University of Waterloo
Faculty of Engineering

Visualizing an Object Memory Simulator

Sun Microsystems Labs
2600 Casey Avenue
Mountain View, CA, 94043
United States of America

Michael Jarrett
Student #99318764
msjarret@engmail.uwaterloo.ca
Computer Engineering - Term 3A

August 16^{th}, 2002

Michael Jarrett
2646 Standish Drive
North Vancouver, B.C.
Canada, V7H 1N1

August 16th, 2002

Dr. Anthony Vannelli, Chair
Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario
Canada, N2L 3G1

Re: 3A workterm report for Michael Jarrett (#99 318 764)

Dear Dr. Vannelli,

Attached is the report "*Visualizing an Object Memory Simulator*", submitted as my third work report. My last academic term was 3A, after which I began a co-op work term at *Sun Microsystems Labs*, working for the *Mayhem Project*.

The Mayhem Project is a group within *Sun Microsystems Labs* working to develop a computer system better designed to handle the objects used in object-oriented languages, concentrating on improving cache and garbage collection behavior. This report outlines the design and implementation of a tool to monitor the behavior of our simulation of the system.

This report was prepared without assistance. It was prepared for Mario Wolczko, Senior Staff Engineer at Sun Microsystems, and manager of the Mayhem Project.

I hereby confirm that I have received no further help other than what is mentioned above in writing this report. I also confirm this report has not been previously submitted for academic credit at this or any other academic institution.

Sincerely,

Michael Jarrett
99318764

Contributions

The Mayhem Project is a team at Sun Microsystems Labs led by Mario Wolczko. It is staffed by myself, Greg Wright, and Matthew Seidl. The team's goal is to design, test, and simulate an implementation of an object memory system suitable for use on a future generation of computer architecture. developed by Sun Microsystems. Such a system should have the ability to do in-cache garbage collection. To this end, we worked on a simulator, called 'tsim', capable of emulating the behavior of Java programs when run on an object memory system based on our design. Simulations for conventional processor and virtual machine implementations were also written so that common performance metrics could be compared between the simulation of an object memory system, and a conventional system.

My task for the Mayhem Project was to address visualization of the simulator. The simulator, which was partially completed when I arrived, had only text-based debugging output, and communicated very little as to whether the simulator was even working. The system's response to Java trace files was very difficult to see, and interaction patterns almost impossible to determine. It was decided that I would interface a tool called 'GCSpy' to the simulator so that it could be visualized in a graphical manner. After having being provided a version of GCSpy to evaluate, my task was to find a way to make the graphical client capable of visualizing a running tsim simulator.

Due to the fact that both the simulator and GCSpy were works in progress, my responsibilites also involved improving GCSpy for stability and for features not available in the initial version. My work led to several improvements in the GCSpy protocol, including rewind support, event information strings, and better interactive control.

Issues regarding the operation of the simulator had to be addressed during development, as GCSpy quickly revealed flaws that had yet to be discovered in the simulator up to that point. The use of GCSpy with the tsim simulator also required documentation, since neither tsim nor GCSpy had any significant documentation of their own.

This report outlines the reasoning and results of implementing a working GCSpy system. It provides a record of functionality, and the effects of running a GCSpy server inside the tsim simulation. It also makes recommendations as to the use of GCSpy, and furthermore,

the directions in which development should continue to improve it as a tool for statistical monitoring. This report will serve a final statement of the results of the integration of tsim and GCSpy for future review by the Mayhem Project.

Sun Microsystems is well-known for its efforts to support object oriented languages. This is shown by their work on the Java programming language. Sun performed the majority of work in designing the language and underlying virtual machine, implementing a free reference implementation, and promoting its use. Many of Sun's latest offerings rely on Java as a base from which developers and users alike can standardize. Therefore, Sun gains a huge advantage by improving their computer systems to take advantage of object-oriented environments.

One of the criticisms of Java is that garbage collection can impose a penalty on performance, and by using hardware to address this, a major barrier into a vast set of markets is removed. By improving performance and robustness of object-oriented languages with hardware, both the object oriented languages and the hardware that supports them serve to benefit greatly.

Summary

The purpose and scope of this report is to investigate and describe the implementation of a visualization of the Mayhem Project's simulator for object memory. The Mayhem Project team is currently writing a simulator to determine the effectiveness of an object memory system; a memory hierarchy that is aware of the existance of objects, and that is able to perform garbage collection in cache. This simulator processes 'trace files' generated by a customized Java Virtual Machine. These trace files are virtual-machine-independent records of the actions of a program.

