Clustering the Heap in Multi-Threaded Applications for Improved Garbage Collection

Myra Cohen
Department of Computer Science & Engineering
University of Nebraska-Lincoln
Lincoln, NE 68588-0115
myra@cse.unl.edu

Shiu Beng Kooi
Department of Computer Science & Engineering
University of Nebraska-Lincoln
Lincoln, NE 68588-0115
skooi@cse.unl.edu

Witawas Srisa-an
Department of Computer Science & Engineering
University of Nebraska-Lincoln
Lincoln, NE 68588-0115
witty@cse.unl.edu

ABSTRACT

Garbage collection can be a performance bottleneck in large distributed, multi-threaded applications. Applications may produce millions of objects during their lifetimes and may invoke hundreds or thousands of threads. When using a single shared heap, each time a garbage collection phase occurs all threads must be stopped, essentially halting all other processing. Attempts to fix this bottleneck include creating a single heap per thread, however this may not scale to large thread intensive applications. In this paper we explore the potential of clustering threads into related sub-heaps. We hypothesize that this will lead to a smaller shared heap, while maintaining good garbage collection parallelism. We leverage results from software module clustering to achieve this goal. Our results show that we can significantly reduce the number of sub-heaps created and reduce the number of objects in the shared heap in a representative application. This suggests that clustering may be a promising optimization technique for garbage collection in large multi-threaded systems with many shared objects.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*performance measures*; D.3.4 [**Programming Languages**]: Processors—*memory management (garbage collection)*

General Terms

Algorithms, Design, Experimentation, Performance

Keywords

Garbage collection, heap clustering, search based software engineering, hill climbing, virtual machines

1. INTRODUCTION

Garbage Collection (*GC*) is the process by which dynamically allocated memory is reclaimed as a program is executing. It reduces the likelihood of memory leaks (i.e. making programs more robust) and frees software engineers from the tedious task of explicit memory management. However, garbage collection can be very performance intensive, consuming as much as 38% of execution time [8]. To increase the efficiency of garbage collection various solutions have been employed. These include (i) allocating objects according to their maturity (program lifespan) so that short lived objects are frequently liberated [12, 26], (ii) estimating the allocated object life expectancy to avoid copying [3], (iii) utilizing multiple threads to determine the objects that can be deallocated [14] and (iv) providing hardware support [20, 21, 22].

Although these approaches have reduced the garbage collection overhead in a single user desktop environment, they do not necessarily scale to large distributed and server systems. First of all memory management on a server normally occurs across several threads and processors. Second the applications running on a server often offer services to thousands of users, require high levels of availability, and need longer execution cycles that cannot be stopped.

Our focus is on the first problem, that of handling garbage collection efficiently across multiple program threads. Figure 1 is a simple example of a program that has a single object which is shared among two threads. In this example an instance of "SharedObject" is created in thread one by the object *c1*. When *c2* is created it contains a reference pointer to the "SharedObject". However, *c2*, runs in thread two. Therefore a reference now exists to this object in both threads.

In a conventional garbage collection implementation, objects from all of the running threads are created in one large shared heap. As the garbage collector starts, all of the threads of the program are stopped except the one in which the garbage collector runs. Such synchronization prevents object inconsistency and invalid references that may occur during the collection process.

In the example both threads must be stopped to perform garbage collection on *c1* or *c2*. This makes sense since the threads containing these objects interact. However, all of the other threads that are active in this program must also be stopped, even if they are not currently involved with *c1* or *c2*. There may several hundred threads running in a distributed or server system at any one time, many of them with no relation to *c1* or *c2*. In addition, the popular adoption of the singleton pattern [9] to build server applications [5] results in an increase in the number of objects referenced

GECCO'06, July 8-12, 2006, Seattle, Washington, USA.

```
class SharedObject{
    // . . .
}
class ClientOne implements Runnable{
    SharedObject shared;
    public ClientOne(){
       shared=new SharedObject();
    }
    // . . .
}
class ClientTwo implements Runnable{
    SharedObject shared;
    public ClientTwo(SharedObject s){
        shared=s;
    }
    // . . .
}
class SharedThreads{
    public static void main(String [] args){
       ClientOne c1 = new ClientOne();
       ClientTwo c2 = new ClientTwo(c1.shared);
       Thread T1 = new Thread(c1);
       T1.start();
       Thread T2 = new Thread(c2);
       T2.start();
    }
}
```

Figure 1: Shared objects between threads

by multiple threads. Therefore, heap synchronization represents a severe performance bottleneck that reduces system productivity and throughput due to poor processor utilization during garbage collection. A related problem associated with the shared heap approach is that the heap has potential to get very large. A large shared heap size means that each garbage collection invocation may take longer to complete [23].

