Analysis of Pure Methods using Garbage Collection

Erik Österlund
Software Technology Labs
Linnaeus University
351 95 Växjö, Sweden
eosst08@student.lnu.se

Welf Löwe
Software Technology Labs
Linnaeus University
351 95 Växjö, Sweden
welf.lowe@lnu.se

ABSTRACT

Parallelization and other optimizations often depend on static dependence analysis. This approach requires methods to be independent regardless of the input data, which is not always the case.

Our contribution is a dynamic analysis "guessing" if methods are pure, i.e., if they do not change state. The analysis is piggybacking on a garbage collector, more specifically, a concurrent, replicating garbage collector. It guesses whether objects are immutable by looking at actual mutations observed by the garbage collector. The analysis is essentially for free. In fact, our concurrent garbage collector including analysis outperforms Boehm's stop-the-world collector (without any analysis), as we show in experiments. Moreover, false guesses can be rolled back efficiently.

The results can be used for just-in-time parallelization allowing an automatic parallelization of methods that are pure over certain periods of time. Hence, compared to parallelization based on static dependence analysis, more programs potentially benefit from parallelization.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*memory management (garbage collection)*; I.2.2 [**Artificial Intelligence**]: Automatic Programming—*automatic analysis of algorithms, program transformation*

General Terms

Algorithms

Keywords

garbage collection, automatic parallelization, dynamic analysis, pure functions

1. INTRODUCTION

Computers are getting more CPU cores, but the cores are not used very well by today's sequential programs. With Moore's law potentially coming to an end [16], a necessity to develop programs running tasks in parallel, thereby using the cores better, has emerged again. In fact, the efficiency of application programs follows the promises of Moore's law only if they execute their tasks in parallel.

Today's application programming interfaces (APIs) for programming parallel applications are getting more programmer friendly. But, regardless of how usable parallelization APIs become and whatever abstractions they provide, the complexity of parallel programming remains inherently higher than the complexity of sequential programming. Moreover, the huge amount of sequential object oriented legacy code that has been developed since the early 90ies and still is in use to date will hardly be rewritten.

Therefore, automatic parallelization of sequential object oriented programs has some advantages. The code preserves its simple sequential nature and allows programmers to focus on the problem and its solution in the system itself, rather than on how to manage correctness, liveliness, and numerical stability of parallel implementations. Legacy codes could be adopted to multi-core machines by recompilation instead of re-implementation.

However, automatically parallelized sequential programs can theoretically not achieve the efficiency of carefully manually coded and tuned parallel programs, and the practical success of automatic parallelization for object oriented languages was, to our knowledge, rather limited in the past. In functional programming, calls to pure functions may be parallelized directly, as they have no side effects. For exploiting this idea in object oriented languages by the parallelization of sequential calls to methods, analysis needs to prove their statements as mutually independent. Since typical object oriented programs operate on dynamically allocated and dynamically bounded objects, few methods can be statically proven as independent.

We observe that the independence of methods is, however, a dynamic property, changing throughout the execution of a program. Objects are mutated and then do not change state over longer periods of time, i.e., they become immutable. Methods that operate on (temporarily) pure objects become (temporarily) mutually independent pure functions and can be executed in parallel then.

Conservative static approaches to detect and parallelize pure methods attempt to prove purity in general. Our approach is dynamic and optimistic instead: if it observes that objects are not changing over time, it optimistically assumes their immutability and "guesses" purity of the methods operating on them. This can be exploited for parallelization. It

MSPC'12, June 16, 2012, Beijing, China.

also observes false guesses: methods assumed to be pure unexpectedly attempt to change the state of alleged immutable objects. In this case parallelization can be rolled back before an illegal state change occurs.

Once again, automatic parallelization cannot outperform programmers in finding optimal solutions for parallelization. But on a smaller scale, automatic approaches can better keep track of temporarily immutable objects and temporarily pure functions and exploit this for parallelization. Therefore, we (1) introduce an efficient just-in-time (JIT) approach that dynamically "guesses" immutable objects and pure functions piggybacking on a carefully designed and optimized garbage collector. We (2) introduce an efficient just-in-time approach to correctly detect all false guesses using write barriers to detect unexpected state changes. We (3) experimentally demonstrate the efficiency and the effectiveness of our approach using a proof of concept implementation: our garbage collector including analysis outperforms Boehm's stop-the-world collector (without any analysis). We (4) make first estimates about the potential benefit of the dynamic analysis with JIT-parallelization as an example. It shows good speed-ups in the most fortunate cases including time for rolling-back on false guessing, and no losses of speed in the unfortunate cases.

Section 2 introduces necessary definitions and notions. Section 3 introduces the dynamic analysis of pure methods, their parallelization and the roll back approach. Section 4 discusses implementation details necessary for achieving high analysis performance, while Section 5 shows this performance in experiments. Section 6 relates our work to prior results. Finally, Section 7 concludes the paper and presents directions of future work.

2. TERMINOLOGY AND DEFINITIONS

First, we define the notions of cells, objects and their purity property.

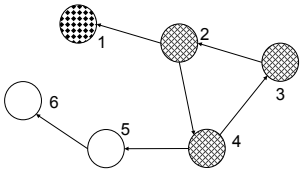
Cells and Mutators A dynamically allocated frame of memory in the heap is called a *cell*. Threads executing the actual program potentially changing cells are called *mutators*.

We use the term cell when we talk about a memory frame from the garbage collector's (GC) perspective. We rather refer to cells as *objects* when seeing them from the mutator's point of view as we specifically target object oriented programs.

Immutable and Pure Objects An object is called *immutable* iff the values of its attributes do not change. An object is called *pure* if it is immutable and can only reach immutable objects.

