GarCoSim: A Framework for Automated Memory Management Research and Evaluation

Konstantin Nasartschuk
kons.na@unb.ca

Marcel Dombrowski
marcel.dombrowski@unb.ca

Tristan M. Basa
tbasa@unb.ca

Md. Mazder Rahman
p3sp4@unb.ca

Kenneth B. Kent
ken@unb.ca

Gerhard W. Dueck
gdueck@unb.ca

Faculty of Computer Science, University of New Brunswick, Fredericton, Canada

ABSTRACT

Many modern programming languages rely on memory management environments that are responsible for allocation and deallocation of objects. Garbage collection phases are used in order to detect inaccessible objects on the heap so they can be deallocated. The performance of garbage collection techniques depends heavily on the environment, implementation specific parameters and the benchmark used. The contribution of this publication is an extendable memory management simulator, which aims to assist developers in memory management evaluation and research. The simulator is capable of reading operations from a trace file extracted from a virtual machine and simulating the memory management needed by the simulated mutator. The framework aims to provide an isolated experimentation and comparison platform in the field of automatic memory management. New algorithms can be added to the framework in order to compare them to established algorithms.

CCS Concepts

•Software and its engineering → Garbage collection; *Allocation / deallocation strategies;* **•Computing methodologies →** *Modeling and simulation;*

Keywords

garbage collection, memory management simulation, trace file

1. INTRODUCTION

As programming languages evolved, memory management went through many developments, including the creation of an abstraction layer to hide the underlying complexity of resource management and software engineering. Automated memory management is one of the key features of many modern programming languages.

Explicit memory allocation, as used in C or C++, requires developers to manage all objects created on the heap explicitly. Objects have to be allocated when they are created and deallocated once the application no longer references them. Studies suggest that 20% of the time consumed by a project is used for memory management planning and optimization [11].

Automated memory management systems use garbage collectors that manage all objects allocated on the heap and free them once they become inaccessible. Garbage collection algorithms such as reference counting [5, 11], mark-sweep [15], and mark-compact [10] have emerged in order to detect unused objects. Most of these techniques are based on a stop-the-world concept. The managed application has to pause execution so the garbage collector can identify and free dead objects. The count and the duration of the stop-the-world phases are two of the main metrics indicating the performance of a garbage collector.

Attempts to optimize these metrics include divided heap structures, concurrency, and generational memory management structures [11]. Research and evaluation of algorithms used in automated memory management are implementation dependent and vary, based on the Virtual Machine (VM) utilized, the application that is being managed by the VM, and user defined parameters involved in the algorithm itself.

Currently, new GC algorithms are evaluated and compared using benchmark suites such as SPECjbb [6], DaCapo [2], SPECjvm [7] and others. This paper discusses an approach of creating an evaluation framework for garbage collection techniques which does not rely on a specific virtual machine. The idea of this project is to provide a memory management simulator, which replaces the VM with a simulation of its memory management capabilities. The application is replaced by a trace file which is extracted from an existing virtual machine.

In addition, the framework is developed in an extendable, open source form, so researchers can evaluate and compare new techniques to the results of others in an isolated environment. The memory management toolkit (MMTK) [1],

VALUETOOLS 2015, December 14-16, Berlin, Germany
Copyright © 2016 ICST
DOI 10.4108/eai.14-12-2015.2262678

which utilizes the Jikes Research Virtual Machine (Jikes RVM) [14], is written in Java and aims to assist researchers in prototyping virtual environment features in a simple virtual machine rather than removing the compilation of the application (mutator) code and a complete isolation of memory management.

A goal of the presented framework is to be simple. Each virtual machine offers a number of optimizations on a variety of levels, such as garbage collection, allocation, just in time compilation and object modelling. Comparing different techniques having such a vast number of parameters can be quite difficult and the search for the source of benefits or causes for disadvantages can take significant time.

