MTM^2: Scalable Memory Management for Multi-Tasking Managed Runtime Environments

Sunil Soman Chandra Krintz[1] Laurent Daynès[2]

[1] Computer Science Department
University of California, Santa Barbara
{sunils,ckrintz}@cs.ucsb.edu
[2] Sun Microsystems Inc.
laurent.daynes@sun.com

Abstract. Multi-tasking, managed runtime environments (MREs) for modern type-safe, object-oriented programming languages enable isolated, concurrent execution of multiple applications within a single operating system process. Multi-tasking MREs can potentially extract high-performance on modern desktop and hand-held systems through aggressive sharing of classes and compiled code, and by exploiting high-level dynamic program information for optimization across program executions.

In this paper, we investigate the performance of a state-of-the-art multi-taking MRE for *concurrent program execution* and find that due to limited support for multi-tasking and performance isolation in the memory management subsystem, multi-tasking performs poorly compared to a production-quality, single-tasking MRE. To address this limitation, we present MTM^2: a comprehensive memory management system for concurrent multi-tasking. MTM^2 facilitates performance isolation and efficient heap space usage through on-demand allocation of application-private regions. Moreover, MTM^2 mitigates fragmentation using a novel hybrid garbage collector that combines mark-sweep with opportunistic copying. Our empirical evaluation shows that MTM^2 improves overall performance, scalability, and footprint for concurrent workloads over state-of-the-art, multi- and single-tasking MREs.

1 Introduction

As desktop and hand-held platforms become more capable (faster multicore CPUs, larger memories, etc.), users increasingly expect more from the software they execute. In particular, users that once executed a single program at a time, now demand that these systems *multi-task*, i.e, seamlessly and simultaneously execute multiple applications (such as, instant messaging, calendar and email clients, audio player, Internet browsers, office suite, etc.). Concurrently, developers of these applications commonly employ high-level, type-safe, portable programming languages (e.g. Java^{TM} and the Microsoft .Net^{TM} languages) for their implementation, since these languages offer high programmer productivity, portability, rapid deployment, and support for verification of safety properties. Programs in these languages are encoded by a source compiler into an architecture neutral format that can be executed on any system

with a managed runtime environment (MRE) for the format. To address both of these demands and to better utilize the underlying resources on modern desktops and hand-held systems, modern MREs have emerged with multi-tasking extensions [5, 4, 26, 28].

Multi-tasking MREs address isolation and resource management for multi-application workloads and provide application developers with a first-class representation of an isolated program execution (e.g., the *isolate* in [15, 5] and the *application domain* in *.Net* [19]). This representation provides the necessary functionality to launch and control the life cycle of multiple, isolated execution units (programs).

MREs have access to high-level program information, can potentially monitor time-varying program behavior and resource requirements, and can dynamically optimize programs as well as the runtime based on prior information. Therefore, they offer potential for more intelligent scheduling and resource management of programs. Prior work has shown that multi-tasking is more effective at enabling cross-program sharing of dynamically loaded and compiled code, and at achieving smaller memory footprint and faster startup times [4, 6] than traditional MREs that rely on process-based isolation. Yet, little attention has been directed at the *performance of multi-tasking MREs for simultaneous program execution*, i.e., concurrent workloads, compared to a more common scenario in which each program runs in its own process.

Figure 1 shows the results from a set of experiments that we have conducted to compare MVM [4, 26], a state-of-the art multi-tasking JVM from Sun Microsystems, with the single-tasking JVM (the Sun Microsystems HotSpot virtual machine version 1.5.0) from which the MVM is derived. The programs are a subset of the benchmarks that we use for our evaluation (that we describe in detail in Section 4) that exhibit significant garbage collection (GC) activity for the old generation (the longer-lived region). The figure shows that the MVM significantly degrades execution performance for concurrent workloads (2, 5, and 10 concurrent program instances in this graph), despite the significant opportunity for sharing (i.e. multiple versions of the same program are executing concurrently).

The MVM prototype that we use in this study is based on prior work [26] and achieves partial performance isolation across applications, reclamation of an application's heap memory upon task termination without having to perform GC, per-application accounting of heap usage, and per-application control of heap size settings. However, our results indicate that the prior state-of-the-art fails to perform favorably compared to its single-tasking counterpart for concurrent workloads that fully exercise the memory management system. The key impediment to scalability is the lack of GC performance isolation and a poorly performing full-heap GC algorithm.

To address these issues, we propose a novel memory management approach, which we call *multi-tasking memory manager (*MTM^2*)*. MTM^2 provides better GC performance isolation between programs while preserving other benefits of multi-tasking (small aggregate footprint, fast startup and sharing of classes and dynamically compiled code). MTM^2 is a generational GC system [29] that employs per-application young generation collection from [26] and introduces a novel *hybrid* approach to old generation collection that (i) maintains the constraint that all live objects within a region belong to the same application (which is key to GC isolation and the accu-

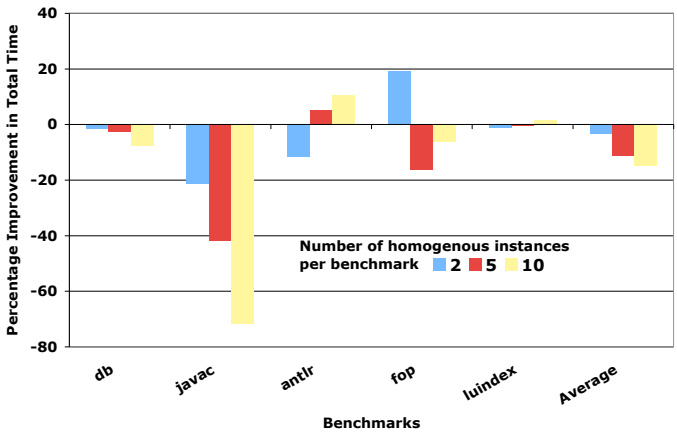

Fig. 1: Performance of a state-of-the-art multi-tasking MRE (MVM) versus multiple instances of the Java^{TM} HotSpot virtual machine for *concurrent* execution of five community benchmarks. No prior work has performed such an evaluation. Our analysis of MVM reveals that the key bottleneck to multi-tasking performance is memory management. MVM enables significant sharing and fewer OS processes, but this benefit does not outweigh the lack of performance isolation in the garbage collection subsystem.

racy of tracking per-application heap usage), (ii) ensures that the aggregate footprint of the multi-tasking MRE is small for concurrent workloads, and (iii) enables space reclaimed through opportunistic evacuation of objects from sparsely populated regions of one program to be made available to other programs. To achieve these goals, MTM^2 performs full collection of a single program's heap in isolation with co-located concurrent programs by combining fast, space-efficient, mark-sweep collection for regions with little fragmentation, with copying collection for regions with significant garbage and fragmentation. MTM^2 combines and extends a large body of recent GC research [8, 26, 25, 2, 20] to facilitate scalability via hybrid collection of independent applications in a multi-tasking MRE.

We have integrated MTM^2 with a prototype of MVM described in [26], and have used it to compare the execution of multiple programs executed using a single multi-tasking MRE versus using multiple concurrent instances of single-tasking MREs (one per program). Two metrics are particularly interesting with respect to the scalability of the two approaches: the overall footprint when executing multiple programs, and the execution times of programs. We demonstrate that on average, MTM^2 achieves better overall execution times and footprint versus its single-tasking counterpart, for concurrent workloads using a number of community benchmarks. Moreover, MTM^2 is able to do so while maintaining the other benefits of running with a multi-tasking MRE. MTM^2 outperforms the HotSpot single-tasking MRE by up to 14% on average for concurrent instances of the same program (homogeneous), and by up to 16% on average for workloads with a mix of programs (heterogeneous). MTM^2 achieves up to 41% reduction in footprint on average for homogeneous workloads, and by up to 33% on average for heterogeneous workloads over the single-tasking MRE. Finally,

we show that MTM^2 outperforms an extant state-of-the-art multi-tasking MRE by 10% to 22% for concurrent workloads.

In summary, we contribute the following:

- the first study that compares multi-instance JVM execution vs multi-tasking execution for concurrent program execution;
- a complete memory management system that provides full GC performance isolation for multi-tasking MREs;
- the design and implementation of a hybrid, multi-tasking aware GC that combines GC approaches that are well understood, i.e., mark-sweep and copying, to balance GC performance and memory footprint. Hybrid GC re-uses reclaimed space across multiple isolated program executions; this design achieves footprint-aware memory management that facilitates runtime efficiency for concurrent workloads;
- an empirical evaluation that shows that multi-tasking MREs when equipped with appropriate mechanisms for GC performance isolation, compare favorably to single-tasking MREs with respect to footprint and program execution time for concurrent workloads. This result further strengthens the case for multi-tasking MREs.

