RC 22020 (98884) 5 April 2001 Computer Science

IBM Research Report

Distinguishing Between Prolific and Non-Prolific Types for Efficient Memory Management

Yefim Shuf[+*] **Manish Gupta**[+] **Rajesh Bordawekar**[+]
Jaswinder Pal Singh[*]

[+]IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

[*]Computer Science Department
Princeton University
Princeton, NJ 08544

Distinguishing Between Prolific and Non-Prolific Types for Efficient Memory Management

Yefim Shuf [‡] [§] Manish Gupta [‡] Rajesh Bordawekar [‡] Jaswinder Pal Singh [§]

[‡] IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598
{yefim, mgupta, bordaw}@us.ibm.com

[§] Computer Science Department
Princeton University
Princeton, NJ 08544
{yshuf, jps}@cs.princeton.edu

ABSTRACT

In this paper we introduce the notion of *prolific* and *non-prolific* types, based on the number of instantiated objects of those types. We propose and empirically validate a new hypothesis, the *prolific generational hypothesis*, which states that the objects of prolific types have short lifetimes. We use this hypothesis to develop a new garbage collection scheme, which is similar in spirit to generational collection, but uses types rather than age to distinguish between different regions of the heap. It looks for garbage first among objects of prolific types. Our approach lets the compiler eliminate redundant write barriers by simply checking the type of objects involved in a pointer assignment. We describe techniques that enable this test to be done effectively at compile time, in spite of the complications due to polymorphism and dynamic class loading in Java. A preliminary implementation of this approach as a non-copying type-based collector in the Jalapeño Java Virtual Machine has shown very encouraging results. Compared with the corresponding generational collector, for the SPECjvm98 and SPECjbb2000 benchmarks, the number of dynamically executed write barriers is reduced by 18% to 74% (except for three programs, for which there is no reduction). The total garbage collection times are reduced by an average of 7.4% over all benchmark programs, with improvements of up to 15.2%. The overall performance improves modestly (by up to 2.8%) for all programs except for one (`mtrt`), for which there is a performance degradation of 2.2%.

1. INTRODUCTION

The popularity of the Java Programming Language [14] has rekindled a great deal of interest in automatic memory management. Java does not provide the programmer with an explicit mechanism to deallocate objects, and requires the underlying Java Virtual Machines (JVM) to support garbage collection. A large number of garbage collection algorithms have been developed by researchers over the last four decades [21, 37].

One of the popular approaches to garbage collection is known as *generational* garbage collection [23, 6]. It is inspired by an observation, known as the *weak generational hypothesis*, that most objects die young [34]. A simple generational scheme involves partitioning the heap space into two regions – a *nursery* (or new generation) and an *old generation*. All new objects are allocated in the nursery. Most collections, termed *minor collections*, only reclaim garbage from the nursery. Survivors from a minor collection are promoted to the older generation, which is subjected to collection only during infrequent, *major collections*. In order to support a generational collection, the compiler has to insert a *write barrier* for each statement writing into a pointer field of an object, to keep track of all pointers from objects in the old generation to objects in the nursery. These source objects in the old generation are added as *roots* for minor collection, so that objects in the nursery which are reachable from those objects are not mistakenly collected. Compared with their non-generational counterparts, generational collectors typically have short pauses, due to the need to look at a smaller heap partition at a time, but lead to lower throughput of applications due to the overhead of executing write barriers.

In this paper, we present a new approach to garbage collection based on the notion of *prolific* and *non-prolific* object types. An empirical study we have conducted on some well-known Java benchmarks shows that for each program, relatively few object types, which we refer to as prolific types, usually account for a large percentage of objects (and heap space) cumulatively allocated by the program. This observation has led us to propose a hypothesis that the objects of prolific types have short lifetimes, and therefore, account for a large percentage of garbage that is collectible at various points during program execution. We validate this hypothesis empirically and propose a new approach to garbage collection which relies on finding garbage primarily among prolific objects. Our approach is, therefore, conceptually similar to generational garbage collection, but it distinguishes between "generations" of objects based on type rather than age.

In our type-based garbage collector, all objects of a prolific type are assigned at allocation time to a *prolific space* (P-space), which is analogous to a *nursery* in a conventional generational collector. All minor collections are performed on the P-space. All objects of a non-prolific type are allocated to a *non-prolific space* (NP-space), which corresponds to the *old generation* in a generational collector with two generations.[1] Unlike generational collection, objects are not "promoted" from the P-space to the NP-space after a minor collection. This approach leads to several benefits over generational collection:

- The compiler is able to identify and eliminate unnecessary

[1]We can extend our approach to be analogous to a generational collector with a larger number of generations by defining multiple levels of prolificacy of types.

write barriers using simple type checks. This leads to performance benefits like:

  - reduction in the direct overhead of executing write barriers, and
  - for most write barrier implementations, a reduction in the number of spurious roots that are considered during minor collections, leading to fewer objects being scanned and potentially fewer collections.