For pure objects, none of the transitively reachable objects can mutate. In Figure 1 the pure, immutable and mutable objects have different texture. Object 1 in this case is mutable, objects $2, 3, 4$ are immutable but not pure as object 1 is reachable. Objects $5, 6$ are immutable and pure. We call objects that are either mutable or can transitively reach mutable objects *dirty*.

Figure 1: Pure, immutable and mutable objects.



A pure object cannot change its own state nor the state of transitively reachable objects. State changes in this context are write operations to an object's attributes. We assume an environment (object) for global state changes such as I/O-operations and certain system calls. Any such state changing operation is regarded as a mutation of the environment.

Pure Methods A method is *pure* iff its receiver and all its parameter objects are pure and it does not mutate the environment.

Pure methods are side-effect-free functions. They are dependent on other methods iff they use their results or provide their parameters. Otherwise, they are independent and can be executed in parallel to other methods.

During program execution objects and their references define an object graph.

Object Graph An *object graph* $O = (N, R, E)$ is a rooted directed graph with each heap object o defining a node $n(o) \in N$. Object o contains an attribute with a reference to object o' iff $(n(o), n(o')) \in E$. Roots $R \subseteq N$ are all nodes of heap objects referenced from the stack.

Live and Dead Cells A cell is *live* iff it can be reached from a root node; it is called *dead* otherwise.

The object graph of a program changes during program execution. A GC detects the subgraphs that are not reachable from the roots in order to free the corresponding memory. Objects of dead cells can be freed.

Extending Dijkstra's color-coding, we distinguish the state of an object during garbage collection using colors of cells of the corresponding object graph.

White and Black Cells A cell is called *white* iff it has not yet been found and *black* iff itself and all transitively reachable cells have been found by the garbage collection (note that this is stricter than Dijkstra's definition).

Finally, we define Strongly Connected Components (SCCs) of directed graphs, later applied to our object graphs.

Strongly Connected Component The *Strongly Connected Components* of a directed graph $O = (N, \_, \_)$ is the partitioning $(N_1, N_2, \ldots, N_k)$ of N such that $n, n' \in N_i, 1 \leq i \leq k$ iff there is a cyclic path in O including n and n'.

3. IDENTIFICATION OF PURE METHODS

Methods operating on pure objects, i.e., with pure receiving and parameter objects, are pure functions and can be automatically parallelized. To identify pure methods, we identify pure objects. Purity of objects changes throughout the execution of a program. Therefore, we dynamically determine immutable objects, i.e., objects which do not change attributes over a period of time, and, based on that, derive the objects that are pure over that period of time.

Our approach is to "guess" whether objects are immutable or not based on the assumption that *if an object has not changed for sufficiently long time, it is probable that it will*

not do so even in the coming near future. Guessing is an automatic, dynamic, and optimistic analysis allowing to parallelize methods which are not pure functions during the *whole lifetime of all* program executions, but only during a *period of time of a particular* execution. As we no longer have to prove purity of a method regardless of the input data and the execution state of a program, we get (hopefully a lot) more opportunities for parallelization compared to what static and conservative analysis provides.

Obviously, we might guess wrongly and an object that hasn't changed over a long time will eventually change. Therefore, once we have guessed and exploited that an object is immutable, we protect against write accesses to it and cancel parallelization whenever the guess proves wrong. Note, that we protect against state changes before they occur. Therefore, we do not need to expensively roll back state transitions. We only need to repeat the parallelized work in sequential order again. We will refer to this as *re-computation.*

3.1 A Garbage Collector which Guesses Purity

Most object oriented languages come with automated memory management using a GC. Our solution to the problem of guessing pure objects is to define and exploit a custom-made GC.

Tracing live objects takes most of the time in both copying [6] and mark & sweep [24, 9] garbage collectors. This can be exploited to our advantage: as caches play an increasingly important role in performance, we can perform some extra analysis work on cells that have already been loaded into caches during tracing. We get this analysis work essentially for free, as the major part of the work had to be performed anyway.

Our solution integrates garbage collection and purity analysis. It is based on two cornerstones: concurrent replicating garbage collector [30, 31] using pointer reversal [34, 22] and the actual purity detection using a modified version of Tarjan's algorithm [35] for detecting Strongly Connected Components (SCCs). We have managed to keep all this down to one single depth-first-search (DFS) pass, making purity analysis very efficient theoretically and practically, as we will later see in the benchmarks.

3.1.1 Concurrent Replicating Garbage Collection

A *replicating* GC [30, 31] is a variation of a copying GC. A *copying* GC separates the heap in two semi-spaces: from-space and to-space. In a normal copying collector such as Baker's, only to-space is visible to the mutators. Live cells are migrated from from-space to to-space whenever a read access occurs in the wrong semi-space. When all cells have been migrated, the GC flips from-space and to-space.

A concurrent replicating GC has a few subtle changes. It is concurrent, i.e., it collects garbage while the mutators are allowed to change the object graph. From-space is visible to the mutator threads; the GC maintains to-space. In order to keep the two semi-spaces synchronized, mutations to heap cells are logged in a *mutation log.* Therefore our concurrent replicating GC uses a write-barrier (instead of a read-barrier as most copying garbage collectors).

The main idea with our approach is to analyze the mutations that become available via the mutation log, which has to be maintained anyway, and to use that information for optimistic purity analysis. The mutation log is our main source of information when we guess what objects are immutable: cells that have no records in the mutation log are guessed to be immutable.

The GC periodically reads the mutation log and updates cells in to-space as needed. As a starting condition, garbage collection kicks in when the heap is almost filled. Our starting condition is a simple percentage of the heap being filled, but there are adaptive solutions to this as well. Once it has started, the GC periodically suspends execution of the mutators at certain *safe-points* where cells in from-space synchronize with corresponding replicas in to-space. Then it samples the root set, and resumes the mutators and continues collecting garbage again. When collection is done, it suspends the execution of the mutators at a safe-point and collects the remaining live cells and flips sides.

