Memoised Garbage Collection for Software Model Checking

Viet Yen Nguyen[1] and Theo C. Ruys[2]

[1] RWTH Aachen University, Germany.
http://moves.rwth-aachen.de/~nguyen/

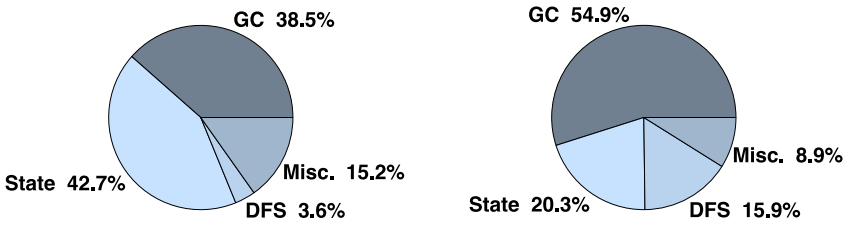
[2] University of Twente, The Netherlands.
http://www.cs.utwente.nl/~ruys/

Abstract. Virtual machine based software model checkers like JPF and MoonWalker spend up to half of their verification time on garbage collection. This is no surprise as after nearly each transition the heap has to be cleaned from garbage. To improve this, this paper presents the Memoised Garbage Collection (MGC) algorithm, which exploits the (typical) locality of transitions to incrementally perform garbage collection. MGC tracks the depths of objects efficiently and only purges objects whose depths have become infinite, hence unreachable. MGC was experimentally evaluated via an implementation in our model checker MoonWalker and benchmarks using the parallel Java Grande Forum benchmark suite. By using MGC, a performance increase up to 78% was measured over the traditional Mark&Sweep implementation.

1 Introduction

Within the software development cycle, model checkers are often used to validate the initial design of a system before actually implementing it. The process of model checking usually consists of three parts: modelling, specification and verification. During the modelling phase, an abstraction is made from the design under verification. This abstraction – the model – is then verified against the specification. This traditional approach has its disadvantages. For, (i) creating an abstraction at the right level is considered difficult, (ii) the abstraction is crafted manually and prone to human error, (iii) the model and its semantics are bounded to the expressiveness of the modelling language, which generally tend to be rigourously formalised (e.g., process algebra's, state machines) [4] and (iv) after validation, the model still has to be transformed to an implementation. As fully automated code generators do not exist (yet), parts of this refinement step have to be done manually.

Software model checking overcomes these labor intensive problems by verifying the implemented system directly instead of the abstract model. This approach has been pioneered by Klaus Havelund [9] in the first version of the Java PathFinder (JPF). This initial version of JPF comprised a Java to Promela translator that enabled the verification of Java programs using the model checker



(a) Java PathFinder (b) MoonWalker

Fig. 1. Profiler data from Raytracer 3-1 benchmark [22] (see Section 4.2) describing the stakes of garbage collection (GC), state storage (State), exploration (DFS) and all remaining functionality (Misc.).

Spin [10]. This experiment highlighted many challenges associated with the software model checking approach. The foremost problem was overcoming the semantic gap between Java and Promela. This was solved by introducing the bytecode interpretation approach in the second iteration of jpf [23, 25]. Ever since, more techniques have been developed to make software model checking more effective [15]. Within the context of this paper, the emergence of thread and heap symmetry reduction techniques [11, 12, 17] are most important.

These developments gave rise to other software model checkers, like xrt [8], BOGOR [20], Bandera [4] and MoonWalker [2, 21, 26]. The latter is a software model checker developed at the University of Twente. It verifies CIL assemblies – better known as Microsoft .net programs – for assertion violations and deadlocks. It is written in C#, runs on Windows, Linux and MacOS X and its main purpose it so serve as a testbed for experimenting with novel model checking techniques. The architecture of the initial version [1] was heavily inspired by jpf's. In terms of performance, MoonWalker is comparable with jpf. For the second iteration of MoonWalker, we developed new techniques for speeding up verification [18]. In this paper, we present one of our contributions.

Using a profiler, we observed that the software model checkers typically spend around half of the time on garbage collection (see Figure 1 for an illustrating example). This does not come as a surprise as after most transitions the heap has to be cleaned from garbage. To lower the stake of garbage collection and therefore reduce the time needed for verification, we developed a new algorithm called the Memoised Garbage Collector (MGC). This algorithm is inspired by incremental graph updates from graph grammars and routing [19], and has a more favourable time-complexity compared to the often used Mark&Sweep (M&S) algorithm [16]. The key idea here is that instead of calculating reachability of a vertex (like M&S does), *track the depths of the objects efficiently and purge objects whose depth becomes infinite*, i.e., became unreachable.

This paper is further organised as follows. Section 2 outlines background research. This is followed by a description of MGC in section 3. Section 4 describes

the benchmark setup, results and discussion. This paper ends on directions for future work (section 5) and the conclusions.