A few approaches have attempted to parallelize the garbage collection process and alleviate this bottleneck. Steensgaard [23] suggests creating a thread specific heap to store objects accessed by a single thread. The objects accessed by multiple threads are moved to a *shared heap*. The identification of multiple referenced objects is achieved through an *escape analysis* [4] of the program. Another approach proposed by Domani *et al.* [7] creates a single sub-heap for each thread, dynamically determining when a thread becomes "global" (i.e. when it accesses something outside of its thread). In both of these solutions, it is not clear if the suggested approach will scale to systems with hundreds or thousands of simultaneous threads since both approaches require creating a sub-heap for every thread. The excessive heap usage may make these approaches infeasible. One possible solution suggested by Steensgaard [23], is to identify clusters of threads that can be assigned to the same sub-heap. To date, this idea has not been explored. Two potential benefits are the reduction of the overall heap memory requirement and the reduction of the number of objects in the shared heap.

In this paper, we extend the work of Steensgaard and Domani *et al.* [7, 23] by experimenting with heap clustering to determine if it is feasible to increase the number of threads that can be assigned to a sub-heap and reduce the number of sub-heaps for a given program. We leverage research on software module clustering [10, 16, 19] to achieve this goal. A heuristic search algorithm is used to cluster threads of a program into separate sub-heaps based on prior run-time traces (histories) of the program's behavior. Specifically we perform a trace dependency analysis and then use a hill climb to find a good clustering with low coupling and high cohesion of threads.

We conduct two case studies aimed at evaluating the effectiveness of clustering in reducing the number of objects in the shared heap. In the first experiment, we apply a hill climb to cluster all shared objects. In the second experiment, we filter out objects that have a high degree of thread dependency prior to applying the hill climb. The results of these two experiments are compared against the thread specific heap approach proposed by Steensgaard [23].

Our hypothesis is that clustering can reduce the number of sub-heaps and the number of objects in the shared heap. The reduction in the number of sub-heaps should improve memory utilization and thus make large multi-threaded applications more scalable. The reduction in the number of objects in the shared heap should improve garbage collection response time as fewer collection invocations are needed in the shared heap. The main contribution of this work is to assess the feasibility of our hypothesis. To do this we have implemented a system that performs a static clustering based on program trace information. An actual system implementation and dynamic performance analysis is left as future work.

The rest of this paper is organized as follows. Section 2 describes the clustering problem and relates it to software module clustering. In section 3 a case study is described that explores the feasibility of the heap clustering approach. Section 4 presents the results of the case study. Section 5 discusses related work and section 6 concludes and presents future work.

2. HEAP CLUSTERING OF THREADS

The main approach to improving parallelization in garbage collection has been to create individual sub-heaps for each thread that store objects with no incoming references from other threads. The rest of the objects are moved into a single shared heap. This approach is often referred to as a thread-local [7] or thread-specific [23] heap solution. One short coming of this approach is that garbage collection in the shared heap still requires all application threads to stop. A potential solution suggested by Steensgaard is to provide a set of shared heaps that can be used by "a family of threads sharing some data structures" [23] to yield greater garbage collection efficiency. This has not been explored. It is unknown if threads can be partitioned or *clustered* in a way that enhances performance. This section describes the heap clustering problem in more detail and discusses how it can leverage prior work in software module clustering.

2.1 Example Clustering

Figure 2 is one example of a potential clustering for a set of six threads. There are eleven objects created by the threads. Thread one, three and four share object references. For instance *object 1* is referenced by T1 and T4 while *object 11* is referenced by T2 and T3. It is desirable to have sub-heaps that are not too large and that contain closely related objects. The objective is to have sub-heaps with maximal intra-cluster dependencies and minimal ("ideally" no) inter-cluster dependencies. In Figure 2 all threads that can be clustered are placed in sub-heaps, while a large shared heap

contains the other objects (ones that share a large number of references with all others making them ineligible for the clustering approach). The example shown is only one possible clustering for this set of threads and objects.

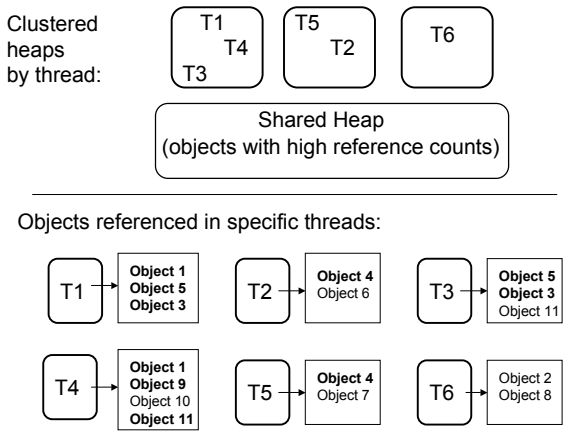

Figure 2: Ideal clustering for performance

Figures 3 and 4 show an example of how heap clustering and garbage collection work together. Each cluster has several objects where an object is represented by a circle. The values inside of the object are the threads that reference it. In Figure 3 if the garbage collector attempts to collect the first sub-heap it must stop all of the threads in *both* sub-heaps (i.e. threads one through four). This is due to the reference from the highlighted object in the first sub-heap to the second sub-heap. The garbage collector finds all potential conflicts and stops these threads. If this object was to be reclaimed and the threads in the second sub-heap were not stopped, a dangling reference [12] may be left behind. A typical implementation of a garbage collector will not allow this to happen but will stop all threads that are referenced by any object in the current heap.