Research by Hayes [18] shows that memory cells are created and destroyed in clusters. Copying collectors, such as ours, generally have good caching behavior since they compact the memory being used. We use depth first search (DFS) traversal, instead of the breadth first search (BSF) traversal commonly used in copying garbage collectors, which sometimes leads to even better caching behavior for many languages. For instance, Moon found out that DFS is superior to BFS in Smalltalk [29].

Pointer reversal was used for DFS. Normally it would be difficult to use pointer reversal in a concurrent GC as it changes pointer words in cells. But, since the mutator only sees from-space, we can use pointer reversal on cells in to-space. This allows us to perform DFS with $O(1)$ memory consumption and without being afraid of running out of memory for a DFS stack.

3.1.2 Purity Detection based on Tarjan's algorithm

Conceptually, we first condense the object graph into its strongly connected components (SCCs). With this new graph, we propagate a dirty property back from mutable cells using a post-order traversal of the graph. All objects whose SCC have not been marked as dirty are pure.

We developed a special variant of Tarjan's algorithm to detect SCCs. In our variant as in Tarjan's original algorithm, cells have to maintain a pointer to their "lowest" cell, which becomes the representative of an SCC. However, there are two things that we would like to remove from Tarjan's algorithm: the index fields in the nodes and the need for a stack.

Normally, Tarjan's algorithm needs to store an index representing the order in which a cell was found in the DFS traversal. With our concurrent replicating GC, this is not needed. The memory address of the cells are guaranteed to be in DFS order, so the index field is no longer necessary. Note that only replicating GCs (to our knowledge) have this property. Other GCs are either not moving GCs or have mutators with barriers that break the order in which cells are moved.

We can also eliminate the stack of cells for eventually identifying SCCs. Note, that we are not interested in the cells contained in a particular SCC. The only information relevant for us is whether cells are pure or not. Therefore, it is not necessary to know what particular cells are in the stack, only if a given cell is in the stack or not. We can decide this based on our *black* cell property. A cell is black when we retreat *from* it in DFS as all of its transitively reachable cells have been recursively visited then. If traversal finds a

cell that is not *white* (i.e. is visited and hence has a root assigned) and its "lowest" cell is not black, it is in the stack meaning it is in the same SCC as the current cell. Otherwise, it is not in the stack, hence, in another SCC. Therefore, the black flag and the pointers to the "lowest" cells provide sufficient information to determine if a cell is in the stack. Therefore, we may safely eliminate the real stack.

The *dirty* property is propagated when we retreat *to* a cell in DFS. A cell is dirty if any of its children is. It is *pure* if neither the cell itself nor its "lowest" cell are dirty. Hence, we do not need to propagate the dirty property in SCCs, avoiding a second traversal.

3.1.3 Correctness Considerations

Since we are not only focusing on detecting objects that certainly are pure, things can go wrong when speculations are wrong. The first consideration is that the algorithm for identifying pure objects might break because it is done in several steps and in parallel to the mutators. While the GC is analyzing, already analyzed cells could be mutated and hence break the purity assumption about SCCs before the decision is finalized, i.e. before semi-spaces are flipped and the mutator gets access to the analyzed cells.

Assume we have an SCC X and a cell $x \in X$. If GC first traverses X and x then mutates and adds a reference to another cell y with lower index (memory address) than x, then the lowest pointer of x will be wrong and $x \notin X$. Hence, the whole algorithm breaks as the SCCs are not detected correctly. Correcting this mistake would require another traversal of already collected heap cells.

Note that this breaks the algorithm only if cells that are guessed to be pure are actually not. More specifically, if we have a cell $x \in X$ where X is a pure SCC and x mutates after its purity is guessed. Naturally this would by definition never happen if X really was pure. If, in contrast, we have a mutable cell x such that $x \in X$, but the analysis incorrectly assumes $x \in Y$, where Y is another mutable SCC then it does not matter if it is placed in the wrong SCC as none of them were mutable. If Y would be a pure SCC, then it still would not matter as the condition for being pure is that the cell itself and its "lowest" both are not dirty. In this case the cell itself is dirty and it is correctly considered dirty.

So how to prevent the analysis from incorrectly marking an object as pure before the decision has been finalized? Each collection *cycle*—a collection cycle is the set of collections that lead up to a flip of semi-spaces—has two collections: an initial one C_i which does the actual collection concurrently to the mutators, and then an exclusive, synchronizing one C_s which collects the objects mutated during C_i and any additional roots. Then it is time to flip semi-spaces. The timing window where the algorithm could break is in C_i because it is executed in parallel to mutators.

If a supposedly pure object mutates directly, we consider it mutable. Other supposedly pure objects that should be considered dirty are still regarded as pure. This is still safe because their purity cannot be exploited for parallelization until we flip semi-spaces. Thereafter, any mutation is trapped by the mutator before it actually happens. Therefore, any situation that breaks the detection algorithm is either detected before it gets exploited by parallelization (object considered pure but is directly mutated in C_i) or trapped and reverted (object considered pure but is dirty because of indirect mutation in C_i).

3.2 Parallelization of Pure Functions

With this dynamic analysis, we suggest using it for JIT-parallelization. It determines if code can potentially be executed in parallel and, on the fly, adds parallel method invocations which are possibly selected using context-aware composition [19, 20, 27].

More specifically, if a method is identified as pure it is invoked in two variants: either in the normal sequential order or in a separate thread with the results being synchronized. These two invocations are guarded by a conditional statement checking three so-called context attributes: purity of the method, number of cores/processors currently available, and profiling information about the potential gain of using the threaded version over the sequential one from previous invocations. Based on these context attribute values, the parallel or sequential variant is selected dynamically. Since JIT compilers often use profiling anyway to determine what methods to compile to native code, the profiling information could be reused here.