The paper is divided into four sections. Section 2 provides a short introduction of automated memory management. To test the memory management simulator, trace files of real world applications were extracted from a JVM. The experimental setup used for this paper is described in Section 3. The extraction process and resulting trace files are topics of Section 4. The design and development of the memory management simulator that allows developers to implement garbage collection techniques is described in Section 5. Results achieved using the combination of trace files and the simulator are presented and discussed in Section 6.

2. BACKGROUND

As Java and other virtual machine-based languages continue to become more popular, the performance of those languages becomes more important. One major drawback of garbage collected languages is still inefficiency, especially for computationally intensive scientific applications [13].

Four memory management techniques build the core of the standard repertoire of virtual machines: Reference counting, Mark-sweep, Mark-compact, and Copying collection. The techniques, excluding reference counting, are introduced in this section.

2.1 Mark-sweep and mark-compact

The mark-sweep garbage collection policy treats the heap structure as a directed graph. Objects are connected using their references to each other. The roots of the graph are represented by pointers in registers, global variables, and variables existing on the stack.

The algorithm is divided into a marking and a sweeping stage. The policy requires a stop-the-world stage—at least for the marking stage [15]. Marking describes the traversal of the object graph and marking all objects reached as alive. Once the traversal is done, the sweeping stage can free the space used by dead objects. One downside of the policy is the stop-the-world phase itself, as the mutator is stopped completely and the time is used for memory management. The main problem of this policy is the fragmentation that occurs when objects are freed.

This drawback can be handled by replacing the sweep stage by a more complex compaction phase. Once all live objects are marked, the objects are compacted to remove all fragmentation from the heap. The space located after the last memory location used is marked as free. Compared to the sweep phase, the compaction requires more time. However, the benefit of not having fragmentation has a large impact on the number of garbage collections required in order to accommodate the mutator. The compaction phase can be initiated for every garbage collection or when the fragmentation becomes too high.

2.2 Copying collection

The natural extension of the mark-compact collection is represented by the copying collector [9, 4]. The idea behind this policy is to remove the marking stage and to only use a compaction phase. In order to accomplish this, live objects have to be copied during traversal. As it is not certain how much space and which addresses will be freed during the process, objects have to be copied into an entirely separate address space. This requires the heap space to be divided into two halves. One, where new objects are allocated and the second space where live objects will be copied during a garbage collection. The benefits are a very short stop-the-world time and the fact that no fragmentation is possible. The largest disadvantage is the fact that the effectively available heap space is half of the physical memory space.

2.3 Combination of policies

Mixed policies appeared that use the knowledge of common application behaviour. An example is the knowledge that many objects are small and die after a short time while large and old objects have the tendency to survive even longer. In order to use this knowledge, the generational garbage collection technique was introduced. The idea behind it is to divide the heap into a young and an old space or spaces. The space of allocation is therefore collected more often than the old space. In exchange, the allocation space is usually only a small part of the heap, keeping the stop-the-world times very short. Objects surviving long enough are promoted into the old space as they are expected to survive even longer.

3. EXPERIMENTAL SETUP

Measuring improvements in automated memory management (MM) performance requires at least the simulation of basic MM operations of a virtual machine (VM). An online simulation is considered expensive as it requires a copy of the VM and an actual run of the benchmark. An alternative approach is to run the simulation off-line using trace files. Trace files offer the flexibility of being able to develop a tailored simulator that focuses only on the basic MM operations and at the same time, be portable to be compiled and run on different platforms.

Trace files were acquired by instrumenting a Java Virtual Machine (JVM), particularly the memory management module. Instrumentation is done by inserting hooks in the JVM code that handle memory-related operations. For each of the operations, a line containing relevant information is written to a file. This output file is post-processed to conform to the standard format specified in Section 4.2. The post-processed file, or trace file, serves as input to the memory management simulator (MMS). It treats the trace file lines as mutator memory operations, allocates space, manages references and performs garbage collections in order to accommodate the requirements of the application.

