objects i.e., temporal locality and spatial locality. The former is measured as an object’s recent access frequency and the latter is measured as an object’s distance from the root object set. Our experiments show that both of them help reduce the rate of pulled-back objects in comparison with a baseline random selection policy. In particular, we found that the spatial locality information can add additional benefit on top of temporal locality. It will be interesting future work to study the generality of this observation and its application in other JVM-related research domains.

Memory constraints can be relieved by extending the garbage collector. With an appropriate migration policy, an extended garbage collector can cooperate with the surrounding environment to support application execution under sizable memory constraints. Even with 40% of the required memory, our experiments show that four of five applications we tested can run to completion in reasonable time. However, we also found that the efficiency of the system varies among applications and is dependent on the available network bandwidth and latency. Applications whose future object access patterns exhibit strong locality can benefit most from our proposed scheme.

Memory offloading can be complementary to the virtual memory system. We have demonstrated that our proposed offloading JVM scheme can help the operating system achieve better system performance at the client device when physical memory is constrained. The performance improvement is achieved by two factors. First, offloading the traversal process required by the JVM garbage collector along with the offloaded objects reduces the execution overhead and memory constraints. Second, Java applications have much smaller object sizes than the physical page size used in the virtual memory system. Thus our offloading scheme can work at a finer granularity in choosing memory for offloading than a page-based system.
6 Related Work

Offloading functionality for resource-constrained devices has been used for many years. Client/server distributed middleware systems, like CORBA [24], provide the ability to write an application in a pre-partitioned fashion. The dynamic placement problem has been researched in load balancing and process migration [22]. Also, several different forms of offloading of applications between mobile clients and surrogates have been examined for resource-constrained devices [18, 19, 31, 32].

The AIDE system [21] is our earlier effort in the general subject of service offloading, including computation, memory, and power consumption. AIDE actively monitors the interactions between different objects. To relieve resource constraints, it uses the inter-object statistics to partition the application and migrate one part to the nearby surrogate. Unlike DCGC, an access to a remote object will be transformed automatically to be a remote procedure call.

The two methods reflect two design points over a spectrum of offloading strategies. Dynamic application partitioning offers bigger opportunities to optimize resource utilization in a pervasive computing environment. However, the system is much more complex than the GC-guided solution. For example, the system has to be more careful at object synchronizations to prevent deadlocks between distributed computing threads. The DCGC approach is limited in its capability to overcome resource constraints other than the memory resource. It is, however, simpler to implement and has the potential to make offloading decisions at a finer granularity. We believe the two methods are thus complementary to each other.

The most notable method for handling resource constraints is to reduce application functionality. Typical examples include the Windows CE/PocketPC platform [17] and various lightweight editions of Java [28]. However, this increases the burden on the user to understand the features and limitations of yet another version of the application.

Previous research has investigated extending memory by remote paging over networks [9, 11, 13, 14, 26, 20]. InterWeave [7] considered remote memory sharing in a heterogeneous environment. Our work is object-based and differs from this past work by considering JVM garbage collection.

Our object frequency counting scheme is reminiscent of that used by Hybrid Adaptive Caching (HAC) [5]. HAC exploits the object-level access information to reduce miss penalties in a page-caching distributed storage system. Its replacement policy uses this information to select victim page frames and compact them to keep “hot” objects in memory. Like HAC, a DCGC client also tries to locate “hot” memory regions at a finer granularity than pages to achieve lower miss rates. The HAC storage system simply provides storage space for application objects. On the other hand, a DCGC surrogate actively traverses the offloaded objects for the client. The client garbage collector can thus avoid accessing these objects and further reduce potential communication penalties.

Many projects have studied the efficient utilization of caching and memory subsystems in garbage collection systems [8, 10, 33, 35]. These projects are complementary in that they focus on improving memory system performance by exploiting access locality for improved object placement within the available memory on the device. Other research addressed improving cache performance by predicting application object behavior using profiling or application-level information [2, 4, 27]. Our migration policy also tries to choose objects based on the prediction of their future lifetime. However, we use prediction done at run-time based on dynamic object access patterns.

A reference-table-based distributed garbage collector was proposed by Mohamed-Ali [23]. Our distributed garbage collector implementation is based on this work, but our DCGC system varies
greatly from the context of their work. Bishop [3] first proposed to solve the problem of collecting global loops in distributed systems by object migration. Java RMI uses a lease-based distributed garbage collection scheme [29], but it does not allow heap objects to be automatically offloaded.

7 Conclusion

Diverse memory sizes on mobile devices hinder development of pervasive Java applications. In this paper, we present a technique called distributed, cooperative garbage collection (DCGC) that overcomes the problem of insufficient memory while maintaining good performance. This approach can be used without a virtual memory system or can be used to improve the performance when virtual memory is available.

We implemented a prototype on ChaiVM that enables a device to automatically use memory resources from its surrounding environment. It selectively offloads rarely used objects from a client device to a surrogate computer in its environment. The surrogate and the client then support a cooperative garbage collector between them. We experimented with five applications under varying amounts of insufficient memory and found that DCGC enabled them to overcome memory limitations with a modest performance penalty in the absence of memory constraints. Our offloading policy that exploits the locality patterns of object accesses can effectively predict future object usage to make better offloading choices. By comparing these applications under similar conditions with Linux’s virtual memory system, our system outperformed virtual memory by 24% on average when physical memory was constrained. Preliminary results also show that we can reduce energy consumption by 53% on average.

In future work, we would like to explore the possibility of leveraging operating system kernel information to help collect object access patterns. We are also interested in using prefetching to improve efficiency. Another interesting avenue for future work is to optimize the migration policy for energy consumption instead of runtime performance.

References