2 Background

Notation-wise, in this paper, a directed graph G with a source vertex is defined as $G = (V, v_0, E)$, where V is the set of vertices, E the set of edges, and $v_0 \in V$ is the initial, root vertex. The direct predecessors of a vertex u in a graph G is defined as the set $Pred(G, u)$. The set of direct successors of a vertex u are defined as $Succ(G, u)$.

2.1 Garbage Collection for Symmetry Reduction

Garbage collection [14] is a form of automatic memory management. It is the process of reclaiming memory allocations that will not be used in the future, thereby freeing up memory. Garbage collection is a rather expensive process, it usually requires the traversal of all memory allocations before it is decidable which allocations can be reclaimed. Within the context of software model checking, garbage collection is used for a slightly different purpose, as identified by Iosif [13].

The scenario of a typical software model checker is as follows. Consider an object-oriented language that disallows pointer arithmetic, like Java or C#. Objects used by a program are internally stored in an array. Yet, because pointer arithmetic's is disallowed, the index of an object (i.e., its address) has no semantic value. Objects can only be reached via dereferencing. When references between objects in an array are mapped, the resulting graph is a heap graph. The shape of the heap graph is of semantic value, because the references between objects are. Due to different interleavings of a program, the software model checker can reach different states such that both have the same heap graph shape, but the objects in question are permutated differently in the respective arrays. If states are matched by simply matching array-equivalence, the semantically equivalent heaps will be seen as different, thereby increasing the number of states unnecessary. Detection of semantically equivalent heaps is called *heap symmetry detection* [11, 15].

To date, two variants of heap symmetry reduction are known to be effective. The technique of Iosif [11] traverses the full heap graph and creates a canonical array of objects out of it. This canonical array is stored in the hashtable. Upon state matching, the state to be matched is canonicalised and then the canonicalised arrays are matched. The technique of Lerda et al. [15] maintains a canonicalised array, instead creating one when necessary. The latter is employed by MoonWalker. Both rely on the garbage collection algorithm to function. A heap graph traversal is needed for purging unreferencable, i.e., garbage, objects. This stems from an important observation by Iosif that garbage objects may differ between states that have different paths leading to them, but are equivalent when canonicalised [13].

In software model checking though, it is observable that changes between successive states are small. Hence, the changes to the heap graph are also small. This can be exploited by tracking these changes and have them drive the garbage collection algorithm. Time can be saved for especially large heaps.

2.2 Incremental Shortest Path

To take advantage of the small changes between successive states, we propose a garbage collection algorithm inspired by an *incremental shortest-path algorithm*. A generalised algorithm for single-source directed graphs with positive weights was devised by Ramalingam and Reps [19]. See Algorithm 1. It can be viewed as an incremental version of Dijkstra's shortest path algorithm [5].

Traditionally, depths of vertices are computed all at once using Dijkstra's algorithm, stored and used when necessary. We use $depth(u)$ to indicate the stored depth of vertex u. When the graph changes from G to G', the real depths of the vertices may change. This is however not reflected in the stored depths. Thus usually, upon a change to the graph, Dijkstra's algorithm is called to globally recompute the stored depths.

For large graphs, it is more efficient to recompute only the stored depths of vertices whose real depths have changed. To date however, there is no method to determine efficiently and precisely this set of vertices. However an over-approximation of this set can be traversed by using Ramalingam and Reps's notion of inconsistency and a top-down traversal order. The former is defined as follows:

> Given the stored depth mapping $depth$, a graph $G' = (V', v'_0, E')$ and the right-handside function $rhs(G', u) = \min_{v \in Pred(G',u)} depth(v) + 1$, a vertex $u \in V'$ is inconsistent if $rhs(G', u) \neq depth(u)$.

Inconsistent vertices are spotted cheaply by monitoring the changes to the predecessor transitions upon graph changes, as shown later in Section 3. Then, the inconsistent vertices are traversed according to their key, which is defined as the minimum of the rhs and the stored $depth$: $key(G', u) = \min(rhs(G', u), depth(u))$. The vertex u with the lowest key is processed first, see line 2 of Algorithm 1. It is the inconsistent vertex closest to the root. If there are multiple vertices with the

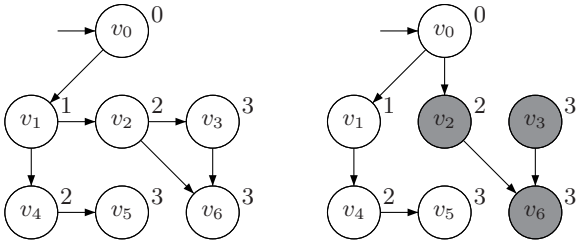
Algorithm 1: RamalingamReps()