Standard loop transformations can be used in preparation of parallelization: *loop unrolling* to get sequences of method invocations to parallelize when the targets are pure and *loop splitting* to separate pure and non-pure method invocations. The transformations can even be combined. Still all transformations are based on speculatively "guessed" instead of conservatively statically analyzed purity.

3.3 Reverting False Guesses

For rolling back when speculations about an object's purity prove wrong, we protect objects against any form of mutation. This allows us to protect against a violation of purity *before* it actually occurs.

Then we simply discard the work done based on incorrect purity speculations. These penalty costs ought to be minimized, i.e., we ought to discard as little as possible. Recall, that parallel method invocations are *started* in sequential execution order, executed then in parallel, and may not end in the same original sequential order. Assume, we are to execute n such invocations sequentially, we guess purity of the targets, and decide to execute them in parallel instead. Further assume invocation $k : 0 \leq k < n$ turns out to be not pure actually, i.e., the object it is invoked on is not pure. Then still the methods $l : 0 \leq l \leq k$ can continue in parallel as the sequential order of state changes is never violated.

Note that if $k = 0$, i.e., the first invocation in the sequential execution order breaks purity, this execution can continue. Hence, the sequential execution never gets discarded and incorrect guesses never affect the original sequential execution time.

The following two sections discuss how we trap mutations and our strategy to control re-computations in a special handler, resp.

3.3.1 Protecting pure objects against mutations

Mutations need to be trapped to avoid the speculatively parallelized pure method invocations from creating inconsistent global states. This can be done in several ways. One way is to let the GC take a little bit more care where it puts the cells that are copied from from-space to to-space. We could for instance copy cells that are assumed to be pure to the bottom of to-space while dirty cells are copied to the top of to-space. Then when the collection cycle is done and it is time to flip sides of the semi-spaces, we first write-protect

the pages of cells that are pure, i.e., at the bottom of to-space. The mutator threads can then determine if a cell is pure by looking at its address and seeing if it is inside of the region we have write-protected.

If a mutator then attempts to mutate the cell, we could trap to the operating system and handle the problem with a re-computation handler by removing the write-protection from the corresponding page and thereby marking all cells within it dirty for the mutators. While this is a simple approach, some pure objects on the same page would then be marked as dirty. GC would find and mark them as pure again the next collection cycle.

The main problem with this approach is that mutators are slowed down and we have to trap to the operating system in case our guesses of purity were wrong. This overhead even occurs regardless of whether we used the analysis for parallelization or not. Even if we had only one mutator thread, we still cannot write to the write-protected cell and hence a re-computation is necessary regardless.

Another problem with this approach is that even cells that did not mutate will be punished if they are on the same page. If we remove write protection to accept a mutation to one cell, all other cells on the same page lose their protection as well. As we cannot detect mutations to these other cells any longer, we need to assume conservatively that mutations occur. Hence, we must stop and re-compute sequentially all parallelized threads that assume these other cells are pure (and protected).

Therefore, we have chosen another solution. Recall, we need to log any mutation to any object because this is how the replicating GC works and we decide now to do logging *before* any such mutation occurs. As a tiny extension of this logging, we additionally check if we attempt to mutate a pure object and notify the re-computation handler in that case (and would not do the mutation). This branch to the re-computation handler is very cheap because of branch prediction: we can easily see that almost never does a mutator try to mutate a cell guessed to be pure.

With this approach we do not have to write-protect the pages. It removes the problem of uselessly managing re-computations in situations where we do not parallelize. If no automatic parallelization is going on, the handler would simply authorize mutation. It also solves the problem of punishing cells on the same page as the cell violating the presumed purity.

The same technique also allows us to trap mutations at the beginning of safe-regions where for instance I/O-calls are located, where page protection would not help.

3.3.2 Re-computation handler

Although we trap each mutated object immediately, a garbage collection cycle has to be completed before we know all objects that are affected, i.e, get all dirty objects reaching the mutated object. This can be handled in several ways as discussed below.

One strategy is to mark the mutated object as dirty and only cancel the thread that encountered the mutation. Instead of waiting for the GC to determine what other objects could transitively reach this mutable object, the concurrent threads that did not yet encounter a mutation continue execution. This is possible as state has not been changed yet because they were trapped *before* the mutation occurred. The pro with this approach is that all automatically parallelized threads do not need to stop because of one single purity violation (mutation). The con is that many threads could be invalidated for automatic parallelization because of this mutation which would lead to many re-computations. If n is the number of alleged pure but actually mutable objects, the penalty would be $O(n)$ re-computing tasks in the worst case.

An alternative strategy is to re-compute immediately but only once and wait for the GC to determine exactly what cells are still pure before any further parallelization. The con is that tasks that were not affected by the object mutation get penalized. The pro is that the worst-case is only one re-computation phase per GC cycle which is $O(1)$ instead.

We have chosen the second variant because we consider it important not to perform worse than sequential execution in the worst case. The results are shown later in the benchmarks.

It should be noted that these two approaches are in no way mutually exclusive. It is possible to continue executing some threads concurrently optimistically until a certain threshold when re-computations would become expensive. Only then the system stops and waits until the GC provides better up to date analysis about the pure objects and (re-)computations resume.

4. IMPLEMENTATION

We implemented our purity analysis in C. POSIX and its `pthread` libraries have been used extensively. Our purity analysis needs to be integrated with a runtime system. Therefore, we created a runtime system in C. The GC of our purity analysis is concurrent and its synchronization reduces to root sampling and mutation logging. We carefully implemented the GC to reduce wait times and synchronization overheads and increased allocation speed. Below are some implementation details for reaching these goals.