4. JVM INSTRUMENTATION & TRACE

The JVM's C/C++ code was modified to log all basic memory management operations into trace files. In this section the MM operations that are captured at JVM runtime are discussed, followed by post-processing of trace files.

4.1 Basic Memory Management Operations

A real life JVM was instrumented to trace four basic MM operations that are described as follows:

4.1.1 Allocation

Most objects allocated by Java applications are short-lived [3]. Given the overhead of allocating an object from the general heap, also known as the *slow-path* allocation, this becomes a bottle-neck in system performance. To circumvent this, performance-oriented JVMs have provisions for assigning regions of memory for exclusive use of a thread. Objects that are identified as being short-lived are allocated from these exclusive regions, also known as *fast-path* allocation, thereby avoiding the need for expensive synchronization operations. Within both the slow-path and fast-path allocations, it is distinguished between single and indexable (arrays) objects. With fast-path indexable objects, it is distinguished between contiguous and discontiguous arrays. All in all, five types of allocation are identified. For every allocation, the information traced includes the names of threads handling the object allocation, the object id, the size of the object, the number of object reference slots, the class name of the object, the total instance size of the class, and the number of static fields of the class.

4.1.2 Store Access

Four different store accesses are as follows:

1. **Store primitives into an object:** This operation is performed when a variable of an object field is assigned/changed to/with data of a primitive type. The recorded data are thread name, object id, field size, field offset, and field type (volatile/non-volatile).

2. **Store references into an object:** This operation happens when an object references another object. Relevant information needed for this operation includes the object IDs of both the parent object and the child object, and the reference slot number. The JVM has implemented a write barrier whenever fields are going to be stored into. Hooks were added in these barriers to output relevant information such as the thread, the parent and the child object, and the object reference slot where the child object will be pointed from.

3. **Store primitives into a class:** This operation is performed when a static variable of a class field is assigned/changed to/with data of a primitive type. The recorded data are thread name, class name, field size, field offset and field type (volatile/non-volatile).

4. **Store references into a class:** This operation is similar to object-to-object referencing except that the object reference slot is a static field. The recorded information is thread name, class name, field offset/index, object id, field size and field type (volatile/non-volatile).

4.1.3 Read Access

An object is also instrumented whenever it (or one of its fields) is accessed. Information from this operation can be used for analyzing the effects of caching.

MM Operations	Notation
Allocation	a T_i O_j S_k N_l C_j
Add to rootset	+ T_i O_j
Store primitive into an object	s T_i O_j $I_x(/F_m)$ S_n V_o
Store object reference into an object	w T_i P_j $\#_k$ O_l F_m S_n V_o
Store primitive into a class	s T_i O_j $I_x(/F_m)$ S_n V_o
Store object reference into a class	c T_i C_j $\#_k$ O_l F_m S_n V_o
Read primitive or reference from an object	r T_i O_j $I_x(/F_m)$ S_n V_o
Read primitive or reference from a class object	r T_i C_j $I_x(/F_m)$ S_n V_o
Delete from a root set	- T_i O_j

T_i: Thread id ($i \geq 0$)
O_j: Object id ($j \geq 1$)
C_j: Class id ($j \geq 1$)
S_k: Size of object in bytes
N_l: Number of reference slots in an object
$I_x(/F_m)$: Field index(/offset)
S_n: Field size
V_o: Field type (volatile or non-volatile)
P_j: Parent object id
O_l: Child object id
$\#_k$: k^{th} slot of reference fields

Table 1: The trace file format.

4.1.4 Rootset Dump

Rootset dumps are a way of inspecting the thread's rootset. Since this should be done in a *stop-the-world* fashion, an asynchronous-handler is signaled whenever allocation is done. The timing of the actual dump depends entirely on the JVM. Rootset additions and rootset deletions are inferred between two rootset dumps in the next phase.