In the sections that follow, we first detail the design and implementation of MTM^2. We then present our experimental framework and empirical results in Section 4. We compare and contrast related work in Section 5. Finally, we conclude with a summary of our findings and contributions in Section 6.

2 Multi-Tasking Memory Management

MTM^2's design enables GC to be performed for a given application in isolation, and concurrently with respect to the mutators (threads modifying heap objects) of other applications.

MTM^2 follows the generational design [29] and each application is provided with a private two-generation heap. As with prior versions of MVM, a third generation, called the permanent generation, is shared across applications. The permanent generation is used to allocate long-lived meta-data, such as the runtime representation of classes (including method byte codes, constant pools, etc.), symbols and interned strings, and data structures of the MRE itself, all of which may be transparently shared across programs. The meta-data stored in the permanent generation may survive the execution of many programs, and is rarely collected.

The permanent generation is a single contiguous area. Memory for the young and old generations of applications originates from two pools of fixed-size regions managed by MTM^2. Each pool uses its own region size. The two pools and the shared permanent generation are contiguous in virtual space, such that old regions are in between the young regions pool and the permanent generation.

Memory for the young generation of a program is allocated at program startup, by provisioning a region from the young generation pool. Memory for a program's old generation is allocated on demand, on a per-region basis, from an *old region pool*. Thus, old and young generations are both made of one or more regions, that are possibly disjoint in virtual space. Regions are made of an integral number of

operating system virtual pages and aligned to page boundaries to enable on-demand allocation/deallocation of the physical pages allocated to regions by the operating system[3]. Backing storage for the virtual pages of a region is allocated only upon allocation of the region to a program. Conversely, when a region is returned to the pool, the backing storage for its virtual memory pages is freed immediately.

A region can only contain objects allocated by the same program, i.e., a region is always private to a program. This constraint facilitates both tracking of program memory usage and instantaneous, GC-less, reclamation of space upon program-termination [26]. It also helps performance isolation since GC only needs to synchronize with the threads of a single application (instead of all applications).

Following standard generational GC practice, programs allocate primarily from their young generation. Threads of a program are each assigned a thread local allocation buffer (TLAB) [14, 9, 16] from the corresponding program's young generation. TLABs satisfy most allocation requests with a simple, non-atomic, increment of a pointer of the TLAB's allocation hand (bump-pointer).

Tracking of cross-generation references uses a card-marking scheme [3, 12, 11, 1]. Old regions are card-aligned and consist of an integral number of cards, so that young generation collection of an application only needs to scan the dirty cards that correspond to the old regions allocated to the application. These are maintained by MTM^2 in a per-application list ordered by increasing virtual address. Each application is also associated with a *current* old region, which identifies the region used to allocate tenured space for the applications. Tenured space is allocated primarily during young generation collection, when promoting young objects, and occasionally, directly by mutator threads of the application to allocate space for large objects.

MTM^2 initiates a young generation collection for an application when the application's young generation is full, and an old generation collection when the application reaches its maximum heap size limit, or when allocation of a region from the pool of old region fails. Minor collection for an application is performed concurrently with respect to other applications using mechanisms described previously [26].

Collection of the old generation of an application's heap space follows a *hybrid* approach that combines fast, space-efficient, mark-sweep for regions of the old generation with little fragmentation or garbage, with a copying collection for regions of the old generation with either significant fragmentation or with a significant amount of garbage. Old generation collection is on a per-application basis, i.e., only the old generation of the application that triggers GC is collected.

MTM^2 also exploits MVM's representation of classes to organize the permanent generation in a way to limit tracing, during young and old generation collection, to objects of the application that initiated the collection (henceforth called the *GC initiator*). The MVM separates the application-dependent part of the runtime representation of classes from the rest of the class representation. When a class is sharable across applications, a *task table* is interposed between the class representation and its application-dependent part, the latter being allocated in the old generation of the corresponding application. The task table for a class has an entry for every application executing in the MRE, and each application is assigned, upon startup, a

[3] E.g., using `map`/`unmap` system calls on the Solaris^{TM} OS.

unique number (*the task identifier*) which is used to index these tables. The entry of a task table holds a reference to the object that holds the application-dependent part of the class when the application associated with that entry loads the class, or a null pointer otherwise [4]. Classes whose representation cannot be shared across programs (e.g., classes defined by program-defined class loaders) refer directly to the application-dependent part. All data structures that directly reference application-dependent data are clustered in a specific area of the permanent generation, which is the only area that needs to be traced during collection of younger generations. When an application does not use program-defined class loaders, tracing is limited to a single entry in every task table (the entry assigned to the GC trigger).

Other data-structures that reference application-dependent data (e.g., JNI Handles) are organized either in a per-application pool or in tables with one entry per application, similar to the task table. MTM^2 is aware of this organization and exploits it to *scan only those pools or table entries associated with the GC initiator*. Further, only stacks of threads of the GC initiator are scanned for roots.

3 MTM^2's Mark-Evacuate-Sweep Garbage Collector

Our experiments with prototypes of MVM suggest that efficient GC is key to making the concurrent execution of multiple programs using multi-tasking a viable alternative to running the same programs using one instance of a single-task MRE per application.

MTM^2's old generation design is constrained by the need to ensure that an old region contains only objects from the same application, for performance isolation, as well as for efficient and accurate tracking of heap resources. This implies that dead space within an old region allocated to an application cannot be reused by another application. This can potentially lead to significant fragmentation and substantially increase footprint for multi-tasking. Copying GC is effective at mitigating fragmentation, but at the cost of excessive copying of live objects, and the necessity of a copy reserve area. In place compaction requires multiple passes over the heap (although recent work has significantly optimized compaction [20]). Mark-sweep, however, is fast, and involves a single pass over live data, but may result in poor space utilization [17].

MTM^2 combines two relatively simple and well-understood techniques: mark-sweep and copying. We use copying to evacuate live objects from only those regions that are fragmented or are sparsely populated, and mark-sweep for the remaining regions. The goal is to maintain a low footprint, but without the overhead of copying of all live objects and a copy reserve for every GC. Space reclaimed via sweeping can only be used by the GC initiator, since the free space may be co-located with live data in the same region. Evacuated regions, on the other hand, can be returned to the old region pool where the backing storage for their virtual pages is freed until the regions are re-assigned to an application.

Candidate regions for evacuation are selected based on the amount of fragmentation the GC *is likely to cause* in the region. Before the collection begins, MTM^2 suspends all the threads of the GC initiator at a GC *safepoint*. The threads are restarted when GC completes.

The collection itself is performed in four phases: marking, selecting candidate regions for evacuation, evacuation (copying), and sweeping and adjustment of regions

(performed in the same pass). The first two phases gather information (liveness, connectivity, occupancy, and estimated fragmentation) necessary for the last two phases. Evacuation and adjustment are optional, and occurs only if the second phase selects regions for evacuation.

Figure 2 illustrates with an example the main phases of MTM^2's hybrid collection. The following sub-sections detail each of the four phases.

3.1 Marking Phase

The marking phase begins a collection and produces two data structures as output: a *mark bitmap* that records live objects of the GC initiator; and a *connectivity matrix* that records connectivity information between old generation regions. Together, these are used to determine regions to evacuate, sweep and adjust.

Storage for the mark bitmap and the connectivity matrix is allocated for the duration of the hybrid GC cycle. The mark bitmap has one bit for every word of heap memory. Marking starts with the roots of live objects for the GC initiator: the stacks of the application's threads; the entry corresponding to the GC initiator in each task table for the runtime representation of shared classes in the permanent generation, and entries in global tables maintained by the multi-tasking MRE (such as JNI handles).

Marking then traverses the object graph from these roots. Because isolation is strictly enforced between applications through application-private regions, the marking phase will never access an object allocated by another application nor traverse a region allocated to another application.

The connectivity matrix is updated when a yet unmarked object is traversed. The matrix is encoded as a two-dimension boolean array, so that an entry (i, j) set to *true* indicates that there exists *at least* one reference from region i to region j. The matrix is initially zero-filled.

Testing whether each reference crosses a region boundary can be expensive. We have observed that inter region object references in the old generation are clustered, and that the distance between the referencing (source) object and the object being referenced (destination) in an old region is often small. Therefore, given an old region size that is large enough, the source and destination objects are likely to reside in the same region. If region size is a power of two, and regions are aligned, testing whether two addresses are in the same region can be inexpensively performed as follows [4]:

```
to == *from;
if (to ∧ from) >> LOG_REGION_SIZE) != 0 {
    // Not in the same region.
    update_connectivity_matrix(to,from);
}
```

When the test fails, a slow path is taken in order to update the connectivity matrix. The choice of an appropriate region size contributes to confine clusters of connected objects to a single region, which has two benefits: reducing the overhead of tracking inter-region connectivity (i.e., the fast path will be taken more often); and limiting

[4] This test for cross-region object references is similar to the test in the write barrier of the Beltway framework [2] except that, in our case, the test is performed at marking-time.

the number of regions that needs to be inspected for potential pointer adjustment after regions are evacuated. We have empirically identified an old region size of 256KB that works well.