4.1 Safe-points and Safe-regions

Our GC needs to stop the mutator for a short moment to access root samples and flip semi-spaces. This is only done when it is safe, i.e., safe-points are emitted where the mutator and collector threads are known to have a coherent state. At these points, a handshaking protocol is used to see if the GC wants to suspend mutator execution. If so, the mutators suspend their execution and give control to GC, and then continues execution again.

The more frequently safe-points are reached in the mutator code, the faster the collector will get response. This is especially important if there are several mutators, because otherwise one mutator might suspend quickly and then wait a long time for another mutator to suspend its execution. Therefore, safe-points are typically emitted in loops so that mutators can quickly respond to a hand-shake request.

Sometimes safe-points are not enough. In a blocking system call like a socket read, it is not feasible to wait until it unblocks before we can respond to GC. Instead, a safe-region is used. Before a blocking operation we tell GC that we enter such an operation, and at the end we check if the mutator is expected to be suspended.

Safe-regions are also used to trap mutations due to system call or I/O operations. This will arguably not affect performance as such a "mutation" takes a lot more time than checking if the current object, which is already cached, is considered pure.

4.1.1 Object Allocation

If only one mutator thread is running, allocating memory becomes very cheap with our GC. It increases the current allocation address and returns the old one. If the address has reached a threshold, the GC thread is started to collect garbage concurrently.

When there are multiple threads running, some kind of synchronization mechanism is needed. We used a lock free solution using thread-local storage (TLS) with an allocation buffer for each mutator thread. Then no synchronization mechanism was needed and we get rid of cache clashes when several mutator threads allocate memory close to another.

4.1.2 Logging Mutations

Logging mutations in from-space was necessary to let to-space know what has to be updated after a collection. This update is then performed by copying mutated cells that GC knows about already. Mutations to white cells don't need a copy but are noted for the analysis (flagged dirty). This way, we can reduce the space necessary for mutation logs while still being able to tell exactly what cells have mutated since the last collection.

Our GC does not log actual mutations but what heap objects did mutate (howsoever). This is done to keep caches coherent which could otherwise cause problems as we cannot write to two addresses at once [5]. As for allocation, we use TLS with one mutation buffer per thread.

Amortized over a program run, a write-barrier (as we use for logging mutation) is typically cheaper than a read-barrier as there are normally more reads than writes in a program. Also, a read-barrier often uses expensive mechanisms such as read-protecting memory pages and trapping to the OS every time a read is performed such as Baker's GC [6]. Although the overheads could be reduced a bit by migrating whole pages to to-space instead of only one object such as in the Appel-Ellis-Li GC [2], this is still expensive.

One could assume that the write-barriers of our GC are still relatively expensive compared to arguably cheaper write-barriers used in a mark & sweep GC. The latter only protects writes to pointer words while ours, as all replicating GCs, must write-protect any kind of mutation (and do a few more things as well in our case). On the other hand, since we log which cell mutated but not how, it is sufficient to use only one write-barrier for each cell between any two safe-points instead of one for each write access. So while a lot of mutations might have happened to a cell, we only have one entry in the log and one single invocation to the write-barrier. A normal write-barrier such as the one used by Kung and Song [24] would have potentially many invocations if many pointer words changed.

5. BENCHMARKS

Our benchmark suite consists of two parts. First we assess GC performance using GCBench and with a benchmark which is unfortunate for our approach: it contains small objects in an (almost) linear object graph. It was run on a 2.93 GHz Intel Core 2 Duo, 4 GB 1067 MHz DDR3 memory, running Mac OS X 10.6.7. Then we demonstrate what the actual speed-up of parallelization could be in a hypothetical example program run on a 2.53 GHz Intel Core i5 (2 CPU cores), 4 GB 1067 MHz DDR3 memory, running Mac OS X 10.7.1.

Table 1: Result of different GCs in GCBench

GC	Execution time	Collector overhead
Boehm	2,555 ms	774 ms (30 %)
Our GC	1,781 ms	263 ms (13 %)
Boehm (inc)	12,255 ms	10,474 ms (86 %)
Manual	3,083 ms	N/A
Boehm	2,017 ms	654 ms (32 %)
Our GC	1,363 ms	110 ms (7 %)
Boehm (inc)	5,521 ms	4,158 ms (75 %)
Manual	3,028 ms	N/A
Boehm	19,985 ms	1,385 ms (5 %)
Our GC	18,600 ms	1,014 ms (5 %)
Boehm (inc)	88,366 ms	69,766 ms (77 %)

A limitation of this study is that we compare our replicating GC to a conservative mark & sweep GC. Furthermore all benchmarks were written in C, but we wish to target OOP languages. More CPU cores would also be useful for the parallelization benchmark. Therefore this should be considered as an initial indication and a proof of concept. More rigorous evaluation remains to be done.

5.1 Garbage collection benchmarks

GCBench is a quasi-standard benchmark originally created by John Ellis and Pete Kovac, later modified by Hans Boehm and others. Boehm-Demers-Weiser [9] is a famous conservative GC that does not require any assistance from a run-time system nor a compiler.

The purpose of this first benchmark is to compare how our GC performs compared to Boehm's GC. Specifically, our intension is to show that dynamic purity analysis can be performed by a GC while still being competitive in performance. Our focus is *not* to show replicating GCs outperform mark & sweep even though this may be true.

As our GC needs some support from the compiler and the run-time system, the benchmark has been modified to use our run-time system. Overheads related to the run-time system are present for every GC, while things specific to our GC such as safe-points and write-barriers are present only when our GC runs the benchmark.

We fixed the heap sizes for the two GCs to 512 MB to make it invariant. Note that having a large heap does not harm Boehm's GC as it uses lazy sweeping, as confirmed in pretests of varying heap sizes. Since our GC is a variant of a copying GC, it will need about twice as many collection cycles.