4.2 Post-Processing and Trace File Format

The instrumented JVM produces a file that contains raw information captured during the execution of an application. Some of this information can be converted to a format that is more simulator-friendly (e.g. conversion of unique identifiers from string to integer). In addition, rootset dumps need to be converted to a series of operations (denoted by '+' and '-') in as much as the trace file is interpreted as a sequence of operations.

Another purpose of post-processing is to defer rootset deletions. At some point, some objects become garbage in the trace file but somehow manage to appear again at some later operations. These objects are allocated by the native code and are not captured in the JVM C++ code. Some approaches move them into what is called *zombie* regions to be resurrected again at some later time [8]. The deletion is deferred from the rootset of its allocating thread until its last access. This prevents it from being collected at least until its last access. Table 1 shows the format of the trace files used in the experiments. Some attributes of classes are captured in class lists. Attributes of a class are denoted by C_i I_j $\#k$ **className**, where, i, j and k represent the class id, class object size in bytes, and number of static references.

4.3 Results of Trace Files

Experiments are performed with both DaCapo [2] and SPECjvm2008 [7] benchmark suites. When running the instrumented JVM, the *Just-in-time* (JIT) compiler as well as the packed object feature were disabled. The *optthruput* GC policy was used to disable concurrent GC, and specify only a maximum of one GC thread to run. The heap size was set to 16GB. The output is referred to as a *raw* trace file that contains operations with relevant information about threads, classes as well as root set dumps. Post-processing results in a trace file with the standard format as specified in Section 4.2. Trace files for eight workloads of DaCapo and for 20 workloads of SPECjvm2008 benchmark suite were

Ben	Workloads	$\#O(m)$	$\#T$	$\#C$	Memory Operations		
					$\#Alloc$	w $\#S_{o\leftarrow o}$	c $\#S_{c\leftarrow o}$
DaC	luindex	829.24	6	685	198,526	8,574,950	1,517
	batik	1,015.73	17	1,653	1,070,877	7,693,402	14,980
	avrora	2,706.08	15	887	1,950,313	22,851,625	1,677
	fop	283.52	10	1,492	2,951,921	6,351,197	2,748
	pmd	816.61	14	1,029	7,059,936	33,691,819	1,890
	jython	4,351.83	10	2,592	42,827,144	133,321,284	6509
	lusearch	3,203.79	21	639	13325559	56945210	1448
	xalan	2,872.67	21	920	6,623,561	41,836,363	7,506
SPEC	*.sunflow	233.57	20	1360	334,032	597,619	4,111
	*.compiler.sunflow	235.04	15	1418	333,354	646,308	4,086
	*.crypto.rsa	234.16	20	1420	330,947	618,608	3,987
	*.crypto.aes	234.38	20	1420	331,363	625,008	3,999
	*.compress	233.46	20	1419	332,047	595,687	4,035
	*.crypto.signverify	234.21	20	1420	331,266	619,195	3,991
	*.mpegaudio	233.41	20	1419	331,414	594,022	4,011
	*.scimark.fft	233.98	20	1420	330,387	613,605	3,987
	*.scimark.lu	233.43	20	1420	331,174	595,948	4,027
	*.scimark.monte_carlo	234.16	20	1419	330,363	619,607	3,991
	*.scimark.sor	233.44	20	1420	331,407	595,314	4,043
	*.scimark.sparse	234.54	20	1420	331,893	631,234	4,035
	*.serial	233.45	20	1419	332,044	595,440	4,035
	*.xml.transform	233.49	20	1419	331,907	596,337	4,011
	*.xml.validation	233.63	20	1419	334,109	598,955	4,115
	compress	3,736.55	16	1,366	331,802	589,686	3,889
	scimark.sor.small	4,816.08	16	1359	329,379	587,526	3,858
	scimark.fft.small	2,434.52	16	1,359	334,936	59,0631	3,889
	scimark.lu.small	6,563.88	16	1,358	367,559	718,042	3,848
	xml.validation	1,023.26	16	1,594	2,419,200	28,176,696	4,406