In summary, tracking of connectivity information helps to reduce the amount of live data that needs to be scanned during pointer adjustment if any region is evacuated.

3.2 Selecting Regions for Evacuation

The decision to evacuate a region attempts to balance the cost of evacuation (copying and pointer adjustment) and heap fragmentation (consequently, footprint). To maintain a low cost for evacuation we evacuate sparsely populated regions, while to maintain a low footprint, we evacuate regions with fragmentation.

That is, our evacuation policy evacuates a region unconditionally if the ratio of the live to dead space (*live ratio*) is less than a certain minimum live ratio ($MinLiveRatio$). The region is also evacuated if it is estimated to be fragmented. This is done by comparing the average size of each contiguous fragment of dead space to a threshold ($MinFragmentSize$). That is, given L, the amount of live data in the region, F, the number of contiguous areas of dead objects in the region, and R the size of the region, a region is selected for evacuation if:

$$(L/R) < MinLiveRatio \vee ((L/R) < K \wedge (F > 1) \wedge ((R-L)/F) < MinFragmentSize)$$

We empirically chose $MinLiveRatio$ to be 0.25, i.e., a region is always evacuated if it contains over 75% of garbage. When the pool of old regions is closed to exhaustion, this parameter is increased up to 0.9 to aggressively evacuate all but the almost full regions. K is the occupancy threshold and chosen to be 0.9, i.e., we look for fragmentation in regions that are at least 90% full. $MinFragmentSize$ is set to 50 bytes by default.

In order to realize this policy, MTM^2 needs to determine the ratio of live to dead data and the number of contiguous fragments of dead space in each region. This is done following the completion of the marking phase, by scanning the mark bitmap. For each region belonging to the GC initiator, MTM^2 walks over the region's objects, using the mark bitmap to determine their liveness and the objects' header to obtain their size. In addition, in this pass, adjacent dead objects are coalesced into a single dead area to reduce scanning time for future passes.

3.3 Evacuation, Sweeping and Adjustment of Old Regions

Live objects in regions selected for evacuation are relocated to new regions allocated from the old regions pool. Evacuation traverses the region being evacuated for live objects using the mark bitmap. Live objects are copied to the new region, and a forwarding pointer is installed in the header of the (old) copied object. Forwarding pointers are necessary for pointer adjustment. This, however, prevents the evacuated regions from being freed before adjustment is complete.

New regions used to store evacuated objects are added to the set of regions that need adjustment, i.e., we assume that the a region is likely to contain objects that point to other objects in the same region.

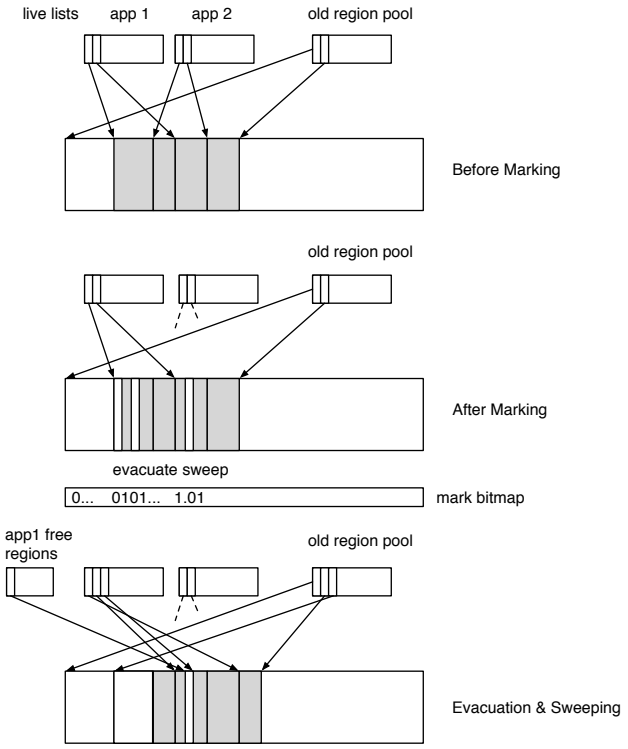

Fig. 2: Marking, Evacuation and Sweeping of Old Regions. Each application has a corresponding list of live areas. Marking traverses live objects for an application and marks live objects in the mark bitmap. After marking, candidate regions for evacuation (or sweeping) are selected based on the amount of live data and fragmentation. Regions selected for evacuation are then evacuated, regions selected for sweeping are swept and free areas in these added to a per-application free list. Pointer adjustment for swept regions is also performed during this pass, if necessary.

Once evacuation is complete, sweeping and adjustment of pointers can be performed in the same pass. For each region that was not evacuated, the mark bitmap corresponding to the region is used to build lists of live and free areas within the region. The connectivity matrix is also checked to determine if the region contains objects that reference objects in evacuated regions. If so, the live objects in the region are scanned to adjust references. Finally, the free and live lists of areas are combined into a list of live old generation areas used by the application. The live list is used to account for the old generation usage of the application, as well as during young generation collection to limit the amount of work that is done during card scanning, i.e., we only need to scan dirty cards that correspond to the application's list of old generation regions. The application's free list that was constructed during sweeping can only be used to satisfy allocation requests for that application (cf. Section 2).

If any region is evacuated, in addition to adjustment of some old regions, we also need to adjust objects in the young generation of the application, the permanent generation, and outside the heap (globals) that reference objects in the evacuated region(s).

The young generation is typically small (the default is 2MB) and can therefore be scanned in its entirety without significant overhead. However, performing an object graph traversal beginning from the roots to identify globals and permanent generation pointers that need to be adjusted can be prohibitively expensive. Instead, we keep track of the locations of these pointers during the marking phase, and update them during pointer adjustment. The space overhead required to keep track of these locations is small, and is reported as part of the total footprint of MTM^2 in Section 4.

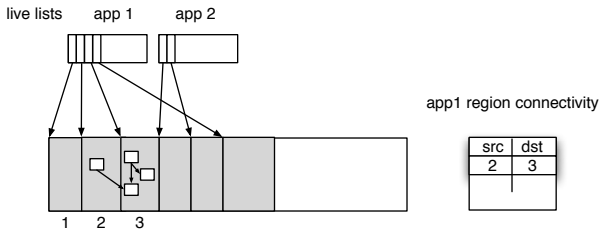

Fig. 3: Adjustment of old regions. Application 1 is being collected. We build the region connectivity matrix for application 1 during the marking phase. Region 2 has outgoing pointers to Region 3, therefore, Region 2 must be scanned if Region 3 is evacuated. However, Region 1 and 4 do not need to be scanned.

Once all regions have been adjusted, the evacuated regions are returned to the pool of free regions, and backing physical memory corresponding to their virtual address pages is unmapped, i.e., freed regions do not consume physical memory and can be later re-mapped and used as part of the old generations for *any* application.

Objects larger than a single region are treated specially. They are assigned an integral number of contiguous old regions large enough to hold the object. MTM^2 notes whether a region is part of a single large object region and whether that object contains no references (scalars only). This information is used to reclaim space when these large objects die (e.g., by returning their regions immediately to the pool, without waiting for adjustment).

4 Evaluation

The design of MTM^2 was motivated by the poor behavior of extant approaches to multi-tasking for concurrent application workloads, especially when compared to running the same concurrent workload using one instance of a single-tasking MRE per application as noted in Section 1.

In this section, we report our assessment of how well a multi-tasking MRE using MTM^2 fares when facing similar workloads. We first compare the performance of the most recent prototype of MVM (described in [26]) to that of an MVM modified

to integrate MTM^2. We then compare MTM^2 to a single-tasking MRE. We use the Java^{TM} HotSpot virtual machine, version 1.5.0-03 [14], a production quality, high-performance virtual machine from Sun Microsystems (which we will refer to as HSVM from now on). Both MVM and MTM^2 derive their implementation from HSVM and share a significant amount of code, which facilitates comparison. MVM and MTM^2 differ only in their memory management sub-systems and modification to the runtime to achieve GC performance isolation. All other mechanisms to support multi-tasking and sharing of the runtime representation of classes, byte code and compiled code (see [4, 5] for detailed descriptions) as well as other virtual machine implementation aspects inherited from HSVM are identical.

The main metrics of interest for our comparison are execution time and the aggregate memory footprint necessary to run concurrent workloads.

We begin with a description of our experimental setup, including hardware, benchmark, and methodology.