Table 1 shows the result of our GC benchmarks. The first part shows the GCBench results when run with holes in the memory. The first column represents the memory management system/GC used, where Boehm (inc) refers to the incremental and generational version of Boehm's collector. The execution time column represents the time from the start to the end of the benchmark in milliseconds. The collector overhead column shows the time for garbage collection alone, again in milliseconds (and as percentage of mutator and collector execution).

It is most interesting to compare our GC against Boehm's stop-the-world collector while Boehm's incremental is outperformed by far and manual memory management is just added as a reference point. Our GC execution time and collector overhead is smaller than the others although it was

triggered more than twice as many times as the others, cf. collection cycles.

For evaluating the results, we ought to keep in mind that our GC is concurrent, and Boehm's GC is not. Therefore, the execution time for Boehm's GC includes collector overhead, while our GC is almost free from collector overhead in mutators, apart from sampling roots, the handshaking protocol, flipping sides, etc.

We measured the collector overhead for our GC by summing the total time the collector thread was awake. For Boehm's collector, we approximated the collection overhead by measuring the time difference between our execution time and Boehm's and used that as a lower bound of the garbage collection overhead.

However, reality is slightly more complicated: the mutators using our GC are not completely free from GC overhead which affects running times. Apart from that, Boehm's GC might suffer from worse caching performance with its mark & sweep approach, which also affects the running time. Conclusively, the time difference between the execution time of our collector and Boehm's is merely an approximation (as all GC benchmarks are).

With our GC, each collection cycle has 2 collections. We also need more than twice as many GC cycles, because we have two semi-spaces. Due to the analysis payload and the auxiliary information necessary to store the root and replica pointers, pointer reversal information and the black flag, our collector is a bit more wasteful in memory overhead.

The second part of Table 1 shows the same type of experiments and results when running GCBench without holes in the heap. Using the same reasoning as before, our GC has about 1/6 of the collection overhead of Boehm's collector.

What becomes clear in this benchmark is that our contiguous memory layout of cells in the heap, that a mark & sweep GC can not achieve, pays off with better caching performance. This time caches obviously played a greater role both for the collector and the mutators.

Finally, we tried running the benchmark with and without doing dynamic analysis with our own GC. Using holes, garbage collection took 257 ms without analysis and 263 ms with analysis. So our hypothesis seems to be supported by this result: we get this dynamic analysis essentially for free, which is what we wanted to show.

Finally, we defined our own benchmark to compare how the GCs perform when cells are small and our analysis memory overhead becomes more important. This benchmark constructs many linked lists of random (but deterministic) size. The results are shown in the third part of Table 1.

In this benchmark we have a lot more collection cycles because cells are small, almost three times as many, but still it performs well compared to Boehm's GC. A conclusion is that the time spent collecting is very low when the data structures being collected have a contiguous memory layout, because of the good caching performance of linear memory accesses.

5.2 Parallelization Benchmark

We constructed a benchmark to determine how good the garbage collector is at dynamically detecting temporarily pure objects in a scenario where the purity of objects is impossible to determine statically.

A tree is constructed where inner nodes have keys and leaves have key-value-pairs. This reflects the structure of for instance a Generalized Search Tree (GiST)[23], B+ trees, R-trees, etc. We traverse the tree using BFS. If a node has no value we print a message and continue. This cannot be parallelized automatically using static analysis as the print message is a global state change that would violate sequential order if run in parallel.

In our specific program, each leaf node contains a matrix A. If we reach a leaf, we compute something (the determinant $det(A)$, then a new matrix $B = A^n$ for some integer n, then the determinant $det(B)$ and finally we compare how much $det(B)^{1/n}$ and $det(A)$ diverge). If the numbers diverged over a certain limit, which depends on the floating point numbers and arithmetics, we print out a message that the result was bad. This is a state mutation that static analysis couldn't possibly detect. The program uses our GC to guess when we could run code in parallel.

Table 2: Result of analysis of BFS traversal of GiST

Run	Iterations	Tree height	Sequential execution
1	3000	4	20.3 s
2	100	5	23.0 s
3	500	4	5.0 s
4	100	4	1.1 s

Figure 2: JIT parallelization using dynamic analysis

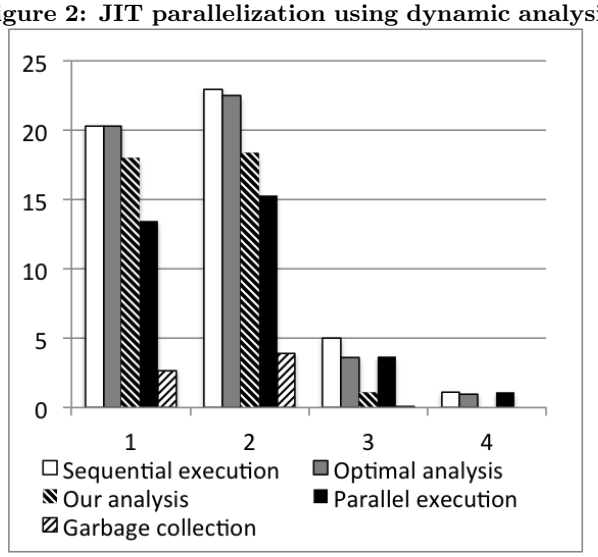


Table 2 shows the results using trees with node degree of 10 and 5x5 matrices. The first column is the run number, the second one the matrix power n. The higher n, the longer the execution time for each task. The third column represents tree height. Higher tree means more nodes, and more nodes means that GC will find the pure objects faster relative to the program execution time. The fourth column is the total execution time of the program when run sequentially.

Figure 2 shows the results of the different runs from table 2. All values are in seconds.

The first bars of each run represent the sequential execution time of the program.

The second bars show the time it takes for all leaves to execute. It is the fraction of the the sequential execution

that could run in parallel assuming perfect dynamic analysis. Our GC can not suggest to execute all leaves in parallel because it needs to see that objects are not mutated first.