Data: graph $G' = (V', v'_0, E')$

```
1 while G' contains inconsistent vertices do
2     u ← the vertex with the lowest key
3     if rhs(G', u) < depth(u) then
4         depth(u) ← rhs(G', u)
5     else if depth(u) < rhs(G', u) then
6         depth(u) ← ∞
```

same lowest key, one is selected non-deterministically. In case its *rhs* is smaller than its stored depth, we know that the changes to the graph moved u closer to the root. We can assign its *rhs* value to *depth* to make it consistent (line 3-4). This could cause its successors, $Succ(G', u)$, to become inconsistent, and they will be processed when their key is the lowest. On line 5-6, we deal with the case that $rhs(G', u)$ is greater than $depth(u)$, thus it moved farther from the root. We assign its stored depth with infinity (∞). This ensures vertex u's key is purely determined by the *rhs* and if it is the lowest, it will be processed again. The cause-and-effect behaviour of making vertices consistent and triggering its successors become inconsistent is guaranteed to reach a fixpoint because of the traversal order by the lowest key. A proof of correctness is provided in [19].

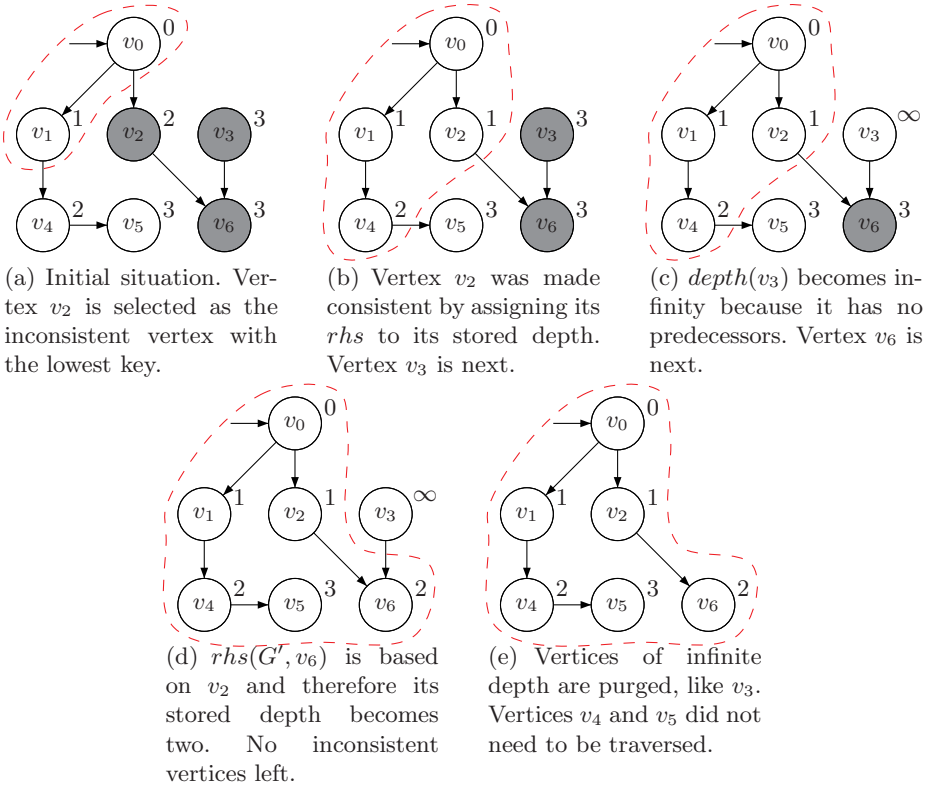
Intuitively, this algorithm determines a subgraph of vertices for which the stored depths reflect the real depths. This is ensured for consistent vertices whose stored depth is smaller or equal to the vertex with the smallest key value. Based on this subgraph, the inconsistent vertex closest to this subgraph is made consistent. This enlarges the subgraph. This is recursively done until all inconsistent vertices are traversed and the subgraph is equal to the graph. A walkthrough of this algorithm is shown in Figure 3. It outlines the steps of Ramalingam and Reps's algorithm on graph G' from Figure 2.



(a) Graph G (b) Graph G'

Fig. 2. Graph G was changed to graph G', but the stored depths (the labels upper-right from the vertex) were not recomputed. Because of this, vertices v_2, v_3 and v_6 are inconsistent, as indicated by the gray fill in graph G'.

The foremost application of Ramalingam and Reps's algorithm is in routing. Routers need to recalculate shortest paths to neighbouring routers when the connections change. Whereas Dijkstra's algorithm recalculates all shortest paths, this algorithm only recalculates shortest paths that have actually changed. For large networks, this algorithm reduces time. In this paper, we show how the idea behind this algorithm improves garbage collection in software model checking.



(a) Initial situation. Vertex v_2 is selected as the inconsistent vertex with the lowest key.