Benchmark	Description
compress	SpecJVM98 compression utility (input 100)
jess	SpecJVM98 expert system shell benchmark: Computes solutions to rule based puzzles (input 100)
db	SpecJVM98 database access program (input 100)
javac	SpecJVM98 Java to bytecode compiler (input 100)
mtrt	SpecJVM98 multi-threaded ray tracing implementation (input 100)
jack	SpecJVM98 Java parser generator based on the Purdue Compiler
antlr	Dacapo antlr: Parses grammar files and generates a parser
	and lexical analyzer for each (default input)
fop	Dacapo fop: XSL-FO to PDF parser/converter (default input)
luindex	Dacapo luindex: uses lucene to index the works of Shakespeare and the King James Bible (default input)
ps	Dacapo ps: Postscript interpreter (default input)
opengrok	Open source code browser and cross reference tool (input: Source files in HSVM "memory" subdirectory, 118 files)
jruby	Ruby interpreter written in Java (uses small scripts as input: beer song, fibonacci numbers, number parsing, thread test)
groovy	Groovy interpreter written in Java (input: unsigns, i.e., strips MANIFESTs) for a number of jar files from Apache ant)
antlr-mixed	mixed workload consisting of antlr, fop and opengrok
luindex-mixed	mixed workload consisting of luindex, fop and ps
javac-mixed	mixed workload consisting of javac, jess, mtrt and jack
scripts-mixed	mixed workload consisting of groovy and jruby

Fig. 4: Benchmarks and workloads used in the empirical evaluation of MTM^2

4.1 Experimental Methodology

We ran our experiments on a dedicated dual CPU 1.5GHz UltraSPARC IIIi system, with 2GB physical memory, 32KB instruction and 64KB data cache running the Solaris^{TM} Operating System version 10.

Figure 4 describes the benchmarks and workloads we used for our experiments.

Programs used in our concurrent workloads are selected from community programs from the SpecJVM98 [27] and Dacapo [7] benchmark suites [5], as well as two commonly

[5] We used version 2006-10-MR2 version of the Dacapo benchmarks, and `ps` from Dacapo version beta-05022004.

used open-source scripting applications, `jruby` [18] and `groovy` [10], and an open-source source code browser and cross reference tool called `opengrok` [23]. We exclude *mpegaudio* from SpecJVM98 (as is commonly done) since it does not exercise the GC.

We experimented with two types of workloads: *homogeneous* and *heterogeneous*. A homogeneous workload consists of multiple concurrent instances of the same program. For instance, 10 instances of `javac` implies that 10 instances of this program are launched simultaneously. A heterogeneous workload consists of concurrent instances of different programs.

	Number of instances					
	2		5		10	
Bmark	Exec time (sec)	Footprint (MB)	Exec time (sec)	Footprint (MB)	Exec time (sec)	Footprint (MB)
compress	10.96	139.44	27.60	351.16	56.41	650.80
jess	4.93	19.82	12.33	33.18	24.54	55.12
db	20.84	35.95	52.50	74.05	105.10	141.00
javac	10.40	49.51	26.78	109.85	53.97	261.03
mtrt	3.39	20.46	8.47	62.27	16.24	114.26
jack	4.21	30.84	10.75	59.53	20.89	104.78
antlr	9.17	67.51	20.95	114.23	39.86	219.61
fop	6.00	51.84	14.31	87.45	28.53	148.49
luindex	40.08	76.68	89.45	173.35	169.42	333.43
ps	27.18	16.63	68.37	23.91	136.80	37.02
opengrok	10.44	101.50	25.40	230.85	51.35	429.77
groovy	10.15	138.06	21.54	366.63	50.92	544.25
jruby	2.58	34.80	5.66	49.67	10.67	73.47
Average	12.33	60.23	29.55	133.55	58.82	239.46

	Number of instances			
	1		2	
Bmark	Exec time (sec)	Footprint (MB)	Exec time (sec)	Footprint (MB)
antlr-mixed	12.64	79.52	24.43	148.06
luindex-mixed	34.44	77.04	42.47	132.76
javac-mixed	13.28	31.97	23.58	63.57
scripts-mixed	8.28	68.68	11.30	118.95
Average	17.16	64.30	25.45	115.84

Fig. 5: Total execution time (in seconds) and footprint (in MB) data with MTM^2 for concurrent homogeneous (multiple instances of same application), and heterogeneous (multiple instances of different applications). The benchmarks are described in Figure 4. All relative performance improvement results for execution time as well footprint in this paper use these values.

We refer to the heterogeneous workload as *mixed* in Figure 4. We present results for 1 and 2 instances each of an application in a heterogeneous workload. For example, 2 instances each for `antlr-mixed` implies that we launch 2 instances each of `antlr`, `fop` and `opengrok` simultaneously, i.e., 6 concurrent instances.

We report total execution time by reporting the time elapsed since the applications in a workload were launched and until the last application completes. We use a harness that executes each application as an *isolate* [15] using reflection and we report total elapsed time using `System.nanoTime()`. To measure footprint, we use the UNIX `pmap` utility, which we execute as an external process in a tight loop and report the maximum of the total RSS (resident segment size) value reported by executing `pmap -x` on the MRE process. Footprint and execution times are reported using independent runs. In case of single-tasking, we sum the RSS values returned by `pmap`

for each individual MRE process (since to execute concurrent workloads, we must launch a single-tasking MRE process for each application).

We perform all HSVM experiments using the *client* compiler and the default serial GC (sliding mark-compact) used for client configuration (i.e., using the `-client -XX:+UseSerialGC` command line flags). HSVM and MTM^2 both use copying GC for collecting the young generation. For all results, we present the average of 5 executions.

4.2 MTM^2 Versus MVM

We first present performance results that compare MTM^2 to MVM. MVM provides performance isolation for the young generation only, per-application resource accounting, and immediate, GC-less reclamation of heap space upon program termination. However, as seen earlier, this MVM performs poorly for concurrent workloads relative to executing the same concurrent workload with multiple instances of HSVM (cf Figure 1).

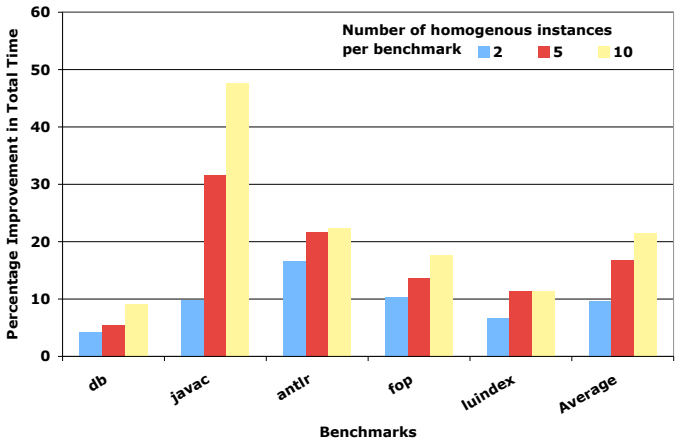

Fig. 6: Percentage improvement in execution time enabled by MTM^2 versus a state-of-the-art implementation of MVM for concurrent workloads that show significant old generation GC activity. MTM^2 enables better performance due to a more efficient old generation GC and performance isolation.

Figure 6 shows the performance improvement enabled by MTM^2 over this MVM. The results indicate that MTM^2 outperforms MVM by 10%, 15%, and 22% on average when running 2, 5, and 10 concurrent instances, respectively. For this experiment, we only present results for applications that show significant old generation GC activity. This performance improvement is possible due to the hybrid old generation GC in MTM^2 that enables performance isolation, as well as improved GC performance.

Figure 7 shows the old generation GC times for MTM^2 versus MVM. MTM^2's hybrid GC significantly improves GC performance over MVM. MVM uses a stop-the-world mark-compact GC for the old generation that performs three passes over the entire old generation (for all applications), with cost proportional to the size of the heap. With more concurrent instances, the cost of old generation GC in MVM increases.

	Number of instances								
	2			5			10		
Bmark	MVM (sec)	MTM^2 (sec)	% imp	MVM (sec)	MTM^2 (sec)	% imp	MVM (sec)	MTM^2 (sec)	% imp
db	0.57	0.28	**51.95**	2.92	0.70	**76.05**	5.47	1.38	**74.81**
javac	3.24	2.51	**22.47**	8.95	3.48	**61.06**	40.10	7.93	**80.23**
antlr	2.44	0.48	**80.17**	4.11	1.29	**68.69**	6.23	1.39	**77.75**
fop	1.18	0.67	**42.96**	2.29	1.11	**51.58**	4.98	2.54	**49.00**
luindex	4.27	1.51	**64.73**	8.36	2.86	**65.82**	14.35	8.24	**42.60**
Average	2.34	1.09	**52.46**	5.32	1.89	**64.64**	14.22	4.29	**64.88**

Fig. 7: Old generation GC times (total) for MTM^2 versus a prior state-of-the-art implementation of multi-tasking (MVM). GC times are presented in seconds along with percentage improvement in GC time enabled by MTM^2 vs MVM. MTM^2 's per-application hybrid old generation GC outperforms MVM's mark-compact old generation GC.