The third bars show the fraction of the sequential execution that could run in parallel using the analysis from our GC. It is close to the optimum in the first two runs. However, when run-times get too small, it does not have the time to provide analysis as garbage collection merely had time to finish. This shows in the third and fourth run.

The fourth bars show the required execution time of the program when JIT-parallelization code is used based on the dynamic purity analysis of our GC, i.e. code is executed in parallel as soon as the GC says methods are pure. The time for GC and for dynamic code selection (sequential vs parallel variants) is included.

The last bars show the garbage collection overhead alone. Note that the full potential of the parallelization was not reached as it ran on a dual core CPU, and there are 3 competing threads when garbage collection occurs.

Additionally, we constructed an interesting special version of run 1 where all assumptions about purity were forced to be wrong. It was designed to demonstrate the worst case when re-computations are needed as many times as possible. As soon as a task is scheduled to be run in parallel, it mutates immediately and the re-computation handler is forced to preserve sequential execution as described in Section 3.3. The execution time using JIT-parallelization was $20.9s$ against the $20.3s$ when run sequentially. This shows the benefit of not having to do a full roll-back, just re-computations.

6. RELATED WORK

The related work on garbage collection has been discussed already together with our dynamic analysis approach in Sections 3 and 4. In this section, we relate to other automated parallelization approaches.

Automatic program parallelization has a long history dating back to the 1970s. An overwhelming majority of papers published in this area, e.g. [1, 37, 7, 36, 4, 21, 25], has a focus on *static* parallelization of programs dealing with *numerical* computations. More precisely, they use automatic loop-index analysis [1] with subsequent loop-dependency analysis [37, 7, 36, 4] to identify loops suitable for loop-level parallelism and loop transformations to parallelize `for`-loops over array based numerical computations, and they only deal with non object oriented languages like Fortran and C.

The number of papers dealing with automatic parallelization of object oriented languages is rather low. And again, these papers use static analysis for parallelization. Moreover, a large part of these papers are related to the so-called *Java Grande* effort (`www.javagrande.org`) to promote Java as a language suitable also for high performance computing [32]. Hence, also these papers, e.g. [3], have a strong focus on parallelizing numerical applications.

Static analysis of object oriented programs for automatically detecting purity of methods ought to combine points-to analysis, e.g., [33, 28], with subsequent side-effect analysis, e.g., [14]. However, the few papers dealing with parallelization of object oriented *general purpose* programs almost exclusively rely on manual source code annotations to decide which parts of the program to transform. Some papers, e.g. [11, 8], present different approaches for how to transform a sequential Java program into a corresponding parallel program. They are all a direct response to the new generation of computers based on multi-core processors.

Duarte *et al.* [15] use algebraic laws to define different types of static source code patterns that are possible to parallelize. They also describe how to parallelize each such identified pattern. However, they do not present any analysis detecting these patterns automatically.

Löwe *et al.* [27] adapt the above analysis and transformation techniques and combine them with context-aware composition [19, 20]. Here, static analysis identifies parallelizable components and transformations add parallel component variants to the sequential ones. The actual selection of a parallel or sequential variant is postponed to *runtime*.

Bradel *et al.* [10] also identify program parts suitable for parallelization. They present a *dynamic* analysis identifying computational intensive parallelizable loops. This analysis information is later used as input to a source code transformation parallelizing these identified loops. Once again, it is an approach focusing entirely on loop-level parallelism.

Some JIT and speculative approaches aggressively parallelize statements regardless of their independency [26, 12, 17]. They use profiling to determine if parallelization is worth it or not. Everything the mutator threads access is copied (including heap and stack) and then results are copied back. All data gets versions and versions not conforming to the sequential order are discarded. These approaches could work with hardware support [13], which is not present on the mainstream market. On standard hardware, some programs benefit but others significantly decrease performance.

7. CONCLUSIONS

We proposed an approach to dynamically guess pure methods at run-time based on analysis performed by a replicating garbage collector. This analysis can be used to automatically parallelize methods invoked on pure objects with pure arguments. Rolling back is trivial as we trap before any state violation. A proof of concept was built for evaluation. Execution time is not negatively affected by the GC nor by incorrect guesses which is confirmed by a number of initial experiments. Experiments also show promising speed-up results of parallelization.

The major next step is to integrate our approach into a real VM. This would allow a more elaborate experimental assessment of pros and cons than the ones presented here.

Minor steps aim for improving our GC using, e.g., an adaptive heap that can expand and contract. This might lead to even smaller synchronization overheads when allocating memory and removes delays related to freezing mutators when allocation is faster than predicted and the programs run out of memory. Other GC optimizations include more compact headers, handling small and big cells differently, and adding generational collection.

Another direction of future work is to use the mutation log to find what parts of an object can be split into several by assessing memory access patterns of methods. This would allow the detection for more pure objects as not the whole original object would have to be pure. Another interesting idea is to find not only pure objects, but subgraphs that are independent of each other, but not everything.

Generally, the idea of a dynamic analysis piggybacking on a GC for free could have other applications than just dynamic purity analysis of pure objects.

8. ACKNOWLEDGEMENTS

This research was supported by the Swedish Research Council under the grant 2011-6185.

9. REFERENCES

[1] A.V. Aho, M.S. Lam, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools (Second Edition)*. Prentice Hall, 2007.

[2] A. W. Appel and Kai Li. Real-time concurrent collection on stock multiprocessors. *ACM SIGPLAN Notices*, 23, 1988.

[3] P.V. Artigas, M. Gupta, S.P. Midkiff, and J.E. Moreira:. Automatic loop transformations and parallelization for Java. In *Int. Conf. Supercomputing (ICS'00)*, page 1ff, 2000.