A problem with the simulator is that it had no effective means of communicating with the user. The only output generated was debug textual output, which is produced at a rate of hundreds of thousands of lines a minute. This information was not easily handled by human operators, and as such, a tool to reduce and organise the information was required. GCSpy was chosen as the tool to use to visualize the system for many reasons. It has a Java interface, which allows it to be used by the simulator. Also, there was a graphical client, which can present information in a usable form.

Integrating GCSpy into the simulation successfully reduced the amount of information presented, and that the information was much more suitable for consumption by a human user. The performance overhead of the GCSpy server in the simulation was shown to be approximately 10% to 15%. By batching simulator runs, then replaying GCSpy results, a simulation could be viewed at any speed desired.

It was concluded that, based on its ability to reduce and organize informatiom, GCSpy was an excellent tool for viewing the behavior of a simulation, especially when debugging the system or simulator. However, GCSpy's primary shortcoming was the lack of tools for statistical analysis or storage of raw data in a processable form.

Based on these conclusions, it was recommended that GCSpy be used as the primary form of communication in the development of the simulator. It was recommended that new GCSpy clients be developed for handling of statistics and graphing of trends not directly visible in the graphical client. The GCSpy system itself would also benefit from more development effort.

Conclusions

From the analysis in the report body, it was concluded that the GCSpy server written for the tsim simulation can greatly summarize the amount of information presented to the viewer. This information was categorized and organized in an understandable manner. Performance costs related to GCSpy were only a small factor in the total simulation speed, and quite reasonable for the computer resources available within Sun Microsystems Labs.

It was concluded that the information provided by the graphical GCSpy client was extremely useful for debugging the simulator, and provided basic insight into the behavior of a particular Java trace. The ability to replay traces made it suitable for batch simulation runs, while not hampering the ability to use it in a real, running system.

It was concluded that the provided GCSpy client was inadequate for precise system measurements: the clients were in no way designed for post-processing of received information. While the graphical client provides an effective overview of a system at a particular point in time, history information over a period of time is neglected, as is the ability to save or process information to make calculations.

Recommendations

Based on the analysis and conclusions in this report, it is recommended that GCSpy be adopted, and effort be put into improving the GCSpy system:

- The graphical GCSpy client should be used with the GCSpy server for the tsim simulation of object memory systems during development as well as when simulations are being run.
- A library of saved GCSpy recordings should be used as a record of results for simulations instead of running GCSpy clients during simulations.

There is a great deal of room for improvement in GCSpy. It is recommended that development of GCSpy be continued in the following directions:

- A great deal more clients need to be created. Some examples of clients include:
  - A client that outputs graphs based on summary data over a period of time. This client could either render these graphs directly, or output data suitable for graphing in a program such as gnuplot or Matlab.
  - A client capable of comparing the output of different servers, or the differences over several runs of the same server, and producing statistical results.
  - A client capable of processing, filtering, and outputting statistical numbers about GCSpy data in a format readable by spreadsheet software.
- The protocol needs to be optimized, at least by adding the ability to incrementally update a space rather than a full update for each event. The libraries would likely need to be improved in the process.

Each change would only cost in time for the developer, most likely between 4-8 developer days per item.

Table of Contents

List of Figures

List of Tables

1 Introduction to the Mayhem Project

1.1 The Problem with Object Oriented Languages

Over the past few decades, object oriented programming languages have become popular throughout the computer world with academia and industry. Languages such as Smalltalk became very popular with language researchers, while C++ and Java both have a secure foothold in the software industry. However, computer architecture has yet to catch up with the rapid growth of popularity in object oriented programming. In particular, memory and cache behavior in a conventional modern computer system is designed to optimize procedural programs with either integer sized or large data structures, where programs tend to exhibit strong spatial and temporal locality in memory access.

Object-oriented languages relying on garbage collection often get much less advantage from caches because the principal of spatial locality rarely holds. With a large number of small objects, one often gets 'false sharing' in cache lines between pieces of unrelated objects. Parts of an unrelated object are pulled from memory into the cache, just to go unreferenced, which means those bus cycles have been wasted. Furthermore, garbage collection algorithms often rely on copying objects and scanning significant portions of the heap in random orders.

> Such poor locality of reference leads to an excessive number of cache misses and page faults. [1]

Rather than suffering reduced performance, the Mayhem Project is designing a system with object-oriented languages in mind. The goal is to design and implement a system taking advantage of object oriented languages. The design of the system is based on the assumption of a 64-bit SPARC-like system designed to support a Java virtual machine; however it could easily be used in more general cases.