In addition, unlike MVM, MTM^2 never pauses tasks to perform GC and all allocation and collection for any application is isolated with respect to other applications. MTM^2 scales better over MVM overall due to performance isolation as the number of instances is increased, as seen in Figure 6. The impact of performance isolation is especially evident in the case of `javac`. For instance, when 10 concurrent instances of `javac` execute, the total old generation GC time for MVM is about 40 seconds. The cost of old generation GC is higher since mark-compact GC needs to scan a larger heap in case of MVM. Further, since *all* applications are paused during old generation GC, MVM significantly degrades execution time for *javac*. In the case of *db* and *luindex*, GC time does not dominate total execution time, and consequently, the improvement enabled by MTM^2 versus MVM is less significant.

4.3 MTM^2 Versus HSVM

We next compare the execution time and footprint of MTM^2 to HSVM. HSVM allows users to specify an initial heap size (32MB by default) and a maximum heap size (64MB by default) when launching an application. The initial heap size controls the heap limit, the point at which a full GC is triggered. The initial heap size grows (or shrinks) after a full GC, if required. For results in Figures 8, 9, 10, and 11, we set the initial heap size for HSVM equal to the maximum heap size. With this setting, HSVM performs less frequent GC and achieves better overall performance (total execution time), compared to when the initial heap size is at the default value. This setting allows single-tasking to perform at its best potential since the application heap is not restricted. We also present results for the other case, i.e., when the initial heap size for HSVM is not set to the maximum initially (the default behavior), thereby allowing HSVM to achieve a smaller footprint (Figures 14, 15, 12 and 13). MTM^2 does not restrict the initial heap size, or use a "soft limit" for applications, yet *we always ensure that we never exceed the maximum heap size setting for an application* (which is set to the HSVM default maximum heap size of 64MB in order to ensure a fair comparison).

Figure 8 shows percentage improvement in total execution time when homogeneous workloads are executed with MTM^2 versus the HSVM virtual machine, i.e., concurrent instances of the same application. We present results for 2, 5 and 10 concurrent instances for each application. Multi-tasking allows sharing of compiled code and classes between applications resulting in reduced overall execution time. MTM^2

enables an improvement of 11%, 13% and 14% for 2, 5 and 10 concurrent applications on average for homogeneous workloads. MTM^2 allows complete application isolation and space reclaimed by evacuating old generation regions for an application to be reused by other applications. Scripting and parsing applications such as `antlr` and `jruby` are commonly used on desktop systems and particularly show a significant benefit due to sharing of compiled code.

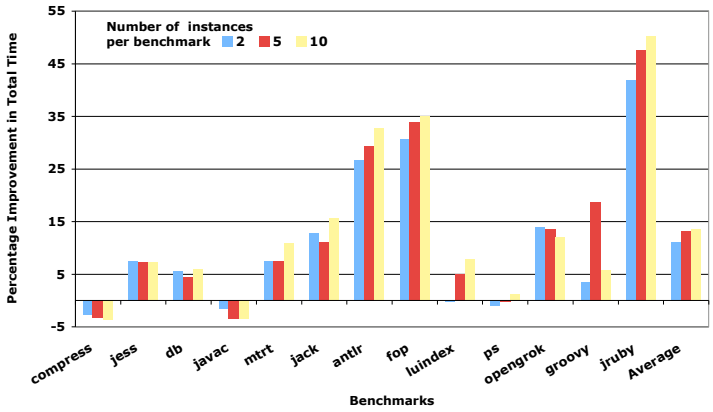

Fig. 8: Percentage improvement in execution time enabled by MTM^2 over HSVM (default initial heap size = max heap size = 64MB) for homogeneous concurrent workloads (multiple instances of the same application). Benchmarks are described in Figure 4.

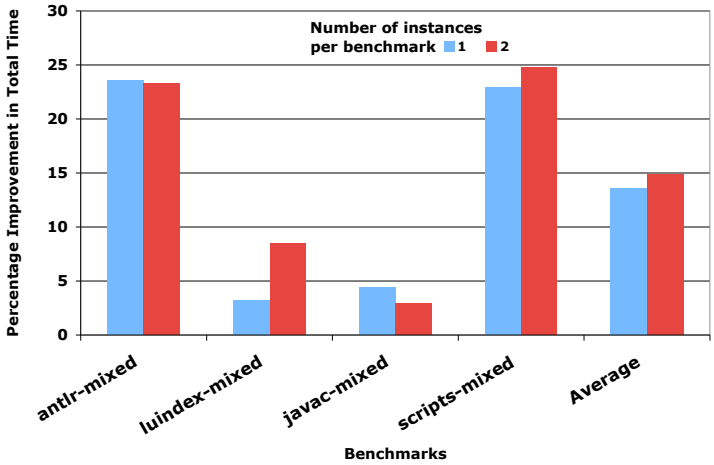

Fig. 9: Percentage improvement in execution time enabled by MTM^2 versus HSVM (default initial heap size = max heap size = 64MB) for heterogeneous concurrent workloads (multiple instances of different applications). Benchmarks are described in Figure 4.

For some applications, such as `compress`, `javac` and `ps` multi-tasking does not outperform single-tasking. For `compress` in particular, multi-tasking performance lags

single tasking due to the fact that it allocate large objects (byte arrays) in the old generation which leads to fragmentation and worse GC performance in a shared old generation address space, and also due to the overhead due to a level of indirection to access static variables [4]. However, MTM^2 attempts to mitigate the adverse impact of fragmentation and achieves a significant benefit for these worst-case applications over the state-of-the-art multi-tasking MRE implementation, as shown earlier (cf Figure 6), while achieving performance that is close to the performance of these applications with single-tasking (within 3%). On average, MTM^2 significantly outperforms single-tasking.

Figure 9 shows the percentage improvement in total execution time for heterogeneous workloads, i.e., concurrent instances of different applications for 1 instances of each application, and 2 instances of each application for every heterogeneous workload (see Figure 4). For example, *antlr-mixed* with two instances indicates that 2 instances each of antlr, fop, opengrok are executed concurrently (6 concurrent applications). On average, MTM^2 improves performance by up to 16%, with improvements ranging from 3% to 25% in individual cases. As seen earlier, scripting workloads in particular perform very well with multi-tasking.

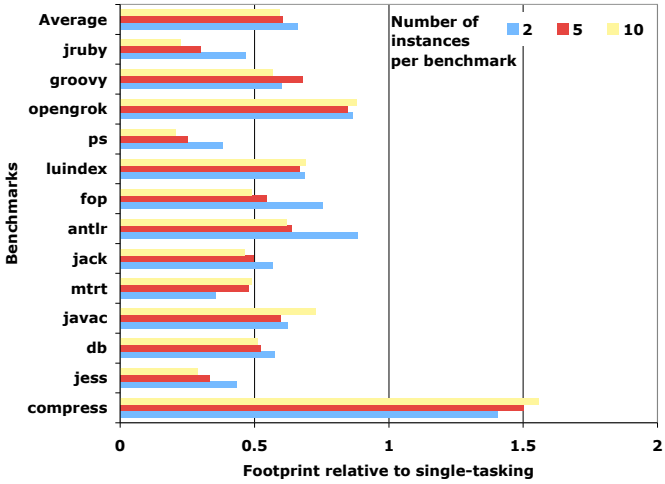

Fig. 10: Percentage improvement in footprint enabled by MTM^2 versus HSVM (default initial heap size = max heap size = 64MB) for homogeneous concurrent workloads (2, 5, and 10 instances of the same application).

Figures 10 and 11 compare the total process footprint for MTM^2 versus HSVM for the same set of applications as in the previous figures. Each bar represents the ratio of the footprint for MTM^2 versus HSVM. The value 1 indicates that MTM^2 and HSVM have identical footprint for a given workload. Values less than 1 indicate that MTM^2 has a lower footprint.