(b) Vertex v_2 was made consistent by assigning its rhs to its stored depth. Vertex v_3 is next.

(c) $depth(v_3)$ becomes infinity because it has no predecessors. Vertex v_6 is next.

(d) $rhs(G', v_6)$ is based on v_2 and therefore its stored depth becomes two. No inconsistent vertices left.

(e) Vertices of infinite depth are purged, like v_3. Vertices v_4 and v_5 did not need to be traversed.

Fig. 3. Walkthrough of Ramalingam and Reps's algorithm on graph G' of Figure 2. The vertices in the dashed subgraph are ensured consistent.

3 Memoised Garbage Collection

Ramalingam and Reps's algorithm works on graphs in general. To make it applicable for garbage collection in software model checking, we introduce additional semantics upon it.

First, a heap does not have a single root object, but multiple, namely the objects referenced from the call stacks of the program threads (see Figure 4). To make a heap graph a single-root graph, we introduce a fictive root v_0 whose successors are the objects referenced from the call stacks. Each reference counts as a distance of one. Given these semantic additions, the resulting graph can be processed by Ramalingam and Reps's algorithm. When the algorithm terminates, objects with an infinite depth are unreachable and can be garbage collected.

Due to the dynamic nature of object oriented software, the algorithm must also deal with newly instantiated objects, as they change the heap graph. To ensure that the new object will be seen as reachable from the fictive root, the stored depth of that object must be initialised with infinity. It will then be seen

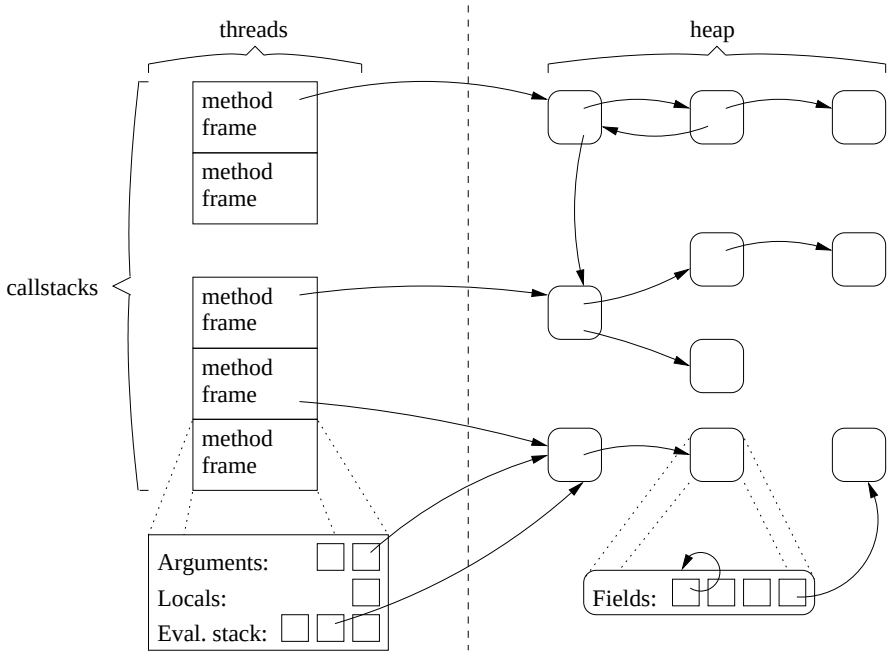


Fig. 4. Organisation of a state in a software model checker. It consists of two threads, each having a callstack. The three objects on the left are root objects, as they are referenced directly from the call stacks.

as inconsistent upon the first next run of the algorithm, and as such, it will be made consistent by assigning it with a consistent depth.

3.1 Implementation

An implementation of the algorithm has three main issues to consider, namely (i) how inconsistent vertices are determined, (ii) how an inconsistent vertex with the least key is found and (iii) how predecessors of an object are determined. Algorithm 2 is an implementation of Algorithm 1 for which these issues have been resolved.

(i) The first issue is tackled by lines 3-6 and lines 15-19. Initially, a vertex could be detected on inconsistency by computing its rhs and compare it against its stored depth. Computing the rhs is expensive as it requires the traversal of all the predecessors. Therefore, we first use a safe indication, which we call "dirtyness". A predecessor set is dirty whenever it was modified, like additions and removal of predecessor objects, from the previous predecessor set, $Pred(G, o)$. Also, it is possible that during one transition, an object is created, used and discarded. Those objects have an empty predecessor set and also have to be considered for inconsistency. When an object passes these tests, then ultimately its rhs is calculated and compared against its depth. Between lines 15-19, successor