[4] T.M. Austin and G.S. Sohi. Dynamic dependency analysis of ordinary programs. In *19th Ann. Int. Symp. Comp. Architecture (ISCA'92)*, page 342ff, 1992.

[5] A. Azagury, E. K. Kolodner, and E. Petrank. A note on the implementation of replication-based garbage collection for multithreaded applications and multiprocessor environments. Technical report, Dept. of Sys. Tech., IBM Haifa Research Lab, Israel, 1998.

[6] H. G. Baker. List procesing in real-time on a serial computer. *Comm. ACM*, 32, 1978.

[7] U.K. Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations*. Kluwer, Norwell, USA, 1993.

[8] A.J.C. Bik, J.E. Villacis, and D.Gannon. javar: A prototype Java restructuring compiler. *Concurrency - Practice and Experience*, 9:1181ff, 1997.

[9] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18:807ff, 1988.

[10] B.J. Bradel and T.S. Abdelrahman. Automatic trace-based parallelization of Java programs. In *Int. Conf. Parallel Proc. (ICPP'07)*, page 26ff, Washington, USA, 2007. IEEE Comp. Society.

[11] Michael K. Chen and Kunle Olukotun. The jrpm system for dynamically parallelizing java programs. In *30th Ann. Int. Symp. Comp. Architecture (ISCA'03)*, page 434ff, New York, USA, 2003. ACM.

[12] M. Cintra and D.R. Llanos. Design space exploration of a software speculative parallelization scheme. *IEEE Trans. on parallel and distributed systems*, 2005.

[13] M. Cintra, J.F. Martinez, and J. Torrellas. Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In *27th Ann. Int. Symp. Comp. Architecture (ISCA)*, 2000.

[14] L.R. Clausen. A Java byte code optimizer using side-effect analysis. *Concurrency: Practice and Experience*, 9(11):1031ff, 1999.

[15] R. Duarte, A. Mota, and A. Sampaioo. Introducing concurrency in sequential Java via laws. *Inf. Process. Lett.*, 111:129ff, Jan 2011.

[16] M. Dubash. Moore's law is dead, says gordon moore. *Techworld*, Apr 2005.

[17] M. Gupta. Techniques for speculative run-time parallelization of loops. In *ACM/IEEE Conf. Supercomputing*, 1998.

[18] B. Hayes. Using key object opportunism to collect old objects. *OOPSLA/ECOOP '91 Works. Garbage Collection in Object-Oriented Systems*, page 33ff, Oct 1991.

[19] Ch. Kessler and W. Löwe. A framework for performance-aware composition of explicitly parallel components. In *Parallel Computing: Architectures, Algorithms and Applications, ParCo 2007*, page 227ff. IOS Press, 2007.

[20] Ch. Kessler and W. Löwe. Optimized composition of performance-aware parallel components. *Concurrency & Computation: Practice & Experience (accepted)*, 2011.

[21] J. Knoop, O. Rüthing, and B. Steffen. Lazy code motion. In *ACM SIGPLAN 1992 Conf. Prog. language design and implementation (PLDI'92)*, page 224ff, New York, USA, 1992. ACM.

[22] D. E. Knuth. Fundamental algorithms. *The art of Comp. Prog.*, 1:501ff, Aug 1973.

[23] M. Kornacker. High-performance generalized search trees. *Proc. 24th Int'l Conf. on Very Large Data Bases*, Sep 1999.

[24] H. T. Kung and S. W. Song. An efficient parallel garbage collection system and its correctness proof. *IEEE Symp. Foundations Comp. Science*, page 120ff, 1977.

[25] A.W. Lim, S.-W. Liao, and M.S. Lam. Blocking and array contraction across arbitrarily nested loops using affine partitioning. In *8th ACM SIGPLAN Symp. Principles & practices parallel prog. (PPoPP'01)*, page 103ff, New York, USA, 2001.

[26] D.R. Llanos, D. Orden, and B. Palop. Just-in-time scheduling for loop-based speculative parallelization. In *Parallel, Distributed and Network-Based Proc., PDP 2008*, page 334ff, 2008.

[27] W. Löwe and J. Lundberg. Towards parallelizing object-oriented programs automatically. In *Book of Abstracts of the International Conference on Parallel Computing (ParCo'11)*, page 132, Ghent, Belgium, 2011.

[28] J. Lundberg, T. Gutzmann, M. Edvinsson, and W. Löwe. Fast and precise points-to analysis. *J. Information and Software Technology*, 51(10):1428ff, 2009.

[29] D. A. Moon. Garbage collection in large lisp systems. *ACM Symp. Lisp and Functional Prog.*, page 235ff, Aug 1984.

[30] S. M. Nettles and J. W. O'Toole. Real-time replication-based garbage collection. *SIGPLAN'94 Conf. Prog. Lang. Design and Implementation*, 29, Jun 1993.

[31] J. O'Toole and S. Nettles. Concurrent replicating garbage collection. *SIGPLAN Lisp Pointers*, VII:34ff, Jul 1994.

[32] M. Philippsen, R.F. Boisvert, V. Getov, R. Pozo, J.E. Moreira, D. Gannon, and G. Fox. JavaGrande - high performance computing with java. In *5th Int. Works. Applied Parallel Computing, New Paradigms for HPC in Industry and Academia (PARA'00)*, page 20ff, London, UK, 2001. Springer.

[33] B.G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In

Int. Conf. Compiler Construction (CC'03), volume 2622 of *LNCS*, page 126ff. Springer, 2003.

[34] H. Schnorr and W. Waite. An efficient machine independent procedure for garbage collection in various structures. *Comm. ACM*, page 501ff, Aug 1967.

[35] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Computing*, page 146ff, 1972.

[36] M.J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, Boston, USA, 1995.

[37] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers*. ACM, New York, USA, 1991.