1.2 Concept of Object Memory

The critical change suggested by the Mayhem Project is to create an 'object memory system', where the underlying cache and memory hierarchy understands the concept of an object. Part of the address space (which is assumed to be larger than the currently-popular 32-bit systems) would be reserved for object addresses, which rather than representing the physical or virtual memory address space, represents an object ID (OID) and offset. Up to a point that is called the '*object boundary*', the system would understand both physical and object addresses. Past this object boundary, object addresses would have to be translated somehow to virtual or physical memory addresses.

Objects are assigned an OID at creation, but not immediately assigned a virtual address in memory. Instead, the active processor would tag cache lines corresponding to the object with the OID and part of the offset. Any references to this object can now be made using the OID and offset rather than a physical address. In fact, one can reference the object without ever allocating memory for it outside the cache.

Since the OID is part of the tag on the cache line, false sharing of object lines is entirely eliminated, though there might be some small amount of unused space in each line.

1.2.1 Garbage Collection

According to the weak generational hypothesis proposed by many garbage collection researchers, "most objects die young" [2]. By designing a garbage collector to target young objects separately, one can make these young garbage collections faster. This forms the basis of a generational garbage collector: newly created objects are stored separately from other objects, and are garbage collected independently of older objects. After a certain period of time, objects which have not died are moved to another 'generation', which is garbage collected infrequently relative to the young generation.

Since objects in an object memory system can be created in-cache, one can define a subsection of the cache hierarchy visible to a particular processor as a young generation of a generational garbage collector. We define the '*garbage collection boundary*' as the region inside which a processor can garbage collect this young generation.

If one can determine that there are no references to an in-cache object either inside or outside of the cache, it is considered to be dead. The scan for references inside the cache is fast due to the relatively quick speed of a cache compared to main memory. Furthermore, the deletion of the object only involves the updating of accounting tables and the clearing of the cache line. This is also very fast compared to traditional garbage collection systems which often rely on the relocation of large numbers of objects during each garbage collection.

1.2.2 The Warden

In-cache garbage collections lose much of their efficiency if main memory has to be read to search for references. It is therefore desirable to track if references have escaped past a garbage collection boundary. Since the hardware is aware of object references, this can be done in hardware by a device called the warden. A warden exists at each GC boundary, scanning all evictions through it. Any time it detects an object reference, it will broadcast a request to the caches to update the header of the line, if it is present, to mark it 'non-local'. These non-local lines cannot be garbage collected in-cache.

1.2.3 The Translator

As objects cross the object boundary, a device is needed to map object addresses to physical addresses and vice versa. The translator does exactly this. Using a tree structure of object tables, similar to a virtual memory page table, it can translate an object address into a physical address, or if necessary even allocate memory if there is no physical address. Similarly, when objects come in from memory, the translator will map the physical memory into object lines with the correct tags, and ensure that the header is tagged as non-local if it enters.

2 Simulating Object Memory

While it is easy to imagine the advantages object memory could bring, it is quite difficult to predict the behavior of such a system for real-world applications. Therefore, the ability to simulate the operation of such a system for real applications, while difficult to implement, is essential in determining the effectiveness of this approach.

2.1 Capturing a Program's Behavior

Since our target is Java programs, we have some flexibility in the way we simulate the system. One could:

- Build a Java Virtual Machine (JVM) for the system, and emulate the behavior of the backend hardware.
- Collect statistics from a standard JVM about the number of occurrences of different actions, and determine how our system would affect these.
- Capture the critical actions taken by the program, known as the 'mutator' in garbage collection terms, but discard information about how the JVM handled these actions.

While building a Java Virtual Machine for the system would eventually be necessary for a final system, modern JVMs are extremely complex, and too large an undertaking for a small research team. Statistics, on the other hand, while easy to collect, do not allow one to determine behavior on a cache level, which is something that is essential for us to observe.

By capturing and recording the actions taken by the mutator, one can simulate the results of that behavior on another system without implementing the full complexities of a virtual machine.

The Tracing Java Virtual Machine [3] is a modified version of a Java Virtual Machine version 1.2. It captures all the events that define a program's execution, such as creating

and deleting threads, setting and accessing memory, and stack manipulations. It does not record anything about internal operation, such as how memory is organized, or when a garbage collection occurs. This allows a simulator to determine the behavior for a system that does these things differently than the VM recording the trace.

2.2 Simulating a Program Using a Trace File

A generic system was built up around Tracing JVM trace files in Java, where one could pass a class to the tracing program, which would read, parse, and then pass the information to a simulator class. Figure 1 illustrates the flow of information from an application through to its simulation.