Ben: Benchmark
Dac: DaCapo
$SPEC$: SPECjvm2008
*: startup
$\#O(m)$: No. of memory management operations in million
$\#T$: No. of threads
$\#C$: No. of classes
$\#Alloc$: No. of objects allocated
$\#S_{o\leftarrow o}$: No. of operations for storing a non-static object reference into an object
$\#S_{c\leftarrow o}$: No. of operations for writing a static object reference into a class

Table 2: Metrics of the generated trace files.

generated. Metrics of the generated trace files are shown in Table 2. Column 3 shows the total number of memory management operations captured in each workload running in the JVM. The number of operations that are most relevant to the GC simulator, such as allocations, stores of object references into other objects and stores of object references into class objects are shown in column 6, 7 and 8 respectively. Empirical results show that 98% of all operations are read access operations that can be used in further research to simulate cache misses, object cache locality, etc.

5. MEMORY MANAGEMENT SIMULATOR

In order to isolate the memory management aspect of a virtual machine from the mutator execution for evaluation purposes, part of this project was the development of a memory management simulator. The input of the simulator is a trace file containing object allocations and reference operations.

The memory management simulator, developed in C++, utilizes an object-oriented and iteration-based approach. The goal of this development process was to ensure a high extensibility of the framework. At the core of the model are classes representing objects, a memory manager, a garbage collector and the allocator. Even though the memory management simulator was created with a monolithic heap structure in mind, the structure allows developers to create divided heap policies. The framework, including the simulator, the trace generator and their source code are publicly available on GitHub [12].

5.1 Simulator Structure

Figure 1 shows the basic class structure of the memory management simulator. The `MemoryManager` uses the `Allocator` module for object allocation and keeps track of all existing objects in order to create and maintain references between

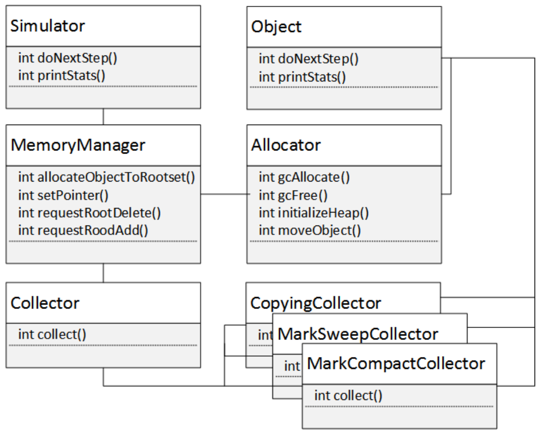

Figure 1: Garbage Collector architecture consisting of MemoryManager, Allocator, Collectors.

them. If an allocation fails or another metric signals that a garbage collection should be initiated, the `MemoryManager` commands the `Collector` to do so. The specific implementation of the `Collector` determines which objects are not referenced any longer and requests a free operation by the allocator.

Only the `MemoryManager` class can allocate and only the `Collector` deallocate objects. The mutator can never request the deletion of an object and the garbage collector can never allocate a new object. The only exception to this is the moving process of existing objects during the copying or compacting stage of different collection policies.

A dynamic object container keeps track of all objects that exist on the heap. This includes active and used objects as well as dead objects, which are not yet freed. The root set is used to initiate an object traversal algorithm to locate all live objects. The container can be iterated afterwards in order to find dead objects and to free them, depending on the collection policy. Alternatively, collections and heap organization can be performed without the use of an object list but rather by iterating over the objects on the heap itself using a base pointer and object sizes.

The classes `Allocator` and `Collector` are interfaces which can be specified using one of the specific policy implementations developed for it. Allocator implementations include first fit and next fit allocation. GC policies available are mark-sweep, mark-compact, and copying collection.

Additional `Allocator` and `Collector` classes can be introduced by implementing the interface calls defined by the parent classes. Examples for both cases are included in the repository. If a changed heap structure is being investigated, changes to the `MemoryManager` class might be necessary.