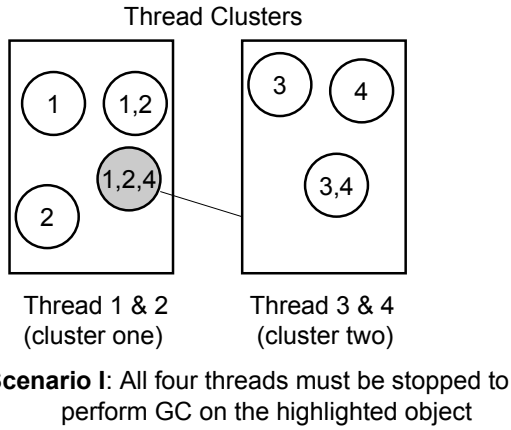

Figure 3: Imperfect clustering

In the second scenario (Figure 4) the object causing the dependencies between heaps has been removed and placed into a shared heap. When garbage collection occurs, on either of the two sub-heaps, only the threads in that sub-heap need to be stopped. For instance if garbage collection occurs on sub-heap one then only threads one and two need to be stopped. Threads three and four can continue to work on other tasks. Anytime, the shared heap is collected, however, all threads in the entire system must be stopped. This highlights our desire to keep the number of objects in the shared heap as small as possible.

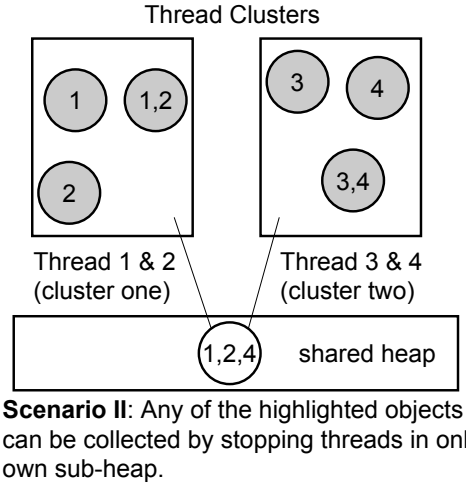

Figure 4: Using a shared heap

Finding a good clustering is not a simple task. We cannot use a brute force approach since the clustering problem is known to be NP Hard [18]. This makes heuristic search a natural candidate for finding a solution.

2.2 Relationship to Software Clustering

Another software engineering domain with a similar clustering problem is that of clustering software modules [10, 16, 17, 18, 19]. Software maintenance tasks often require reverse engineering of software modules. Mancoridis *et al.* [17], Mitchell [18, 19], Mahdavi [16] *et al.* and Harman *et al.* [10] have explored heuristic search techniques for clustering software modules. They attempt to find clusters that reduce the number of inter-module dependencies (i.e obtain low coupling) while at the same time increasing intra-module dependencies (i.e. high cohesion). The Bunch tool was developed for this purpose [17]. Its uses a hill climbing algorithm as the primary method to cluster modules.

2.2.1 Heuristic Search

Both heuristic and meta-heuristic search techniques have been used to solve the software module clustering problem [18]. The most successful method to date for the clustering problem for software modules has been a hill climb [10, 16, 18, 17]. Although we do not know that this is also the best approach for our problem, we use this as a starting point to determine if clustering will prove at all successful. The details of the hill climb algorithm used are described in the feasibility study.

2.2.2 Thread Dependency Graph

Mitchell, *et al.* [16, 19] use a *module dependency graph*, MDG, as a starting point for the hill climb. The graph defines the relationship between the software modules. Each

module is a node in the graph, with edges representing dependencies between modules. In our application, we measure the dependency at the thread level. This is the abstraction that seems to make the most sense for garbage collection since the threads are the beneficiaries of a good clustering. Even though individual objects may escape threads where they are created, they are always related back to a thread, not to another object. In addition even a small program may have thousands or millions of objects, which may make the clustering problem even harder.