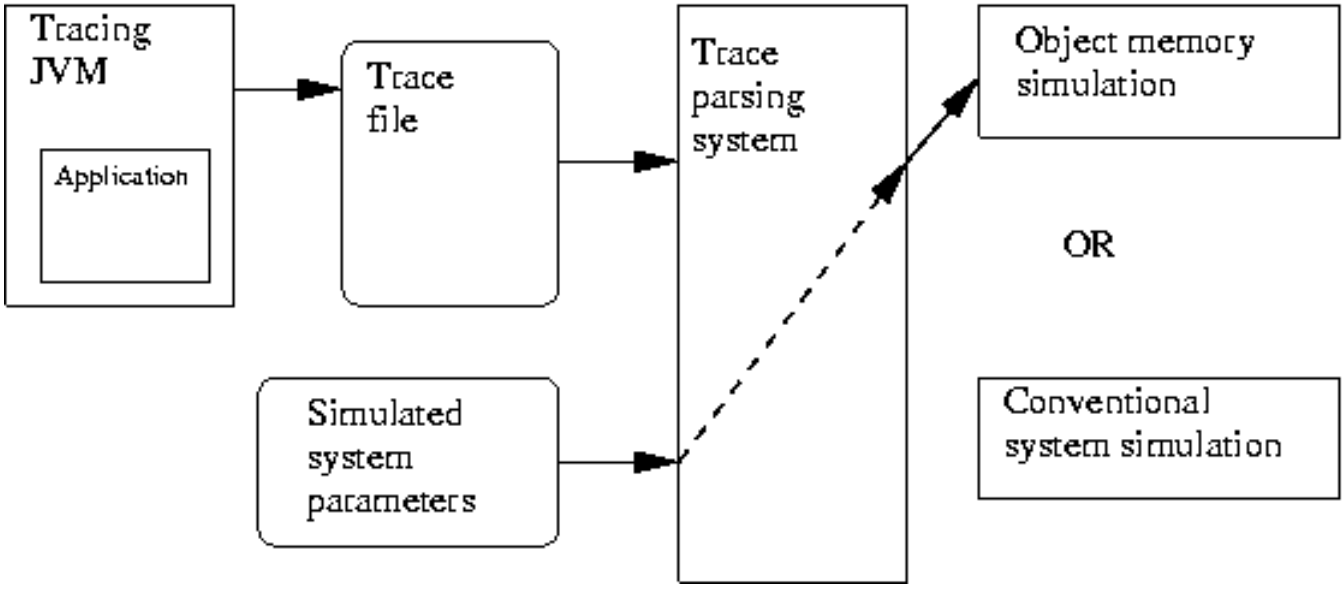

Figure 1: Flow of data between an application and a simulator

Classes were designed to represent two types of system: conventional, and object memory. Both were somewhat contrived, each having an option of one or two 64-bit processors, each with only one level of cache on each processor (and no shared level 2 cache). Included was a basic scheduling system and a simulation of a basic JVM's behavior for thread management. Most aspects of an actual JVM's operation need not be simulated because the trace is the result of a program having already been run on the Tracing JVM. The conventional system was given a generational garbage collector, while the object system was given the in-cache garbage collector (but nothing yet for collecting the object memory 'old-space').

Underlying these is a set of classes to model all the elements of a memory hierarchy, including several types of cache, interlinking busses, and a main memory. A great deal of useful information can be derived from these classes about cache behavior.

3 Visualizing the Tsim Simulator

3.1 Reasons for a Visualization

While having a simulator can aid in understanding how a system could be implemented, it only becomes truly useful once it can provide feedback. This is a very different task from simulating the operation of a system, and has a different set of challenges involved. With the Tsim system, a method of providing reasonable feedback is essential, both for debugging during the system design, and for observing the system behavior once the code is completed. However, this is not a simple task due to the amount of information available. Table 1 shows the massive amounts of data consumed by the simulator for some basic test suite programs. The output is at highest verbosity, averaged over three five-minute runs of operation on a dedicated machine. Running a simulation to completion can take many days at these rates, even for small traces. It is obvious that human consumption of this information would be impossible due to the sheer volume of text produced.

Table 1: Trace sizes and verbose output rate

Trace	**Simulation**	**Input Trace Size (MB)**	**Event Processing Rate (events / minute)**	**Average Output Speed (lines / minute)**
Hello, World!	Two-processor object simulation	30.1	1.71×10^5	6.21×10^5
Pretzel builder	Two-processor object simulation	183	1.80×10^5	6.09×10^5
SPECJVM javac (size 10)	Two-processor object simulation (gzip -1 decompression)	Compressed to 630	1.69×10^5	6.23×10^5
SPECJVM mpegaudio (size 10)	Two-processor object simulation (gzip -1 decompression)	Compressed to 9083	1.69×10^5	6.36×10^5

Even with heavy information filtering, the one inescapable fact is that there is a large amount of information going in, and that to communicate what is happening in text,

a great deal of information must be written out. Furthermore, text is not friendly for human consumption on such a large scale: while it may communicate correctly what is happening, it is difficult for a human to get an overview of the system from lists of occurrences or numbers. The ability to use pictures, graphs, maps, and other graphical elements, gives a human the ability to get a better intuitive feel for a system.