MTM^2 shows a better footprint compared to HSVM and on average, MTM^2 achieves 34% to 41% reduction in footprint for homogeneous workloads, and 31% to

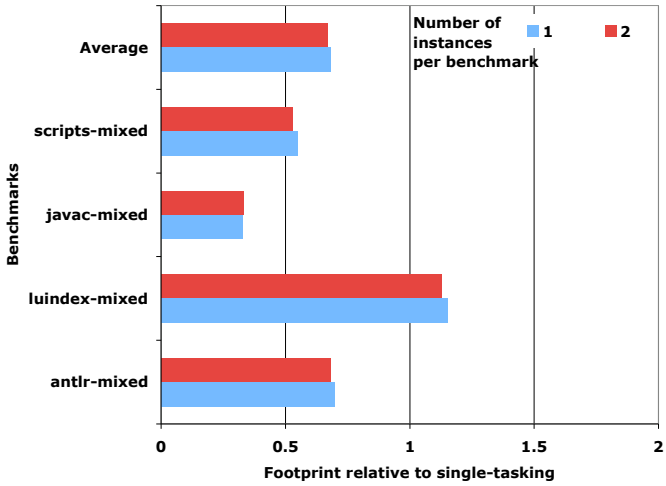

Fig. 11: Percentage improvement in footprint enabled by MTM^2 versus HSVM (default initial heap size = max heap size = 64MB) for heterogeneous concurrent workloads. 1 denotes 1 instance each of the mix of applications that constitute a heterogeneous workload. 2 indicates 2 instances of each application in the mix.

33% benefit for heterogeneous workloads. These savings are possible due to sharing of classes and compiled code in a multi-tasking MRE.

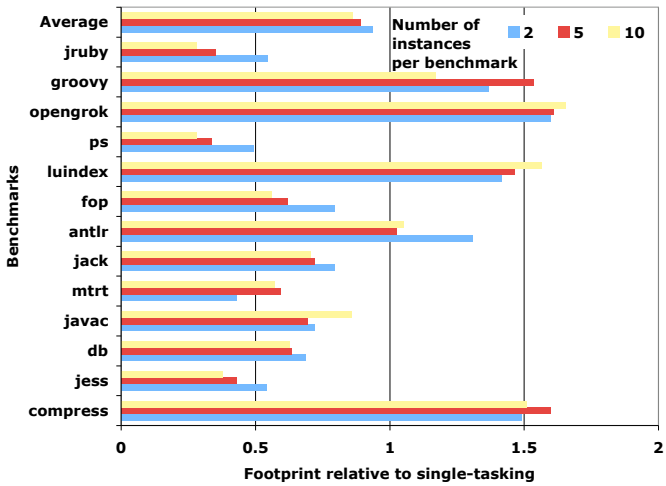

Fig. 12: Percentage improvement in footprint enabled by MTM^2 versus HSVM (default initial heap size = 32MB) for homogeneous concurrent workloads (2, 5, and 10 instances of the same application).

However, `compress` shows worse footprint (around 50% or 1.5x). The worse footprint for `compress` can be attributed to large scalar objects (objects that do not hold references, such as byte arrays). As noted earlier, `compress` allocates a significant

number of large byte arrays which are directly allocated in the old generation. Since our old generation is non-contiguous, and since we allocate large scalar objects in a separate region, which we can safely skip during pointer adjustment, allocation of very large (> minimum region size, which is 256KB by default), byte arrays leads to excess fragmentation. A new region needs to be allocated for each such large byte array, and this region needs to be aligned to the region boundary for correctness. However, the number and size of these is unknown at runtime, without a priori profiling. Therefore, we cannot pre-allocate a suitable sized region. As part of future work, we plan to address the allocation of large objects, by providing a per-application large object region that is sized differently and collected separately from the old generation. Note that `compress` is a numerical computation benchmark and does not represent typical MRE workloads.

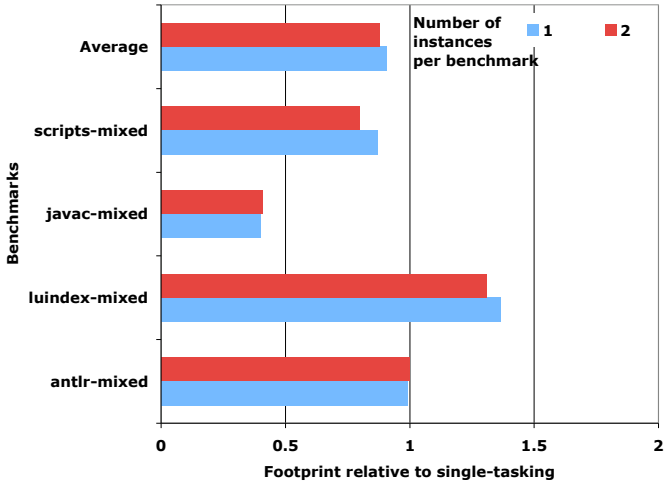

Fig. 13: Percentage improvement in footprint enabled by MTM^2 versus HSVM (default initial heap size = 32MB), heterogeneous workloads, i.e., multiple concurrent instances of different applications. 1 denotes 1 instance each of the mix of applications that constitute a heterogeneous workload. 2 indicates 2 instances of each application in the mix.

Figures 12 and 13 compare the process footprint for MTM^2 versus HSVM, when the initial heap size for HSVM is restricted and increased gradually. In this configuration, HSVM gradually increases the heap (if required), starting from an initial default (32MB), in order to achieve smaller footprint. As expected, HSVM runs in a much smaller heap and consequently, the process footprint is lower. On average, MTM^2 shows a footprint improvement of 6% to 14% for homogeneous workloads, and 12% to around 15% for heterogeneous workloads. Note that these values are smaller compared to the earlier configuration of HSVM, i.e. when the initial heap size for HSVM is not restricted. However, if we look at the execution time for MTM^2 versus HSVM (Figures 14 and 15) when the initial heap size for HSVM is restricted, MTM^2 outperforms HSVM by a *greater margin* than when we do not restrict the

initial heap size for HSVM. On average, MTM^2 shows an improvement of 15% to over 17% for homogeneous workloads, and 19% to 21% for heterogeneous workloads.

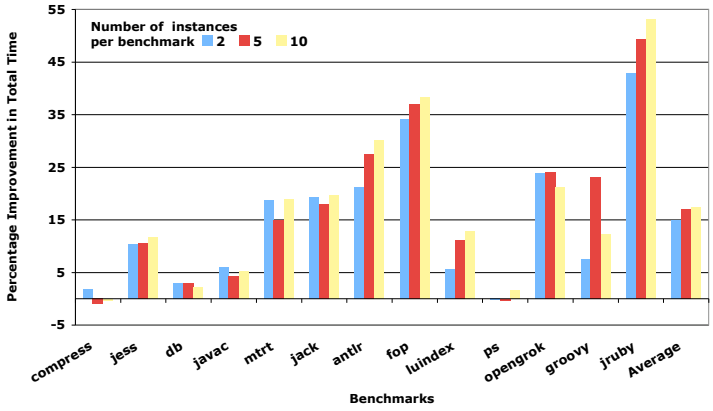

Fig. 14: Percentage improvement in execution time enabled by MTM^2 versus HSVM (default initial heap size = 32MB) homogeneous concurrent workloads. Benchmarks are described in Figure 4.

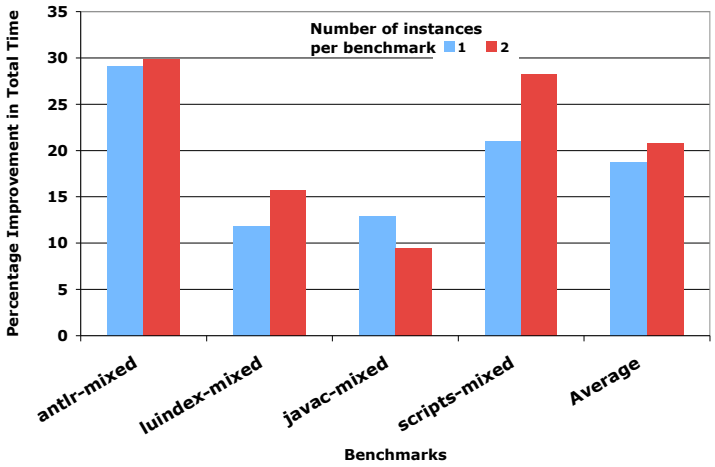

Fig. 15: Percentage improvement in execution time enabled by MTM^2 versus HSVM (default initial heap size = 32MB) for heterogeneous concurrent workloads (multiple instances of different applications). Benchmarks are described in Figure 4.

In summary, by controlling heap growth the single-tasking HSVM virtual machine can achieve a better footprint when the heap is not restricted, however, MTM^2 shows a comparable or better footprint on average across concurrent workloads that we looked at. Further, MTM^2 outperforms HSVM by a larger margin, since there is a reduction in performance for the single-tasking MRE due to more frequent GC. There

exists a tradeoff between execution time and footprint by choosing the threshold at which GC is triggered. We believe that manually having to select an appropriate per-application heap size in a context of a multi-tasking VM is counter-productive. On average, MTM^2 significantly outperforms HSVM and has a better footprint without having to manually select an appropriate initial per-application heap size.