We define a *thread dependency graph* (TDG) as a directed graph where each node is a thread and each edge represents a dependency with another thread. This can be created either statically or by using a post-analysis process from a system trace. For our analysis we have created this from historical system traces. To create the TDG, the trace is processed to identify all objects. For each of these objects a list of all threads that references it is created. We only consider objects that are referenced by multiple threads. The object-thread list is converted into a list of thread pairs, one for each dependency. In the example given in Figure 2, the TDG will consist of the following four edges : $T1-T3, T1-T4, T2-T5, T3-T4$. T6 is not found in this graph since it has no other thread dependencies.

2.2.3 Objective Function

In [18, 19] the authors presents a Module Quality metric (MQ). It is specifically designed to be maximal with high cohesion and low coupling. In [10] the authors compared the relative effectiveness of this metric against another clustering metric, EVM [25]. Both performed well, although EVM was found to be more robust under degradation (i.e. when there is a large amount of noise in the graph). We have chosen to use the MQ metric as an initial fitness function since it was developed specifically for an application that is close to ours in the desired outcome. The MQ metric is defined as follows [16, 18]. Each cluster has a Modularization Factor (MF) that is calculated by:

$$MF = \begin{cases} 0 & \text{if i} = 0 \\ \frac{i}{i+\frac{j}{2}j} & \text{if i}> 0 \end{cases}$$

where i is the number of internal edges in a cluster and j is the number of edges between this cluster and the other clusters. It is possible to use a weighted value for j but we leave this for future work. The Modularization Quality (MQ) is then defined as the sum of the MF's across all clusters: $MQ = \sum_{m=1}^{n} MF_m$.

It should be noted that this is a relative metric, rather an absolute metric since the possible maximum MQ value is dependent on the number of clusters.

3. CLUSTERING FEASIBILITY STUDY

A feasibility study was performed on a distributed application using the .NET framework. We have several benchmark programs that can be used in a distributed environment. We selected a peer-to-peer application described below for this study.

3.1 Platform and Subjects

The feasibility experiments use the Shared Source Common Language Infrastructure (SSCLI) or Rotor [24] from Microsoft. SSCLI is a research virtual machine that contains the core functionality of the Common Language Infrastructure which forms the basis of the .NET framework. The benchmark application used is a decentralized Peer-to-Peer (P2P) file sharing network program. A peer can serve as both a client and server at any time. It is modeled on programs from [6, 15]. The P2P network was configured to contain 80 nodes. Each node requests 240 files from other nodes and delivers 240 files to other nodes. A singleton object is used in each node as a gateway to take the incoming requests. The file size is set to be 75KB. We monitored one of the P2P nodes, which spawns 46 threads. These threads make and handle the requests for uploading and downloading files.

3.2 Clustering Process

Steensgaard [23] uses an escape analysis to determine which objects are referenced from more than one thread. In this paper we use a slightly different approach, although similar results may be obtainable via an escape analysis. We do a post analysis and use a program trace collected during an execution of the benchmark application. The trace file provides information such as object creation, thread identification, and reference assignments. The recorded information is then analyzed to derive the thread dependency graph.

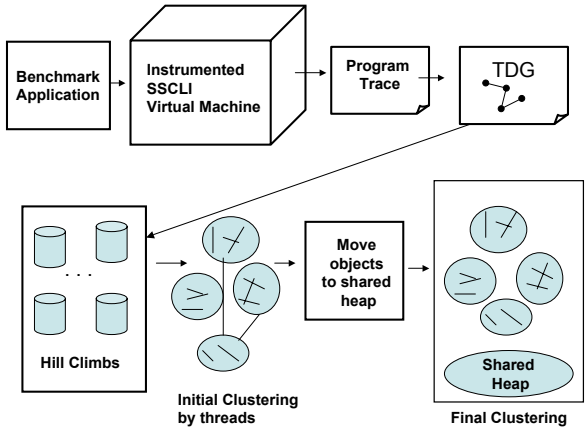

Figure 5: Clustering process

A diagram of process used is shown in Figure 5. Initially the program is run and a trace is created. As a first step, all objects that never escape their threads are removed from our analysis. The assumption is that these can be placed into the sub-heap containing their thread without affecting the quality of the clustering itself. It is possible that this will affect the garbage collection process however, since the final heap sizes matter. This is left for a later analysis with simulation. The next step in the process is to create a thread dependency graph from the trace. The TDG is then fed into parallel hill climbs. After each of the hill climbs has completed the best clustering is kept. It is probable that the clustering will still have some inter-heap dependencies. Although this may be feasible for software module clustering it may cause undue performance hits in our domain so these are removed in a post-processing step. Any objects that have incoming references from threads assigned to other clusters are moved down to the shared heap. This represents the final clustering.