3.2 Adoption of GCSpy

The Mayehm Project decided to integrate a tool known as 'GCSpy' with the tsim simulation to gain insight into the operation of the simulator. GCSpy, in the words of the author, is "an architectural framework for the collection, transmission, storage and replay of memory management behaviour." [4] It allows the expression of a number of abstract concepts:

- **Space:** A region or type of memory. A space has a name, and common information (streams) presented across all of its elements.
- **Stream:** An individual type of information transmitted over a space. A stream consists of a single value for each block in a space. This value may be either an integer, or a string from a finite set.
- **Block:** The resolution limit for a space. Each block has exactly one value associated with it for each stream in the space.
- **Event:** An occurrence in a system that triggers the update of stream information. Events can be caused for any reason; critical update points, a timer, steps in a particular process, or any such point where information is retrievable in the system.

GCSpy features a client-server model. A simulation or running system acts as the server, providing stream information at each event to GCSpy clients. The clients are tools which collect this data and are responsible for displaying it.

Included with GCSpy are reference libraries for implementing the protocol and framework for the Java and C programming languages. Tools written in Java provide the ability to save and replay traces, as well as a simple graphical client for visualizations. A sample

screenshot from GCSpy visualizing a Java Virtual Machine with this graphical client is shown in figure 2.

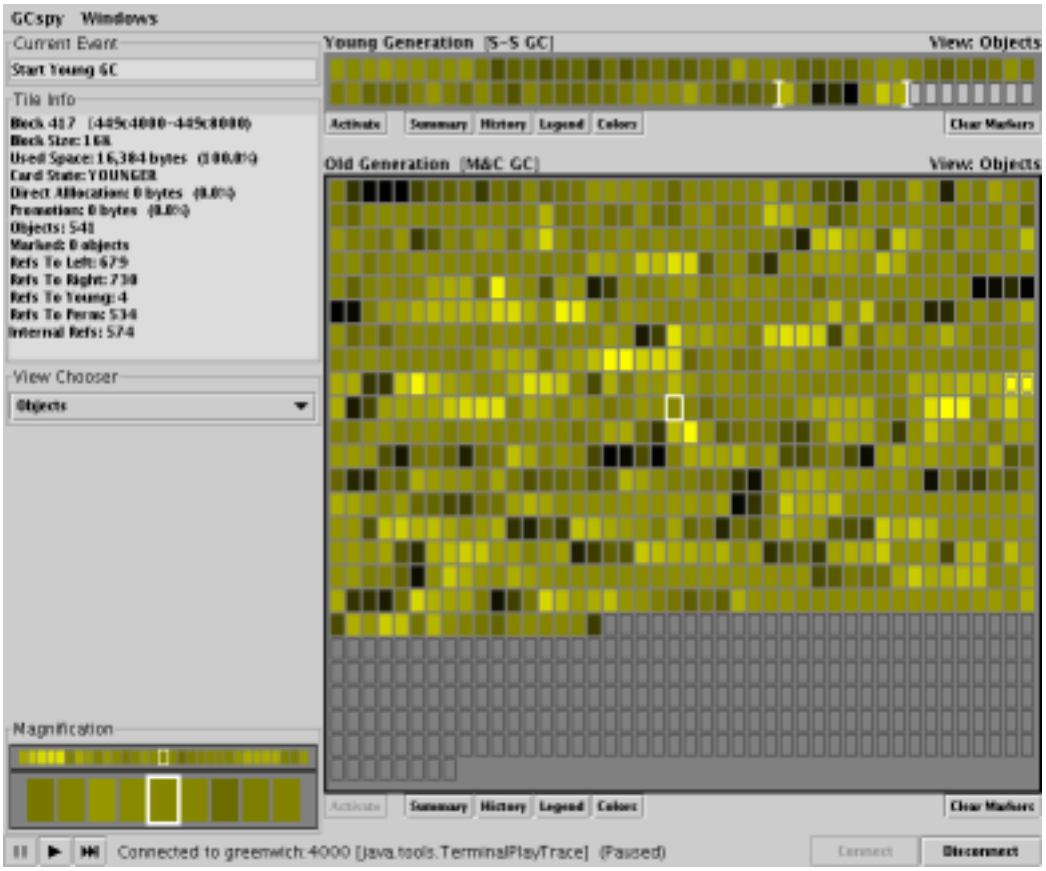

Figure 2: GCSpy graphical client visualizing a Java Virtual Machine [5]

3.3 Integration of tsim and GCSpy

The client-server model of GCSpy and availablity of Java libraries made it ideal for visualizing the object memory simulator. There were several desirable qualities for the GCSpy server in this case:

- The ability to include or not include the GCSpy server in a build of the simulator.
- Minimal performance penalty when not using GCSpy.
- Sufficiently quick performance using GCSpy that one could reasonably watch a visualization of a running simulator.