Algorithm 2: MemoisedGC(s, s')

```
   Data: priority queue Q
 1 G = (V, E, v_0) ← the object graph associated with state s
 2 G' = (V', E', v_0) ← the object graph associated with state s', the successor of s
 3 foreach object o ∈ V' do
 4     if Pred(G', o) is dirty ∨ Pred(G', o) is empty then
 5         if rhs(G', o) ≠ depth(o) then
 6             add o to Q with order key(G', o)
 7 while Q is not empty do
 8     u ← dequeue element from Q with smallest order
 9     if rhs(G', u) < depth(u) then
10         depth(u) ← rhs(G', u)
11         Affected ← Succ(G', u)
12     else if depth(u) < rhs(G', u) then
13         depth(u) ← ∞
14         Affected ← Succ(G', u) ∪ {u}
15     foreach o ∈ Affected do
16         if rhs(G', o) ≠ depth(o) then
17             if o ∈ Q then adjust o on Q with order key(G', o)
18             else add o to Q with order key(G', o)
19         else if o ∈ Q then remove o from Q
```

objects are traversed that could have become inconsistent because their common parent has become consistent. The inconsistent childs are added to the priority queue Q so that they will be made consistent.

(ii) The second issue is determining the object with the least key. Inconsistent objects added to Q are sorted by their key. Due to this order, the object with the least key can be extracted in constant time. In case the key changes of an inconsistent object's that is already in Q (because its predecessor has a changed depth), then this change is reflected by an update to the queue, as done in line 17.

(iii) The third issue relates to function rhs and Algorithm 2. The heap only stores the successor relation explicitly. The predecessor relation can be derived implicitly from this. However, to speed up the algorithm, we maintain an explicit predecessor relation. In MoonWalker, this relation has to be updated in the following situations:

- Upon interpretation of the `stfld` instruction, if an object reference is stored into an object's field.
- Upon interpretation of the `stelem` instruction, if an object reference is stored into an array element.
- Upon a `System.Array.ArrayCopy` internal call, when object references are copied to the destination array.
- When an object reference is pushed on the call stack; then the referenced object becomes a child of the fictive root.

– When an object reference is popped from the call stack; then the referenced object is removed as child of the fictive root.

When state collapsion [23] is applied, the predecessor relation also has to be updated similarly upon restoring an object, array and callstack. Furthermore, the predecessor relation has to be stored as a bag (i.e., a counting set). It is possible that an object references another object multiple times by holding the same object reference in multiple fields. If one of these references is removed, then the predecessor relation still holds. The predecessor relation between objects is discarded when all references to the successor object are removed.

3.2 Time Complexity

Whereas the time complexity of M&S is lineair to the size of the heap, the time-complexity of MGC is expressed in other terms. The main term is that of an affected object, which is an object whose stored depth has changed during one run of the algorithm. The extended size of an affected object o is $|Pred(o)|$. Given these terms, [19] showed that the worst-case time-complexity of algorithm 2 is $O(N \cdot (log(N) + M))$, where N is the sum of extended sizes of affected objects plus the amount of affected objects, and M the cost to calculate rhs.

4 Experimental Evaluation

To evaluate the effectiveness of MGC, we wanted to compare it against M&S. M&S was already implemented in MoonWalker. For running the experiments, we also implemented MGC. It took us three man-months to implement it, including the learning-curve necessary to pick up the .net platform, to get familiar with MoonWalker's code and implementing several enhancements to MoonWalker in between.

4.1 Bandera's Models

Instead of crafting our own benchmarks, we purposely used existing benchmarks. Otherwise one could interpret the results with bias. Thus, we took three models from Bandera's suite, namely `Pipeline`, `SleepingBarbers` and `BoundedBuffer`, and manually ported them to C#. However, after running these academic examples through MoonWalker, we found them unsuitable for our comparison. The more favourable time-complexity of MGC would only be advantageous for models with big heaps and long verification times. The three small examples have either short verification times (around a second) or very small heaps.

4.2 Java Grande Forum Benchmarks

Benchmarks that resemble real life situations usually have bigger and complex heaps and larger state spaces, making them more interesting and challenging

to verify. The three multi-threaded models in the Java Grande Forum Benchmark suite (JGF) [22, 24] are such models. These models were developed for the scientific community to evaluate emerging parallel programming paradigms and to expose their weaknesses. Two of these models, MolDyn and Raytracer, were usable. The third benchmark, MonteCarlo, uses file I/O which is not (yet) supported by MoonWalker. The benchmarks have two parameters, denoted as $t - d$, where t is the number of threads and d is the datasize. For MolDyn, the datasize means the number of particles that is simulated. For Raytracer it means the number of pixels in both width and height that is being rendered. A higher t and/or a higher d will lead to a larger state space. Additionally, to get an idea of the models's size and complexity, its metrics are shown in table 1.

Metric	**MolDyn**	**Raytracer**
#Lines of code	965	1540
#Classes	9	17
#Methods	28	71
#Statements	433	421
#Source code size in Kb.	26	49

Table 1. Metrics of the MolDyn en Raytracer benchmarks.