The flow of the program starts with the `Simulator` class, which is responsible for simulation control and input handling. Its main task is to read in memory instructions from the trace file and to advice the `MemoryManager` which action is currently requested by the mutator. The `MemoryManager` class represents the orchestration module of the application and routes requests to instances of `Allocator` or `Collector` classes depending on metrics, trace file request and the success or failure of allocations. The `Object` class includes metrics and information about one specific object on the heap.

It includes the address, object id, creation time and other values, which could be of interest when researching automatic memory management. Additional information can be added if needed.

6. EXPERIMENTAL RESULTS

In order to evaluate the simulator, ten trace files extracted from the JVM were processed using the collection policies included in the current version of the simulator. The main metrics for the evaluation of the simulator were the count of garbage collections, the average collection time, and the execution time of the full benchmarks.

As each benchmark has an individual heap structure, the heap size for each run of the benchmarks was increased in 25MB steps and the change in execution time was recorded. A watermark metric, which triggers a GC before the heap is filled, was not used in any of the tests in order to stress the allocator and garbage collection policies. The machine used for the benchmarks utilized an Intel Core i7-2600 processor and had 8 GB of RAM.

Table 3 shows the total execution time for each benchmark. All combinations of heap size and collection policy are listed. It can be observed that in general, the mark-compact policy requires the smallest heap as it does not suffer from fragmentation and does not divide the heap into two separate parts. The copying collection requires the second largest heap and almost always, mark-sweep requires the largest heap in order to be able to run. An extreme case for this metric was the `pmd` benchmark, which executed successfully using a 100 MB heap when using mark-compact or the copying policy, but required a 250 MB heap in order to finish utilizing mark-sweep garbage collection.

Benchmarks such as compress show the cost of garbage collection. While stop-the-world phases are required, the execution time of the simulation is 25 seconds slower compared to a heap which never becomes full during run time. In addition, this particular case shows once again, that this point is reached with a much larger heap when the copying policy is used.

As described in literature, copying collection was observed to deliver the fastest execution results due to small average collection times [11]. Mark-sweep (when a run finished successfully) had the second fastest results. The compaction phase of the mark-compact policy slowed down garbage collections.

The number of garbage collections for each policy were collected using a 100 MB heap as the heap size allowed for all but two tests to finish while the heap was still small enough to require stop-the-world phases. The last column in Table 3 lists the average garbage collection time for each benchmark when using a heap size of 100MB. The average collection count is listed in parentheses. Those two metrics can be used in order to determine the raw garbage collection time of the benchmark.

An example of garbage collection times and their comparison is the `scimark.lu.small` benchmark. Even though the collection policy required two collections in order to accommodate the memory needs of the benchmark, the stop-the-world phases were only about 3 seconds on average. The mark-sweep policy had to halt the application for 9.6 seconds and the mark-compact phase due to its compaction phase 12.2 seconds.

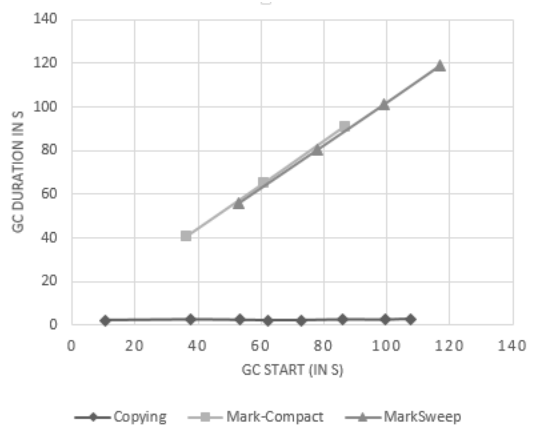

Figure 2: GC start during runtime and its duration when running xml.validation with a heap of 75MB.

The simulator presented is capable of not only comparing different runs using different parameters, but can also be used in order to investigate a single simulation process in more detail. An example would be the investigation of GCs when using the same heap size. In particular the time when a GC happens and how long it takes during the execution process. Using the result log files of the simulator, it is possible to retrieve this data and visualize the information as shown in Figure 2.