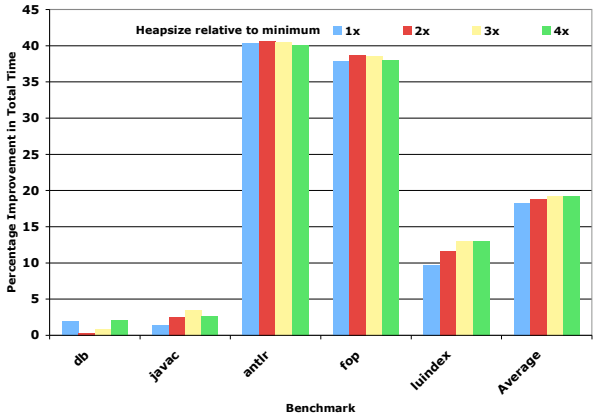

Fig. 16: Percentage improvement in execution time enabled by MTM^2 over HSVM for 1 through 4 times the minimum heap size that each benchmark needs to execute in MTM^2 .

We next examine the performance of MTM^2 versus HSVM as the heapsize is varied from the minimum that an application requires to execute in MTM^2 , to 4 times the minimum for that application (Figure 16). We only consider benchmarks that show significant old generation GC activity. The minimum heap size selected is 45MB for `luindex` and 22MB for the rest. Across heap sizes, MTM^2 outperforms HSVM by 18 – 19% on average.

However, HSVM is able to execute programs in a smaller heap compared to MTM^2 (i.e., $<$ 45MB for `luindex` and $<$ 22MB for other benchmarks). HSVM uses in-place sliding compacting GC, which is more space efficient than MTM^2's hybrid GC for small heaps. This is due to the fact that evacuation, although it is partial and selective, requires a copy reserve for the duration of the GC to copy live objects. For highly memory constrained scenarios, HSVM's GC may be a more suitable choice compared to evacuation. We are investigating mechanisms to perform in-place compaction across disjoint regions as part of future work.

4.4 Sensitivity Analysis

In the next set of results, we examine how MTM^2 with selective evacuation (copying) and mark-sweep compares to only mark-sweep and only copying. Our hybrid GC can operate as a mark-sweep only GC (by setting the *MinLiveRatio* threshold described in Section 3 to `0`), or as a copying only GC (by setting the *MinLiveRatio* threshold to `1`, i.e., 100%).

In particular, in Figure 17 we present total process footprint for MTM^2 with hybrid GC versus MTM^2 with mark-sweep only, and MTM^2 with copying only,

Bmark	Number of instances: 2 — MTM² (KB)	MTM² MS (KB)	MTM² CP (KB)	% imp vs MS	% imp vs CP	5 — MTM² (KB)	MTM² MS (KB)	MTM² CP (KB)	% imp vs MS	% imp vs CP	10 — MTM² (KB)	MTM² MS (KB)	MTM² CP (KB)	% imp vs MS	% imp vs CP
javac	49.5	127.7	65.4	**61.2**	**24.3**	109.9	297.5	117.7	**63.1**	**6.6**	261.0	602.1	261.7	**56.6**	**0.3**
luindex	76.7	128.4	83.5	**40.3**	**8.2**	173.4	302.9	182.0	**42.8**	**4.7**	333.4	589.5	351.9	**43.4**	**5.2**

Bmark	Number of instances: 1 — MTM² (KB)	MTM² MS (KB)	MTM² CP (KB)	% imp vs MS	% imp vs CP	2 — MTM² (KB)	MTM² MS (KB)	MTM² CP (KB)	% imp vs MS	% imp vs CP
antlr-mixed	79.5	86.1	80.6	**7.6**	**1.3**	148.1	156.6	148.2	**5.5**	**0.1**
javac-mixed	32.0	51.9	41.6	**38.4**	**23.2**	63.6	91.3	87.5	**30.4**	**27.4**
scripts-mixed	68.7	94.8	104.5	**27.5**	**34.3**	119.0	127.0	140.3	**6.4**	**15.2**

Fig. 17: Footprint for MTM^2 with hybrid GC (mix of mark-sweep and copying) versus mark-sweep (MS) only and copying GC (CP) only for a set of homogeneous (instances of the same application) and heterogeneous (different applications) concurrent workloads. Hybrid GC achieves a footprint that is lower than always choosing mark-sweep or always choosing copying.

for a subset of benchmark programs. We only present results for benchmarks that show significant change in footprint compared to either mark-sweep or copying (> 5%). For all other benchmarks, we did not find a significant change in the footprint (however, MTM^2 with hybrid GC never shows a worse footprint compared to either mark-sweep or copying).

For `javac`, `luindex`, `javac-mixed` and `scripts-mixed`, hybrid GC has a much smaller footprint compared to mark-sweep. We believe this is due to fragmentation due to using mark-sweep only without any compaction. For `javac-mixed` and `scripts-mixed`, copying has a higher footprint, since always copying all live data requires a larger copy reserve space during GC. While performing evacuation, the old as well as the new (copied to) regions need to be occupied (mapped) for the duration of the GC cycle.

We next examine the effect of using hybrid GC, mark-sweep only, and copying only, on execution time for `javac`, which shows a significant difference in performance (Figure 18). Using mark-sweep only results in excess fragmentation. Fragmentation has an interesting effect on execution time for `javac` – an increase in young generation GC time by 8% on average (or 0.51 sec, 0.66 sec and 1.17 sec for 2, 5 and 10 instances respectively) due to an increase in card scanning time, since more cards need to be scanned. Using copying alone results in excess copying and adjustment, and consequently, performance suffers due to an increase in old generation GC time by around 6% (or 0.07 sec, 0.16 sec and 0.70 sec for 2, 5 and 10 instances respectively).

For other benchmarks, we did not encounter a significant change in execution time (however, in all cases, hybrid GC never performs worse than using mark-sweep or copying alone).

To summarize, hybrid GC achieves a lower footprint in many cases for benchmarks that show significant old generation GC activity, while maintaining performance that is on-par or better than using mark-sweep or copying alone.

	Number of instances														
	2					5					10				
				% imp vs					% imp vs					% imp vs	
Bmark	MTM^2 (sec)	MTM^2 MS (sec)	MTM^2 CP (sec)	MS	CP	MTM^2 (sec)	MTM^2 MS (sec)	MTM^2 CP (sec)	MS	CP	MTM^2 (sec)	MTM^2 MS (sec)	MTM^2 CP (sec)	MS	CP
javac	10.40	10.82	10.57	**3.9**	**1.6**	26.78	28.14	27.29	**4.8**	**1.9**	53.97	56.48	55.25	**4.4**	**2.3**

Fig. 18: Execution time MTM^2 with hybrid GC (mix of mark-sweep and copying) versus mark-sweep (MS) only and copying GC (CP) only for the `javac` benchmark.

5 Related Work

Our work relates directly to other multi-tasking implementations of MREs. To our knowledge, no prior work conclusively demonstrates that multi-tasking has the ability to outperform a single-tasking MRE in terms of execution time, as well as overall footprint for concurrent workloads (multiple applications executing simultaneously). MVM is the most well-known, state-of-the-art implementation of a multi-tasking MRE. Prior work on MVM reports substantial improvement for startup, footprint and execution time compare to a corresponding single-tasking JVM [4, 5]. However, execution times were measured for serial execution of programs, and footprint of concurrently running programs were obtained when applications were quiescent, and do not reflect the footprint of programs when they are actually running concurrently and are exercising the memory management system. Recent attempts at improving GC performance isolation for MVM [26] only address young generation GC and GC-less instantaneous reclamation of the heap space of terminated programs, and demonstrates only provision of performance isolation for short-lived programs that do not stress the GC. Sun Microsystems' CLDC HotSpot Implementation, aimed at small hand-held devices, supports multi-tasking in a way that is similar to MVM, but uses a single heap shared by all tasks [28], with no provision for GC performance isolation. We were not able to find any information about GC performance isolation for .NET application domains.

Our work also builds upon and extends a large body of important contributions to memory management for single-tasking MREs.

Our hybrid GC bears some similitude to incremental copying GCs that divide the heap into equally sized regions that can be evacuated independently of others. In our case, heap space partitioning is primarily motivated by the need to allocate private tenured space to isolated applications on demand. Like our hybrid GC, Garbage First [8] only evacuates regions that can be reclaimed with little copying. Information regarding the amount of live data in regions is provided by a concurrent marker (as opposed to a synchronous marking phase in our case). Bidirectional remembered sets between regions are maintained by mutators (with help from the concurrent marker) to allow any set of regions to be collected independently of the others. In the case of our hybrid GC, this property is achieved by gathering cross-regions connectivity information during marking. The Mature Object Space (MOS) collector of Hudson and Moss [13] is another region-based incremental copying GC. It uses unidirectional remembered sets, which requires regions to be evacuated in order. MOS cannot therefore pick an arbitrary region to evacuate based on cost-related criteria (e.g., amount of live data). Both Garbage First and MOS are evacuation-only GC.