As the benchmarks are written in Java, we had to convert them to C#. Due to the size of the code, converting it manually as we did for Bandera's small examples is too error-prone. Instead, we used Microsoft's Java Language Conversion Assistant 3.0, which is included with Microsoft Visual Studio 2005. The conversion was nearly complete and self-contained. The only two things that were not automatically converted were `assert` statements and final field attributes. The first was fixed by manually converting the assert statement to a `System.Diagnostics.Debug.Assert` statement in the resulting C# code. The second was fixed by adding the `readonly` attribute to fields which are marked final in the Java code.

While running initial runs, MoonWalker found an assertion violation in both models due to a datarace. The datarace occurs over the accesses to variables used to check the assertion and therefore the race does not affect the behaviour of the model. Data races in the Java Grande Benchmarks have also been detected by [6]. While the datarace can be fixed by proper synchronisation of accesses to the concerning variables, we purposely did not do that. We wanted to keep the benchmarks as pure as possible, and secondly, the datarace only increases the state space, so the only side-effect is that the model checker has to do more work.

4.3 Setup

All benchmark runs were performed on a cluster of nine identical systems. Each system has a 2.4 GHz CPU, 2 GB of memory, running Windows XP and installed

with .NET 3.0. For both benchmarks, all configurations from 2-1 to 3-3 were ran, with a total of six configurations. Each benchmark run was performed with both static and dynamic partial order reduction enabled [18], a memory threshold of 1.5 GB and a time-limit of 10 hours. A grand total of 24 runs were made, which took a day on the cluster to complete.

4.4 Results

The results of the experiment are summarised in two tables. Table 2 describes the results of the MolDyn benchmark and Table 3 describes the results of the Raytracer benchmark.

config.	gc.	heap size (#obj.)	time (sec)	memory (Mb.)	states ($\cdot 10^3$)	revisits ($\cdot 10^3$)	states stored ($\cdot 10^3$)	states stored/Mb	states/sec
2-1	MGC	45	434	1470	1482	1063	1482	1008	5863
	M&S		458	1470	1482	1063	1482	1008	5560
2-2	MGC	101	1447	o.m.	1928	790	978	652	1878
	M&S		1553	o.m.	1926	788	977	651	1748
2-3	MGC	253	78	o.m.	246	0	246	164	3163
	M&S		72	o.m.	249	0	249	166	3475
3-1	MGC	60	913	o.m.	2726	3022	1664	1109	6296
	M&S		1038	o.m.	2724	3018	1662	1108	5531
3-2	MGC	144	91	o.m.	328	0	328	218	3591
	M&S		98	o.m.	327	0	327	218	3324
3-3	MGC	372	153	o.m.	152	0	152	101	993
	M&S		68	o.m.	151	0	151	101	2238

Table 2. MolDyn results with the Memoised Garbage Collector (MGC) and the Mark & Sweep Garbage Collector (M&S).

The heap size column describes the max. heap size encountered during verification. The time column is the verification time in seconds. A verification that has run out of time is indicated by "o.t.". The memory column is the maximal memory used during verification in megabytes. A verification that has run out of memory is indicated by "o.m.". The states column is the amount of states in the state space. The revisits column is the amount of states revisited during verification. The states stored column is the amount stored in the hashtable. This may differ from the amount of states in the state space due to the ex post facto transition merger that is enabled with stateful dynamic partial order reduction [7]. Note that three columns are represented in thousands for the results from MolDyn benchmarks. The state stored/Mb. column gives an indication of the memory utilisation efficiency. The states/sec. column is the amount of states processed per second during verification. It is calculated by adding the amount of states with the revisits and have that divided by the verification time.

From the table of MolDyn, we observe that MGC is faster (in terms of states/sec) for configurations 2-1, 2-2, 3-1 and 3-2, with respectively 5%, 7%, 14% and 8% performance increase. The average performance increase with MGC on these configuration is 9%. Table 3 shows that MGC is faster for all configurations except configuration 2-3. The increases are respectively, 3% for both configurations 2-1 and 2-2, 78% for configuration 3-1, 8% for configuration 3-2 and 40% for configuration 3-3. The average performance increase of these configurations is 26%.

config.	gc.	heap size (#obj.)	time (sec)	memory (Mb.)	states	revisits	states stored	states stored/Mb	states/sec
2-1	MGC	935	1	37	844	579	844	23	1231
	M&S		1	36	844	579	844	23	1198
2-2	MGC	940	109	664	65923	53264	65923	99	1091
	M&S		113	655	65923	53264	65923	101	1055
2-3	MGC	3254	o.t.	1151	79673	19899	79673	69	3
	M&S		o.t.	1373	97233	24289	97233	71	3
3-1	MGC	1368	38	483	53631	71076	53631	111	3278
	M&S		68	475	53631	71076	53631	113	1842
3-2	MGC	1368	o.t.	1571	32520383	248967	187623	119	910
	M&S		o.t.	1572	30093872	246229	185707	118	843
3-3	MGC	1384	23	o.m.	43330	0	43330	29	1890
	M&S		32	o.m.	43323	0	43323	29	1347

Table 3. Raytracer results with the Memoised Garbage Collector (MGC) and the Mark & Sweep Garbage Collector (M&S).