3.2.1 Pre-processing Step (Filtering)

In the Bunch tool [17], a pre-processing step is used to remove omnipresent modules. These are modules that have many external references and are likely to make the clustering difficult. We observed that there were a small number of objects with a large reference count. As part of this study we wanted to determine if a pre-processing stage that removes these would provide a better clustering. In experiments that use pre-processing we sorted the object reference patterns by length in ascending order and observed that the number of objects for a particular thread reference pattern was only one or two for the top part of the list. This abruptly changed at a point in the file to patterns with several hundred objects. We selected this change as the cut-off point for choosing omnipresent objects. When filtering is used all omnipresent objects are placed into the shared heap and therefore not part of the clustering.

3.2.2 Hill Climb Implementation

The hill climb used in these experiments encodes the clusters as a string of values. This is similar to the Bunch representation [17]. The length of the string is the number of threads in the TDG. An initial random clustering is created and then a series of transformations is applied. The neighborhood of a clustering consists of all possible clusterings that can be obtained by moving one thread from one cluster to another. Given the string representation this is one mutation of a single location in the string. The number of clusters will change dynamically as the program runs. For instance, if a thread resides in a cluster of its own and is moved to a different cluster, the number of clusters will be reduced by one after this move. We do not use a steepest ascent method, but rather select a single new neighbor at random and evaluate the resulting MQ. If the MQ is greater then we accept the new clustering. If the MQ is less we accept it with a probability of 1/1000. A small probability for making a bad move was left in to help prevent converging too quickly in a local optimum. The stopping criterion used is 12,000 iterations or 800 consecutive bad moves.

3.3 Experiments

We ran forty five trials of the clustering process first with and then without the pre-processing filter that removes omnipresent objects. These were run on Linux 2.6, on a 45 node cluster where each node is a dual Opteron 250 SMP running at 2.4GHz with 16GB of RAM spread equally between the two processors.

We recorded the initial MQ's , the final MQ and the number of clusters and number of objects in the shared heap for each of the hill climbs. Finally, the post-processing step moved all remaining objects with external inter-heap references into the shared heap and the number of remaining objects in each cluster and in the shared heap were recorded.

4. RESULTS

The experimental goal is to evaluate the effectiveness of clustering by observing two performance criteria: the number of sub-heaps and the number of objects in the shared heap. The following subsections report our findings.

4.1 Reduction of Sub-Heaps

By clustering, we expect to reduce the number of sub-heaps while maintaining good garbage collection parallelism. Because the hill climb does not yield perfect clustering, we filter the input data in one experiment to eliminate obvious candidate objects that should be placed in the shared heap. The other experiment is conducted with all possible shared objects. Table 1 reports the result of the two experiments. In this table we show data taken from clusters with the best five and worst five MQ values for each method. If we had used a thread-local approach for garbage collection we would have had 46 sub-heaps for this program. Instead, with clustering we can reduce the number to 4, or by as much as 91.3%.

	Number of Clusters (% of reduction)		
	Best Five MQ Values		
Run	Thread-Local	With Filter	Without Filter
1	46	4 (91.30%)	4 (91.30%)
2	46	4 (91.30%)	4 (91.30%)
3	46	4 (91.30%)	4 (91.30%)
4	46	4 (91.30%)	4 (91.30%)
5	46	4 (91.30%)	4 (91.30%)
	Worst Five MQ Values		
Run	Thread-Local	With Filter	Without Filter
1	46	6 (86.96%)	6 (86.96%)
2	46	6 (86.96%)	6 (86.96%)
3	46	6 (86.96%)	6 (86.96%)
4	46	6 (86.96%)	7 (84.78%)
5	46	6 (86.96%)	6 (86.96%)

Table 1: Comparing the number of sub-heaps between thread-specific and clustering approaches

Larson and Krishan have shown that as the number of heap partitions increases, the efficient use of memory decreases since memory for each sub-heap is often over-allocated [13], therefore this clustering should significantly reduce the amount of heap memory required and improve memory utilization.

We also investigate the number of shared objects (excluding singly referenced objects) in each sub-heap. We find that objects are not evenly distributed in each sub-heap. Table 2 is an example of one distribution. It represents the clustering that moved the largest number of objects (134,406) out of the shared heap. In a real implementation, the heap must be partitioned based on the memory requirement. The effect of an uneven clustering on system performance is left to future work.