- Minimal code changes to tsim to incorporate GCSpy.
- No change in simulation results when GCSpy is added.

Several concepts were used to achieve these objectives:

- A set of 'forwarders' for major base classes for which there were several subclasses. Relying on the virtual function behavior of Java, these objects would pass the requests to the original simulator object, but also to other objects who requested certain pieces of information. The objects who registered with the forwarder were unable to modify the state of the simulation.
- A set of 'hooks' for important classes for which there are no subclasses in use. These objects are subclassed so that they can be given the ability to send information to registered objects, as well as inspect internal object state.
- An initialization system based on a parameter passed at runtime. By default, it used a default initialization class, which created the simulation objects. A class was provided which would add appropriate hooks and forwarders to properly visualize the simulation with GCSpy, and start the GCSpy server concurrently with the simulation.

This approach allowed the seamless runtime addition and removal of GCSpy without any runtime overhead when not using it, and an efficient system for collecting information when it was used.

3.4 Matching GCSpy Primitives to Tsim

To use GCSpy as a visualization tool, one must match concepts in a system to GCSpy primitives. Since GCSpy was initially designed to visualize memory managers, most of the concepts carried over cleanly.

3.4.1 Cache Spaces

A cache space is a physical view of a cache, with each block representing a certain number of cache lines. All caches (including the translation cache) can collect basic operating statistics:

- Cache hits
- Cache misses
- Cache line replacements

Caches that are in a hierarchy also can collect cache coherence information, such as owned line and exclusive line counts. Object-aware caches can do even more, with the ability to display:

- Object and non-object line counts.
- Number of object headers.
- Object lines with the object header residing in another cache.
- Number of lines belonging to non-local objects (and optionally to the objects these objects refer to).
- Cache-allocated lines (only for the caches closest to a processor).
- Object lines for which no physical memory has been allocated.

3.4.2 Backing Memory

This is not a view of the heap. Rather, it is a physical view of the areas of memory that support evicted object cache lines. Memory is handled in 'regions' that are allocated as a block, and filled using an incrementing pointer. Each block in this space represents a consecutive set of bytes of this memory, whose addresses are set as the block names. Regions are divided from each other by separator controls.

The following information can be shown about backing memory:

- Amount of memory that is currently allocated.
- How many times since the last event a particular block has had a pull from the cache hierarchy.

3.4.3 Object-Space

Rather than a physical representation, the object space allows one to look at the full set of objects in the system, and collect statistics about them. This information, especially the information summaries, can be used to determine the performance of various aspects of an object system.

The information collected consists of:

- Number of object IDs that currently exist over a range of objects.
- Count of objects which are flagged as non-local.
- Count of objects which have physical memory allocated to them.
- Number of times an object has been accessed in some way, regardless of source, since the last event.

3.4.4 Object Table List

The resolution of the object space makes it hard to visualize objects in great detail. The object table list shows information much like object space, but only for object tables. Each block represents a single object table, so detailed information on their contents can easily be displayed.

The information collected consists of:

- Number of object tables referred to in this object table.
- Number of objects (excluding other object tables) referred to in this object table.

4 Results of Integration

4.1 Information Reduction

An important part of the visualization was to heavily summarize and group information without losing aspects that are important to understanding the system. Table 2 shows the reduction of information by comparing the number of Tracing JVM events to the number of GCSpy events out and amount of raw data stored. GCSpy's event-recording utility was used in uncompressed mode to determine how much data was stored for events. The simulator in each case was run once for five minutes each.

Table 2: GCSpy information reduction

Trace	**Trace Events**	**GCSpy Events**	**GCSpy Data**
Pretzel builder	2,032,000	203	2,502,117
SPECJVM javac (size 10)	4,844,000	121	1,530,041
SPECJVM mpegaudio (size 10)	4,892,000	73	1,238,895

In all cases, several million events were reduced to a couple of hundred occurrences. Even including control data, the amount of information communicated is less than one byte per event. While this says little about the relevance of the stored information, or the importance of information lost, it does show that the data has been reduced to a manageable working set.