We hypothesised that the increase of performance correlates with the heap size. This is partially true. We saw that Raytracer configurations have bigger heaps, and as such the performance increase is generally higher than those of the MolDyn benchmarks. The latter configurations however revealed a surprising result, namely a huge decline in performance for configuration 3-3 and a moderate decline in performance for configuration 2-3. We investigated this using a profiler and observed that our initial assumption does not always hold. We assumed that the heap does not change much between successive states. This depends however on the heap property that is being measured. The heap shape does not change much, but we did observe that the depth labelling changes much for MolDyn configurations 2-3 and 3-3. As object references are popped and pushed upon the callstacks, the successors of the fictive root change, and thus, also the object graph. Also, these affected objects can cause a chain reaction of changed depth labelling of subsequent successor objects. The MGC bases object reachability on this depth labelling.

Furthermore, the profiler revealed an overhead in the maintenance of parent lists. These list are updated upon every change to the object graph. The changes

are especially heavy when a collapsed state is restored, where it is not uncommon that many objects change.

Note that both observations depend on the model that is being verified. The Raytracer model is less susceptible to massive depth-labelling changes between successive states, thereby benefiting more from MGC.

When it comes to memory overhead (in terms of states stored per Mb.), we see that there is no significant difference between MGC and M&S. This means that the memory overhead for maintaining parentlists is neglectible.

5 Future Work

Profiler measurements revealed that MGC decreases the stake of garbage collection from 55% to 24% on Raytracer 3-1. Yet, during development of MGC, we identified several opportunities for further optimising the garbage collection process.

Implementation. The implementation can be further improved by using a HOT queue [3] instead of currently used the Interval Heap for Q in Algorithm 2. The HOT queue has a better time-complexity for monotone increasing keys, which holds for this algorithm. Also the calculation of the rhs function can be done in constant-time using the improved algorithm by [19]. Both are more difficult to implement efficiently and for this reason, it is deferred as future work.

No garbage or lots of garbage. While studying the effect of MGC, we observed that lots of calls to the garbage collector do not result in the collection of garbage objects and that some calls result in the collection of lots of objects. The first especially happens when the callstack of a thread grows by successive method calls. If one would develop a method to detect this beforehand and disable the garbage collector, time is saved, especially when combined with M&S. The latter, collection of lots of garbage, occurs when exploration comes closer to the end state. By switching from MGC to M&S, thus a hybrid-approach, would benefit here.

Other incremental shortest path algorithms. The incremental shortest path algorithm by Ramalingam and Reps set off an active field of study on incremental shortest path calculation. Since its publication, hundreds of publications describing refinements and specialisations have emerged. It is well possible that improvements have been developed that are also applicable to MGC.

Incremental cycle detection with reference counting. The improvements mentioned above are merely to improve the MGC. Our study also gave us an idea for a more fundamental improvement. MGC uses the depth of a vertex as a property to determine reachability. In the end, it is all about the latter, not about the depth. Other properties of a graph might be used instead. For example, a fundamental different approach is to combine reference counting with a form of incremental cycle detection. The incremental cycle detector exploits the changes in a transition by incrementally maintaining the list of cycles in a heap. The reference garbage collector only has to check whether the change causes the cycle to become unreachable from the object graph, and if so, collect the cycle.

Applications of incremental computation. The incremental nature of the MGC is also applicable to other algorithms. For instance, [17] describes an incremental heap canonicalisation algorithm based on Iosif's canonicalisation algorithm [11]. They use the shortest path to achieve this, and, as they suggest themselves, can be calculated incrementally. This can be further extended to gain an incremental k-BOTS algorithm, such that thread symmetries [12] can be detected incrementally.

6 Conclusions

Software model checkers spend around half of their time on garbage collection using the Mark&Sweep algorithm. To optimise this, we describe the Memoised Garbage Collecter, which has a better time-complexity than M&S. In particular, we show how depth-information can be used to determine reachability, how an incremental shortest-path algorithm can be applied to track the depths efficiently, how this algorithm drives the MGC, how this garbage collection algorithm can be implemented and finally an experimental evaluation of it on real-life benchmark models. The performance gain observed from our benchmarks is up to 78% percent, depending on the model and configuration.

Through our work on MGC, we identified several directions for future work (see Section 5) which hopefully lead to more improved garbage collection algorithms and faster software model checking in general.