4.2 Reduction of Shared Objects

To study the effectiveness of clustering on the number of shared objects, we again conduct two experiments: using all shared objects and filtering out omnipresent objects. Table 3 shows the best, worst and average MQ values for 45 runs of our process using each of the two methods. We present the MQ's both before and after the hill climb. The data points labeled RMQNF and RMQF represent the randomly created clusters before the hill climb begins. RMQNF is the version that does not pre-filter while RMQF represents data that has been filtered. HCQNF and HCQF represents the MQ after the hill climb for the non-filtered and filtered versions respectively. Figure 6 shows a box plot of the MQ values. The two plots on the left are the random MQs before the hill climbs. The plots on the right illustrate the values after the hill climbs. Although we see no significant differences

Clusters	Number of Threads	Threads	Total Objects
1	7	7,9,10,12,13,18,29	34,733
2	11	1,3,4,5,6,8,11,14,15,16,21	39,817
3	13	2,17,20,24,26,27,28,30,32,33,34,35,44	33,251
4	15	19,22,23,25,31,36,37,38,39,40,41,42,43,45,46	26,605
Total Objects Assigned to Clusters			134,406

Table 2: Distribution of objects into clusters (from the run that minimizes number of objects in the shared heap)

in the values between the filtered and non-filtered versions after the hill climbs, the best individual MQ's were obtained with the filtering process.

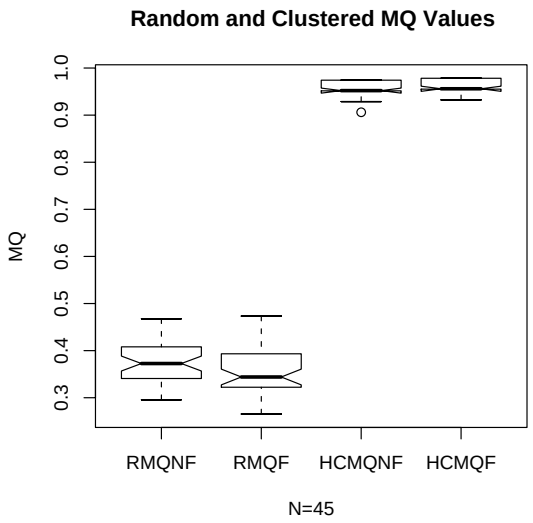

Figure 6: Comparison of MQ values before and after clustering

	Best MQ	Worst MQ	Avg. MQ	Std. Dev.
Random at Start				
Thread-Clustering no filtering (RMQNF)	0.468	0.296	0.371	0.040
Thread-Clustering with filtering (RMQF)	0.473	0.265	0.357	0.049
After Hill Climb				
Thread-Clustering no filtering (HCMQNF)	0.974	0.906	0.960	0.017
Thread-Clustering with filtering (HCMQF)	0.978	0.932	0.960	0.018

Table 3: MQ pre and post hill climb

In our study there were over 6 million objects created in the subject program. An analysis such as one proposed by Steensgaard [23] would have placed about 8 % of the total objects into the shared heap (452,679). Table 4 shows

	Objects in Shared Heap	
	Number Objects	**Percent Reduction**
Thread-Specific Approach	452,679	0.00 %
Thread-Clustering with no filtering		
	328,289	27.48 %
	318,273	29.69 %
	331,585	26.75 %
	335,005	26.00 %
	338,479	25.23 %
Thread-Clustering with filtering		
	337,221	25.51 %
	345,657	23.64 %
	347,361	23.27 %
	339,833	24.93 %
	332,503	26.55 %

Table 4: Comparison of objects in the shared heap

the numbers of objects remaining in the shared heap for each of the best five MQ values for both the filtered and unfiltered hill climbs after clustering. The data in the table is ordered in descending order of MQ value (i.e the best MQ for each method is shown first). In the best case, as many as 134,406 objects would have been moved into clustered sub-heaps using our approach, leaving 318,273 objects in the shared heap. This is a 29.69 % reduction over the Steensgaard method. Figures 7 and 8 provide a graphical representation of this data. In general it seems that the data which was not filtered does slightly better in reducing objects. One interesting observation is that the best MQ did not always directly translate to the largest reduction in objects. This may suggest more work is needed on the fitness function.

5. RELATED WORK

In work by Steensgaard an escape analysis is used to statically determine which objects can "escape" their given threads. All objects that do not live within a single thread are moved to a shared heap. All other threads are given their own heap [23]. Domani *et al.* [7] use a dynamic approach to partition their heap into one sub-heap per thread. At run-time, monitoring of the write barrier (trap of reference stores) is done to detect "global" objects. When garbage collection occurs, if a thread is not local then all threads are stopped. Another solution is to allocate objects on the stack whose lifetime is limited to a threads run-time call stack [4]. KaffeeOS, a Java virtual machine based operating system, divides the heap into process-specific heaps and a shared

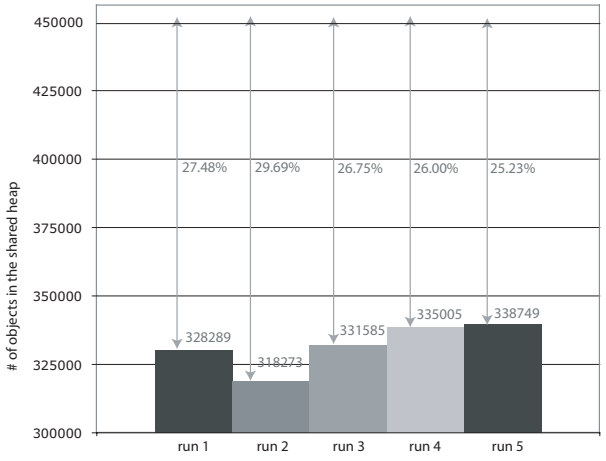

Figure 7: Reduction of shared objects (without filtering)