4.2 Performance Penalties

One downside to visualization is that it can slow down the execution of the simulator. Tests were run to determine the runtime impact of the GCSpy server.

The following test was run as the sole user on a four-processor Sun Fire 420R with 4GB of RAM, running Solaris 9. Debugging output was turned completely off to minimize IO blocking from debug output. In reality, a non-GCSpy simulator run would require a significant amount of debug output, but we do not model that here. The results, shown in table 3 are event counts per minute over three five-minute runs of the two-processor simulator (using gzip as required), with and without the GCSpy initializer.

The compressed traces show better performance than the uncompressed trace, most likely due to the greater time spent reading from the trace file over a network file system.

Table 3: Trace event counts with and without GCSpy

Trace	**Event Rate Without GCSpy (events/minute)**	**Event Rate With GC-Spy - no clients (events/minute)**	**Event Rate With GCSpy - two clients (events/minute)**
Pretzel builder	4.15×10^5	4.11×10^5	4.07×10^5
Compressed SPECJVM javac (size 10)	1.11×10^6	9.98×10^5	9.62×10^5
Compressed SPECJVM mpe-gaudio (size 10)	1.10×10^6	9.95×10^5	9.77×10^5

Table 4 shows the percentage decreases in performance using GCSpy without clients and with two clients over the unmodified simulation. The numbers seem to indicate that in most cases the penalty imposed by GCSpy will not be greater than 15%. On a sufficiently fast system, this overhead will not negatively affect simulation. It further indicates that in some cases this number will be even lower, as shown by the Pretzel builder trace, where the speed of reading the input file was the limiting factor instead of GCSpy.

Table 4: Performance reduction using GCSpy

Trace	**% Reduction in Event Rate (no clients)**	**% Reduction in Event Rate (two clients)**
Pretzel builder	0.964%	1.93%
Compressed SPECJVM javac (size 10)	10.1%	13.3%
Compressed SPECJVM mpe-gaudio (size 10)	9.55%	11.2%

4.3 Information Utility

While hard to measure concretely, the usefulness of the presented information is critical.

GCSpy very quickly revealed several critical bugs in the tsim simulation. Each of these

errors reveals an aspect of the simulation that was conveyed to the GCSpy user that was not apparent from debug output.

- Garbage collections were stopping after a small number of garbage collections (10-20 per GC region for most traces) due to failure to properly track simulation thread states. This appeared in GCSpy when garbage collection events stopped while memory allocation events continued.
- Stack frames were not being unallocated when a simulation function returned. As such, huge numbers of redundant stack frames built up. This showed up in GCSpy as an unusually-large space with thousands of small stack frames.
- The translator was not correctly hooked into the memory hierarchy. Memory was not being allocated for objects that were evicted, and object memory was being used in memory directly. This appeared in a lack of activity on object tables and backing memory despite heavy cache usage.

These reveal that a great deal of information is being communicated to the user that was not being communicated before.

5 GCSpy Advantages and Disadvantages

5.1 Advantages

GCSpy is excellent at reducing information to manageable levels. Many of the traces are fractions of a terabyte, and produce gigabytes of output in short periods of time. GCSpy provides the ability to summarize a large amount of information over time.

Organization of information is also greatly improved. Different types of information can be browsed by clicking, rather than having to search or filter text output. The ability to move backward and forwards in time quickly, and the ability to choose what information one wants to display allows one to quickly filter through large amounts (even after reduction) of information.

Graphical presentation, as demonstrated by the errors discovered with the tool, is very effective at communicating with humans.

The saving and replaying of GCSpy recordings allow one to replay the GCSpy view of a simulation as quickly as the user can view it. This recording can also be 'rewound', meaning it can be viewed backwards in time, which the simulator is incapable of doing on its own. This allows for batch runs of the simulator while presenting results to the user as if the simulation was currently executing.

5.2 GCSpy Limitations

Very little is provided in the way of graphing or time-based analysis tools. The 'history' feature of the graphical GCSpy client allows individual streams to be rendered at an extremely small scale, creating a graph-like image. These histories can be saved into graphical image files of the TIFF file format. Figure 3 shows a sample history for the backing memory space used memory stream, and cache space object header stream; while the backing memory image communicates an increasing memory demand, the cache history offers very little insight. These histories are acceptable for predictable spaces like backing memory, where allocations tend to increase with time until reaching

a compaction, while less-predictable spaces, like the cache space, just render random pixels. Summary data would be very useful to graph, for example the hit rate over time or total cache allocations compared to memory allocations, but there is no facility for this in the graphical client.