Lang and Dupont [21] describes a hybrid mark-sweep and copy similar to ours. The heap is divided into equal size segments. During GC, a single segment is compacted, while others are swept. Like our hybrid mark-sweep-evacuate GC, the collector is primarily mark-sweep. The cost of compaction is bounded since a single segment is collected. However, the segment compacted at each GC is chosen arbitrarily. By contrast, we use copying opportunistically, only to evacuate sparsely populated regions or highly fragmented one. We may thus evacuate several regions during a single GC, or none if the regions are densely populated with little fragmentation.

MC^2 [25] and its predecessor, Mark-Copy [24] describe an incremental copying GC that uses a marking phase to precisely annotate equal size regions of the old generation of the heap with the amount of live data in them, like our GC, and then build uni-directional remembered set to update pointers to evacuated objects. MC^2 builds precise remembered sets, whereas we build an imprecise connectivity matrix that only records regions that references other regions. MC^2 aims at achieving good throughput and low pause times for memory constrained devices.

Beltway [2] provides incremental and generational GC by partitioning the heap into *belts* and collecting a single belt during GC. Garbage cycles larger than a belt cannot be reclaimed by collecting a single belt. However, Beltway has a provision for performing full GC by providing a separate belt with a single region and collecting this when it occupies half the heap space. Our per-application GC is complete and reclaims all garbage for that application. We, therefore, do not require precise remembered sets between regions or need mechanisms to ensure completeness.

McGachey et al [22] present a scheme that uses a generational GC with a reduced copy reserve, with the ability to dynamically switch to a compacting GC if necessary.

Page unmapping as well as compaction has been used to reduce application memory footprint in prior work, such as the Compressor [20]. However, Compressor is a concurrent, parallel compacting GC that achieves low pause times. Our goal is different: to provide a relative simple, per-application GC that achieves good footprint and overall performance for desktop or small client applications, while allowing other applications to execute concurrently, without interference.

6 Conclusion

Multi-tasking has been proposed as a means to enable sharing of code and classes between applications in order to enable better startup performance, footprint and for faster overall execution compared to single-tasking, i.e., executing each application in a separate MRE process. While prior implementations of multi-tasking have demonstrated the above for serial execution of programs, and for execution of multiple programs with little simultaneous activity, we show that the prior state-of-the-art performs poorly for concurrent workloads. We attribute this to lack of performance isolation and a poor performing garbage collector for full garbage collection (GC).

We have described MTM^2, a scalable approach to memory management for multi-tasking managed runtime environments. MTM^2 enables complete performance isolation with respect to GC, provides each application with a private heap, and employs generational GC with a hybrid GC for old generation collection. MTM^2's hybrid GC combines mark-sweep with copying collection in the same GC cycle along with fast

adjustment for copied objects, to achieve good performance and a low footprint while avoiding the overhead of full copying GC. The hybrid GC uses marking to gather information (liveness, connectivity, occupancy, and estimated fragmentation) necessary to determine regions of the old generation to evacuate (if any) and to sweep and to identify which regions need to be scanned for pointer adjustment.

We have integrated MTM^2 with MVM, a multi-tasking implementation of the JVM, and a compare it to a widely used, production-quality, single-tasking MRE for concurrent application workloads. Our results show that MTM^2 enables significant performance, as well as footprint improvement compared to single-tasking for concurrent workloads. MTM^2 outperforms single-tasking by up to 14% on average for homogeneous workloads (instances of the same application) and up to 16% on average for heterogeneous workloads (mix of different applications). MTM^2 achieves up to 41% reduction in footprint on average for homogeneous workloads, and up to 33% on average improvement for heterogeneous workloads over single-tasking. In addition, MTM^2 achieves better performance for concurrent workloads over the extant state-of-the-art multi-tasking implementation, outperforming it by 10% to 22%. These results indicate that multi-tasking is a viable approach for executing concurrent applications and strengthens the case for multi-tasking MREs.

Trademarks

References

1. S. Blackburn and K. McKinley. In or Out? Putting Write Barriers in Their Place. In *International Symposium on Memory Management (ISMM)*, 2002.
2. S. M. Blackburn, R. Jones, K. S. McKinley, and J. E. B. Moss. Beltway: Getting around garbage collection gridlock. In *Conference on Programming Language Design and Implementation*, June 2002.
3. C. Chambers. *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for an Objected-Oriented Programming Language.* PhD thesis, Stanford University, Mar. 1992.
4. G. Czajkowski and L. Daynès. Multitasking without Compromise: A Virtual Machine Evolution. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Oct. 2001.
5. G. Czajkowski and L. Daynès. A Multi-User Virtual Machine. In *USENIX 2003 Annual Technical Conference*, June 2003.
6. G. Czajkowski, L. Daynès, and N. Nystrom. Code Sharing among Virtual Machines. In *European Conference on Object-Oriented Programming (ECOOP)*, June 2002.
7. The Dacapo Benchmark Suite, version beta050224. `http://www-ali.cs.umass.edu/DaCapo/gcbm.html`.
8. D. Detlefs, C. Flood, S. Heller, and T. Printezis. Garbage-First Garbage Collection. In *International Symposium on Memory Management (ISMM)*, Oct. 2004.

9. A. Garthwaite, D. Dice, and D. White. Supporting per-processor local-allocation buffers using lightweight user-level preemption notification. In *First International Conference on Virtual Execution Environments*, June 2005.
10. Groovy: An agile dynamic language for the Java Platform. `http://groovy.codehaus.org/`.
11. U. Hölzle. A Fast Write Barrier for Generational Garbage Collectors. In *OOPSLA/ECOOP Workshop on Garbage Collection in Object-Oriented Systems*, Oct. 1993.
12. A. L. Hosking, J. E. B. Moss, and D. Stefanović. A Comparative Performance Evaluation of Write Barrier Implementations. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Oct. 1992.
13. R. L. Hudson and J. E. B. Moss. Incremental Garbage Collection for Mature Objects. In *International Workshop on Memory Management (IWMM)*, 1992.
14. S. M. Inc. The Java Hotspot Virtual Machine White Paper. `http://java.sun.com/products/hotspot/docs/whitepaper/Java_HotSpot_WP_Final_4_30_01.html`.
15. Java Community Process. JSR-121: Application Isolation API Specification. `http://jcp.org/jsr/detail/121.jsp`.
16. R. Jones and A. C. King. A Fast Analysis for Thread-Local Garbage Collection with Dynamic Class Loading. In *Fifth International Workshop on Source Code Analysis and Manipulation (SCAM'05)*, 2005.
17. R. E. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley and Sons, July 1996.
18. JRuby: Java powered Ruby implementation. `http://jruby.codehaus.org/`.
19. A. Kennedy and D. Syme. Combining generics, pre-compilation and sharing between software-based processes. In *Proceedings of the Second Workshop on Semantics, Program Analysis and Computing Environments for Memory Management (SPACE'01)*.
20. H. Kermany and E. Petrank. The Compressor: concurrent, incremental, and parallel compaction. *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*.
21. B. Lang and F. Dupont. Incremental Incrementally Compacting Garbage Collection. In *Symposium on Interpreters and Interpretive Techniques*, 1987.
22. P. McGachey and A. L. Hosking. Reducing Generational Copy Reserve Overhead with Fallback Compaction. In *International Symposium on Memory Management (ISMM)*, June 2006.
23. OpenSolaris Project: OpenGrok. `http://opensolaris.org/os/project/opengrok/`.
24. N. Sachindran, J. Eliot, and B. Moss. Mark-copy: Fast Copying GC with less Space Overhead. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Oct. 2003.
25. N. Sachindran, J. Eliot, and B. Moss. MC^2: high-performance garbage collection for memory-constrained environments. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Oct. 2004.
26. S. Soman, L. Daynès, and C. Krintz. Task-Aware Garbage Collection in a Multi-Tasking Virtual Machine. In *International Symposium on Memory Management (ISMM)*, June 2006.
27. SpecJVM'98 Benchmarks. `http://www.spec.org/osg/jvm98`.
28. Sun Microsystems Inc. CLDC HotSpot^{TM} Implementation. `http://java.sun.com/javame/reference/docs/cldc-hi-2.0-web/`.
29. D. Ungar. Generation Scavenging: A Non-disruptive High Performance Storage Recalamation Algorithm. In *Software Engineering Symposium on Practical Software Development Environments*, Apr 1992.