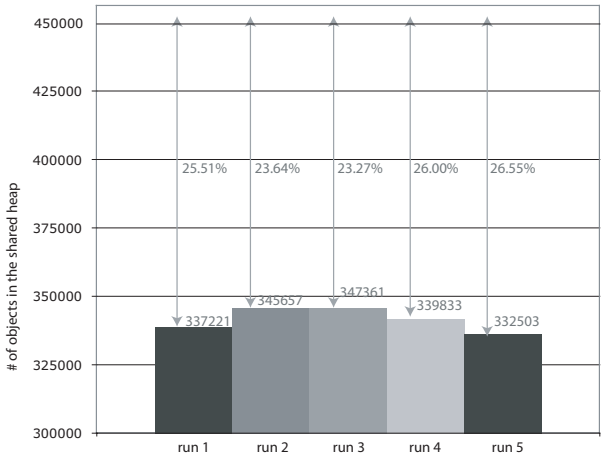

Figure 8: Reduction of shared objects (with filtering)

heap for data [2]. It does not allow objects in the shared heaps to reference objects in the process specific heaps.

Mitchell *et al.* [18, 19] approached the problem of software module clustering using genetic algorithms and hill climbing. They developed the MQ metric which was used in this paper as a fitness function. Mahdavi *et al.* extended this work by using a multiple hill climb that fuses common clusters between successes hill climbs forming basic building blocks [16]. Harman *et al.* [10] compared 2 different fitness functions for robustness in this application.

In this work, we differ from the other garbage collection sub-heap schemes in that we cluster our heaps. In the other garbage collection approaches each thread is given its own sub-heap to store objects that are referenced by that thread. Any objects that are referenced by multiple threads are moved into a shared location. In our approach, we find threads that have shared references and move these into common areas. We also differ in that the approach is a post-analysis of real trace data from a run of the program which should be a representative profile of the program. Our work is similar to the work on software module clustering, however, the application and domain differ. In software module clustering it is desirable to remove as much coupling as is possible, but in garbage collection even a small amount of coupling in resulting clusters may have a negative impact on performance.

6. CONCLUSIONS AND FUTURE WORK

In this paper we have explored the feasibility of parallelizing garbage collection by clustering threads into sub-heaps using a hill climbing algorithm. We examined a peer-to-peer distributed application and explored two versions of clustering. In the first experiment we clustered using all objects. In the second we filtered omnipresent objects and then performed the clustering. After each clustering all remaining objects with inter-heap references were moved to the shared heap.

We have found that it is possible to find a clustering that will reduce the number of sub-heaps by as much as 91.3% and the number of objects in the shared heap by as much as 29.69% in our case study application and expect that this will provide improvements in garbage collection performance. We found that the highest MQ did not always provide the best reduction in the shared heap indicating that we may need to examine the fitness function further.

In future work we plan to refine the clustering algorithm and to evaluate system performance. On the algorithm side, we plan to apply alternative fitness functions including one that incorporates weighted clustering. We also plan to try using the technique of multiple hill climbing with building blocks as presented in [16]. In addition, we will experiment with other benchmark programs to determine the scalability and generality of this approach.

Although we have reduced the number of sub-heaps as well as the number of objects in the share heap, it is not clear if this reduction will translate directly to a performance gain. We plan to examine two performance metrics on this front: *garbage collection parallelism* and *garbage collection execution time*. Garbage collection parallelism will measure the number of threads that must be stopped during each garbage collection invocation. We are building a simulator based on the methodology introduced by Hertz *et al.* [11] to perform this task. It will allocate objects to the assigned sub-heaps defined by the clustering algorithm or one of the other approaches such as the thread-local approach. It will obtain exact object lifespans from the program traces and then simulate all of the object allocations of threads and track all of the garbage collection invocations. We will also investigate changing parameters such as the size of each sub-heap and finer grain optimization that considers the amount of time that an object is shared by multiple threads.

In order to study garbage collection execution time we will implement thread clustering into a virtual machine. A major challenge is to implement the thread assignments dynamically. A recent study by [1] found that "cross-run information" (run-time information from previous executions) can be used for performance tuning. Their work creates an architecture to store and exploit cross-run information; this information is then applied to help with the selective optimization process in a Java virtual machine. In a similar fashion, cross-run information can be used to assist in assigning threads and forming and sizing clusters. We will use this approach, performing the analysis off-line and applying it to subsequent runs.