References

1. N. H. M. Aan de Brugh. Software Model Checking for Mono. Master's thesis, University of Twente, Enschede, The Netherlands, August 2006.
2. N. H. M. Aan de Brugh, T. C. Ruys, and V. Y. Nguyen. MoonWalker: Verification of .NET Programs. *LNCS*, 2009. Proceedings of TACAS 2009.
3. B. V. Cherkassky, A. V. Goldberg, and C. Silverstein. Buckets, Heaps, Lists, and Monotone Priority Queues. In M. Saks, editor, *SODA'97: Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, pages 83–92, Philadelphia, PA, USA, 1997. Society for Industrial and Applied Mathematics.
4. J. C. Corbett, M. B. Dwyer, J. Hatcliff, and Robby. Bandera: A Source-Level Interface for Model Checking Java Programs. In *ICSE 2000*, pages 762–765, 2000.
5. E. Dijkstra. A Note on Two Problems in Connexion with Graphs. In *Numerische Mathematik*, volume 1, pages 269–271, 1959.
6. T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: Efficiently Computing the Happens-Before Relation Using Locksets. In K. Havelund, M. Núñez, G. Rosu, and B. Wolff, editors, *FATES/RV 2006*, LNCS 4262, pages 193–208. Springer, 2006.
7. C. Flanagan and P. Godefroid. Dynamic Partial-Order Reduction for Model Checking Software. In J. Palsberg and M. Abadi, editors, *POPL 2005*, pages 110–121. ACM, 2005.
8. W. Grieskamp, N. Tillmann, and W. Schulte. XRT- Exploring Runtime for .NET Architecture and Applications. In B. Cook, S. Stoller, and W. Visser, editors, *Proceedings of the Workshop on Software Model Checking*, volume 144, pages 3–26, 2006. Proc. of SoftMC 2005.

9. K. Havelund. Java PathFinder, A Translator from Java to Promela. In D. Dams, R. Gerth, S. Leue, and M. Massink, editors, *SPIN 1999*, LNCS 1680. Springer, 1999.
10. G. J. Holzmann. *The* Spin *Model Checker – Primer and Reference Manual*. Addison-Wesley, Boston, Massachusetts, USA, 2004.
11. R. Iosif. Exploiting Heap Symmetries in Explicit-State Model Checking of Software. In *ASE 2001*, pages 254–261. IEEE Computer Society, 2001.
12. R. Iosif. Symmetry Reductions for Model Checking of Concurrent Dynamic Software. *STTT*, 6(4):302–319, 2004.
13. R. Iosif and R. Sisto. Using Garbage Collection in Model Checking. In K. Havelund, J. Penix, and W. Visser, editors, *SPIN 2000*, LNCS 1885, pages 20–33. Springer, 2000.
14. R. Jones and R. Lins. *Garbage Collection*. John Wiley & Sons, Chichester, 1996.
15. F. Lerda and W. Visser. Addressing Dynamic Issues of Program Model Checking. In M. B. Dwyer, editor, *SPIN 2001*, LNCS 2057, pages 80–102. Springer, 2001.
16. J. McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Communications of the ACM*, 3(4):184–195, 1960.
17. M. Musuvathi and D. L. Dill. An Incremental Heap Canonicalization Algorithm. In P. Godefroid, editor, *SPIN 2005*, LNCS 3639, pages 28–42. Springer, 2005.
18. V. Y. Nguyen. Optimising Techniques for Model Checkers. Master's thesis, University of Twente, Enschede, The Netherlands, December 2007.
19. G. Ramalingam and T. W. Reps. An Incremental Algorithm for a Generalization of the Shortest-Path Problem. *Journal Algorithms*, 21(2):267–305, 1996.
20. Robby, M. B. Dwyer, and J. Hatcliff. Domain-specific Model Checking Using The Bogor Framework. In *ASE 2006*, pages 369–370. IEEE Computer Society, 2006.
21. T. C. Ruys and N. H. M. Aan de Brugh. MMC: the Mono Model Checker. *Electr. Notes Theor. Comput. Sci.*, 190(1):149–160, 2007. Proc. of Bytecode 2007.
22. L. A. Smith, J. M. Bull, and J. Obdrzálek. A Parallel Java Grande Benchmark Suite. In *ACM/IEEE Conference on Supercomputing (SC 2001)*, New York, USA, 2001. ACM.
23. W. Visser, K. Havelund, G. P. Brat, and S. Park. Model Checking Programs. In *ASE 2000*, pages 3–12. IEEE Computer Society, 2000.
24. The Java Grande Forum Benchmark Suite. http://www.epcc.ed.ac.uk/research/activities/java-grande/.
25. Java PathFinder. http://javapathfinder.sourceforge.net/.
26. MoonWalker. http://www.cs.utwente.nl/~ruys/moonwalker/.