The results Figure 2 are as expected. As the copying collector splits the heap in two, the active heap part fills up quickly and collections are required earlier compared to other techniques. On the flipside, the time needed for each collection is constant no matter how many live objects remain on the heap. This is not the case when looking at the graphs of the mark-sweep and mark-compact policies. As the number of live objects and therefore the heap size usually increases, the stop-the-world times of mark-sweep and mark-compact increase as well. This underlines the statement, that a frequently filled heap usually benefits from a copying collection [11].

7. CONCLUSIONS AND FUTURE WORK

This paper presented the design and evaluation of a garbage collection simulator which aims to be used for automated memory management research. The structure of the simulator and the process of extracting real application input files for the framework from a JVM was described in detail and the execution and results gathered were discussed in order to show possible applications for the framework.

The simulator is under heavy development. Future work in this project include the development of a trace file generator. It can be used to mimic the memory requirement behaviour of real life applications and to create trace files of any size without the need of extracting them from a virtual machine.

Another field of research is the implementation of additional garbage collection techniques. This includes split heap structures such as generational garbage collection, thread local garbage collection, and others.

Benchmark	Policy	50M	75M	100M	125M	150M	175M	200M	225M	250M	Avg GC time (count), 100M
avrora	C	121.28	114.193	126.047	120.397	110.032	104.892	103.865	78.715	74.651	12.466 (2)
	MC	228.448	161.843	175.648	86.414	78.216	75.481	80.642	81.417	89.291	86.082 (1)
	MS			278.804	75.462	77.671	78.51	79.369	78.688	73.21	91.274 (2)
batik	C		87.864	74.575	72.282	64.501	58.467	59.14	60.998	59.906	7.751 (3)
	MC	164.921	136.528	80.491	100.637	41.526	33.3	33.425	34.083	36.259	37.363 (1)
	MS				70.516	38.794	33.575	35.104	33.486	34.582	
compress	C		32.752	28.223	22.838	24.648	26.661	10.266	10.748	10.921	4.671 (2)
	MC	46.14	35.093	9.193	9.331	9.795	9.942	9.847	10.625	11.045	0 (0)
	MS	32.199	23.594	8.943	8.426	9.765	10.14	10.262	9.801	10.916	0 (0)
fop	C	116.718	102.432	98.09	82.571	80.282	87.989	88.514	80.876	82.417	6.524 (4)
	MC	245.933	225.998	171.13	93.13	110.713	117.605	127.162	45.862	45.412	52.806 (2)
	MS	351.722	178.896	143.116	93.408	93.838	100.419	112.677	45.674	45.949	43.985 (2)
luindex	C	30.358	25.596	23.609	26.655	26.076	28.095	26.549	21.981	29.355	0 (0)
	MC	22.848	27.86	27.248	26.615	27.029	26.912	26.421	27.676	28.042	0 (0)
	MS	28.29	26.515	20.535	26.243	27.041	27.272	35.583	26.566	27.578	0 (0)
pmd	C		408.8	292.74	334.743	314.149	291.701	239.141	209.069	216.057	4.975 (17)
	MC			818.403	718.756	735.743	574.013	341.939	368.925	392.716	107.926 (6)
	MS									343.782	
scimark.fft.small	C	27.707	23.477	19.78	21.572	23.541	25.303	10.131	10.487	10.768	6.295 (1)
	MC	22.656	28.955	9.147	9.232	9.531	9.807	10.18	10.454	10.831	0 (0)
	MS	19.187	23.735	9.119	9.182	9.458	10.002	10.125	10.432	10.803	0 (0)
scimark.lu.small	C	24.434	22.062	20.075	18.661	19.149	19.416	20.676	22.302	8.623	3.266 (2)
	MC	32.042	20.913	25.298	11.349	8.095	7.941	8.483	8.59	8.623	12.2 (1)
	MS	26.038	18.606	20.08	7.154	10.422	8.236	8.323	9.402	8.716	9.599 (1)
scimark.sor.small	C	11.753	10.391	11.374	5.124	5.413	5.518	5.782	6.004	6.324	3.286 (1)
	MC	15.061	4.657	4.973	5.308	5.376	5.559	5.823	6.06	6.468	0 (0)
	MS	12.125	4.951	4.948	5.386	5.364	5.589	5.753	6	6.184	0 (0)
xml.validation	C	180.149	135.206	132.794	158.282	169.955	161.932	127.563	153.944	139.763	3.643 (6)
	MC	735.03	330.131	273.436	338.09	227.003	280.423	203.744	193.258	199.035	83.621 (2)
	MS		502.177	325.567	315.409	223.821	222.375	221.177	179.534	186.048	76.743 (3)