7. ACKNOWLEDGMENTS

This work was sponsored in part by an EPSCoR FIRST Award, NSF Award CNS-0411043 and by the Army Research Office through DURIP award W911NF-04-1-0104. We thank Mulyadi Oey for work on the benchmark programs.

8. REFERENCES

[1] M. Arnold, A. Welc, and V. T. Rajan. Improving virtual machine performance using a cross-run profile repository. In *Proceedings of the International Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 297–311, 2005.

[2] G. Back and W. C. Hsieh. The KaffeOS Java runtime system. *ACM Transactions on Programming Languages and Systems*, 27(4):583–630, 2005.

[3] S. M. Blackburn, S. Singhai, M. Hertz, K. S. McKinely, and J. E. B. Moss. Pretenuring for Java. *SIGPLAN Notices*, 36(11):342–352, 2001.

[4] B. Blanchet. Escape analysis for JavaTM: Theory and practice. *ACM Transactions on Programming Languages and Systems*, 25(6):713–775, 2003.

[5] D. Browning. Integrate .NET remoting into the enterprise. In *.NET Magazine*, November 2002.

[6] D. Conger. *Remoting with C# and .NET: remote objects for distributed applications*. Wiley Publishing, Inc., 2003.

[7] T. Domani, G. Goldshtein, E. K. Kolodner, E. Lewis, E. Petrank, and D. Sheinwald. Thread-local heaps for Java. In *Proceedings of the 3rd International Symposium on Memory management*, pages 76–87, 2002.

[8] L. Dykstra, W. Srisa-an, and J. M. Chang. An analysis of the garbage collection performance in Sun's HotSpot JVM. In *Proceedings of the IEEE International Performance Computing and Communications Conference*, pages 335–339, 2002.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[10] M. Harman, S. Swift, and K. Mahdavi. An empirical study of the robustness of two module clustering fitness functions. In *Proceedings of the Conference on Genetic and Evolutionary Computation*, pages 1029–1036, 2005.

[11] M. Hertz, S. M. Blackburn, J. E. B. Moss, K. S. McKinley, and D. Stefanović. Error-free garbage collection traces: how to cheat and not get caught. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, pages 140–151, 2002.

[12] R. Jones and R. Lins. *Garbage Collection: Algorithms for automatic Dynamic Memory Management*. John Wiley and Sons, 1998.

[13] P. Larson and M. Krishan. Memory allocation for long running server applications. In *Proceedings of the International Symposium on Memory Management*, 1998.

[14] C. D. Lo, W. Srisa-an, and J. M. Chang. A multithreaded concurrent garbage collector parallelizing the new instruction in Java. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, 2002. (CD-ROM).

[15] M. MacDonald. *Peer-to-Peer with VB.NET*. Apress, 2003.

[16] K. Mahdavi, M. Harman, and R. Hierons. A multiple hill climbing approach to software module clustering. In *Proceedings of the International Conference on Software Maintenance*, pages 315–324, 2003.

[17] S. Mancoridis, B.S.Mitchell, Y.Chen, and E.R.Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *Proceedings of the International Conference on Software Maintenance*, pages 50–59, 1999.

[18] B. Mitchell. *A Heuristic Search Approach to Solving the Software Clustering Problem*. PhD dissertation, Drexel University, Department of Computer Science, 2002.

[19] B. S. Mitchell, S. Mancoridis, and M. Traverso. Search based reverse engineering. In *Proceedings of the International Conference on Software Engineering and Knowledge Engineering*, pages 431–438, 2002.

[20] K. Nilsen and W. Schmidt. A high-performance hardware-assisted real-time garbage collection system. *Journal of Programming Languages*, pages 1–40, 1994.

[21] D. J. Roth and D. S. Wise. One-bit counts between unique and sticky. In *Proceedings of the International Symposium on Memory Management*, pages 49–56, 1998.

[22] W. Srisa-an, C. D. Lo, and J. M. Chang. Active memory processor: A hardware garbage collector for real-time Java embedded devices. *IEEE Transactions on Mobile Computing*, 2(2):89–101, 2003.

[23] B. Steensgaard. Thread-specific heaps for multi-threaded programs. In *Proceedings of the International Symposium on Memory management*, pages 18–24, 2000.

[24] D. Stutz, T. Neward, and G. Shilling. *Shared Source CLI Essentials*. O'Reilly and Associates, 2003.

[25] A. Tucker, S. Swift, and X. Liu. Variable grouping in multivariate time series via correlation. *IEEE Transactions on Systems, Man and Cybernetics, Part B*, 31(2):235–245, 2001.

[26] D. Ungar. Generational scavenging: A non-disruptive high performance storage reclamation algorithm. In *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 157–167, April 1984.