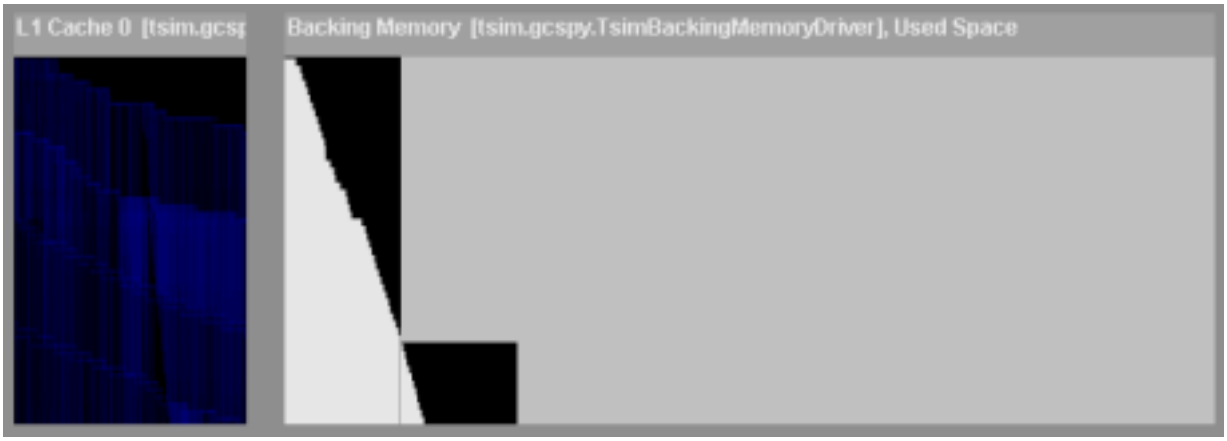

Figure 3: History view of a cache space and a backing memory space

No features exist in the provided GCSpy client for actually saving concrete data in a parsable format. One can view individual items in GCSpy, but if one wishes to do a comparison of the actual numbers, there is no way to do so in an automated fashion.

The protocol used in the reference GCSpy package is extremely inefficient. Almost no checking is done to see if data has been updated since the last event, resulting in the full set of data being sent for each event without any compression. The saved file format is much the same, except with the one advantage that compression can be applied to the results.

One issue encountered during development was that screen space on the graphical client became a factor. The issue was avoided by running two separate GCSpy servers. This allowed selective viewing of data, and cut down on the amount of data transmitted if only one was viewed. Unfortunately this also made the system harder to interact with, as pausing and step commands are not synchronized, and when replaying the data from a recording, the two recordings must be replayed independently.

Glossary

cache coherence The process by which a system with multiple caches prevents different processors from reading outdated values for a memory address when another cache contains an updated value.

cache line The basic unit transacted by caches. Consisting of a small number of consecutive bytes, a cache always reads and writes cache lines as a whole.

eviction A cache line is evicted when the location it is residing in a cache is needed for another cache line. The evicted line is either discarded or passed to another cache for storage.

GC *Garbage Collection* The operation of a piece of code that is responsible for seeking out objects which are no longer in use by a program, and making that memory the object was occupying available for reuse.

GC Boundary The border of a region of a cache hierarchy that a processor can garbage collect.

non-local A part of an object's state. A non-local object has either had its header evicted at least once past a garbage collection boundary, or has had a reference to it pass through a warden. Non-local objects cannot be garbage collected.

object boundary The border of the region in a cache hierarchy that understands object addresses.

principal of spatial locality The concept that after a memory access to one location, that a memory access to a nearby location is more likely than one far from the original address.

principal of temporal locality The concept that after a memory access to one location, that a memory access to a nearby location is more likely in the near future than farther in the future.

translator A device that is present at an object boundary, translating any object addresses into physical addresses, allocating memory for the object if necessary.

warden A device that is present at a GC boundary, marking objects as non-local when references to them pass through it.

References

[1] R. Jones and R. Lins, *Garbage Collection: Algoritms for Automatic Dynamic Memory Management*, John Wiley & Sons Ltd, West Sussex, England, 1996, p. 143.

[2] R. Jones and R. Lins, *Garbage Collection: Algoritms for Automatic Dynamic Memory Management*, John Wiley & Sons Ltd, West Sussex, England, 1996, p. 144.

[3] M. Wolczko, "The Tracing JVM", *The Tracing JVM*, http://research.sun.com/people/mario/tracing-jvm/index.html (current July 2002).

[4] T. Printezis and R. Jones, "GCSpy: An Adaptable Heap Visualisation Framework," *Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA 2002), Seattle, Washington, 2002.

[5] T. Printezis, "Monday, April 8, 2002," *GCspy*, http://www.dcs.gla.ac.uk/ tony/gcspy.www/16/index.html (current July 2002).