Table 3: Execution time of the simulation in seconds. Each benchmark was run with different heap sizes and different collection policies (Copying, Mark-Compact, and Mark-Sweep). An empty cell stands for an insufficiently large heap. The last column lists average GC times when using a heap size of 100 MB. The times are listed in seconds.

8. ACKNOWLEDGMENTS

The authors would like to acknowledge the funding support provided by the Atlantic Canada Opportunities Agency (ACOA) through the Atlantic Innovation Fund (AIF) program. Furthermore, we would also like to thank the New Brunswick Innovation Fund for contributing to this project. Finally, we would like to thank the Centre for Advanced Studies—Atlantic for access to the resources for conducting our research.

9. REFERENCES

[1] S. Blackburn, R. Garner, and D. Frampton. Mmtk: The memory management toolkit, 2006.

[2] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, et al. The dacapo benchmarks: Java benchmarking development and analysis. In *ACM Sigplan Notices*, volume 41, pages 169–190. ACM, 2006.

[3] S. M. Blackburn, R. Jones, K. S. McKinley, and J. E. B. Moss. Beltway: Getting around garbage collection gridlock. In L. J. Hendren, editor, *Proceedings of PLDI'02 Programming Language Design and Implementation*, pages 182–196, Berlin, June 2002. ACM Press.

[4] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, 1970.

[5] G. E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3(12):655–657, 1960.

[6] S. P. E. Corporation. SPECjbb2013. https://www.spec.org/jbb2013/ [Online. Last accessed: 2015-06-08], 2015.

[7] S. P. E. Corporation. SPECjvm2008. https://www.spec.org/jvm2008/ [Online. Last accessed: 2015-06-08], 2015.

[8] S. Dieckmann and U. Hoelzle. A study of the allocation behavior of the specjvm98 java benchmarks. Technical report, Santa Barbara, CA, USA, 1998.

[9] R. R. Fenichel and J. C. Yochelson. A lisp garbage-collector for virtual-memory computer systems. *Communications of the ACM*, 12(11):611–612, 1969.

[10] C. H. Flood, D. Detlefs, N. Shavit, and X. Zhang. Parallel garbage collection for shared memory multiprocessors. In *Java Virtual Machine Research and Technology Symposium*, 2001.

[11] R. Jones, A. Hosking, and E. Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall/CRC, 2011.

[12] K. B. K. K. Nasartschuk, M. Dombrowski and G. Dueck. GarCoSim repository website. https://github.com/GarCoSim [Online: last accessed: 2015-06-08], 2015.

[13] J. Mathew, P. D. Coddington, and K. A. Hawick. Analysis and development of java grande benchmarks. In *Proceedings of the ACM 1999 conference on Java Grande*, pages 72–80. ACM, 1999.

[14] J. RVM. Jikes RVM Home. http://www.jikesrvm.org [Online. Last accessed: 2015-06-08], 2015.

[15] R. A. Saunders. The lisp system for the q-32 computer. *The Programming Language LISP: Its Operation and Applications*, pages 220–231, 1974.