- In a copying collector, the overhead of copying objects of non-prolific types across generations is avoided.

We have implemented our approach in the Jalapeño JVM [2]. The experience from this preliminary implementation is very encouraging. Our type-based non-copying collector delivers the advantages of generational collection (shorter pause times than non-generational collection), with reduced overhead of write barriers. Compared with the corresponding (non-copying) generational collector as well, it exhibits shorter pause times and shorter overall garbage collection times.

This paper makes the following contributions:

- It proposes and provides empirical data to support a novel hypothesis that a few prolific types in a program usually account for a large fraction of cumulatively allocated heap space and collectible garbage during program execution.

- It presents techniques to identify prolific types, and to efficiently check for an object being of a prolific type in a language with polymorphism and dynamic class loading.

- It proposes a new type-based approach to garbage collection that provides much of the benefits, like short pause times, of generational collection, but significantly reduces the overheads due to generational collection, such as executing write barriers and copying costs.

- It provides experimental results from an implementation of the type-based (non-copying) collector in the Jalapeño VM. Compared with the corresponding (non-copying) generational collector, for the SPECjvm98 [27] and SPECjbb2000 [28] benchmarks, the number of dynamically executed write barriers is reduced by 18% to 74% (except for three programs, for which there is no reduction). The total garbage collection times are reduced by an average of 7.4% over all benchmark programs, with an improvement of up to 15.2% for `javac`. The overall performance improves modestly (by up to 2.8%) for all programs except for one (`mtrt`), for which there is a performance degradation of 2.2%.

The rest of the paper is organized as follows. Section 2 presents the basic ideas underlying the notion of prolific and non-prolific types and techniques to identify prolific types. Section 3 discusses our proposed approach to type-based garbage collection. Section 4 describes an implementation of the simplest, non-copying version of the type-based collection approach in the Jalapeño VM. Section 5 presents the experimental results obtained by us. Section 6 discusses related work. Finally, Section 7 presents conclusions and ideas for future work.

2. BASIC IDEA

In this section, we introduce the concept of prolific and non-prolific types. We use this concept to propose and empirically validate a *prolific generational hypotheses*. We then describe techniques to identify prolific types for a program.

2.1 Prolific and Non-Prolific Types

It is well known that for most applications, a large percentage of the program execution time is spent in relatively small section of the code (according to a 90/10 rule of thumb, 90% of the time is spent in 10% of the code). This behavior is exploited by adaptive runtime compilers like the Hotspot compiler [19] and the Jalapeño adaptive optimization system [4], as they focus expensive optimizations on those “hot-spots”. It is not surprising that a similar hot-spot behavior is exhibited by object-oriented programs with respect to the types of objects that are created in those programs. In a study of some well-known Java benchmarks (namely, SPECjvm98 and SPECjbb2000), we have confirmed this observation. For example, in the `jack` program, 12 types account for 99% of all objects allocated during a typical stead-state phase between minor collections, occupying 96% of the heap space allocated during that phase.

We use the term *prolific* type to refer to a type that has a sufficiently large number of instances. More specifically, a type is prolific with respect to a program if the fraction of objects of this type exceeds a certain threshold [2]. All remaining types are referred to as *non-prolific*.

2.2 The Prolific Generational Hypothesis

We postulate a hypothesis that the objects of prolific types have small lifetimes – we refer to it as the *prolific generational hypothesis*. An intuitive basis for this hypothesis is that if this were not true, the application would have unsustainable memory requirements, as it would keep creating objects of prolific types at a fast pace without reclaiming sufficient space. It has been observed empirically that the size of the reachable portion of the heap is often small relative to the total amount of space cumulatively allocated by most applications [22].

Stated another way, our hypothesis predicts that the objects of prolific types die younger than objects of non-prolific types. It follows that most of the garbage collectible at various stages of the application would consist of objects of prolific types. This leads us to propose a different way of collecting garbage, described further in Section 3, which looks for garbage first among objects of prolific types.

Interestingly, the prolific generational hypothesis has some resemblance to a phenomenon commonly found in nature. Offsprings of prolific species are often short-lived [36].

2.3 Validating the Hypothesis

We now show some data on the allocation behavior and survival characteristics of a typical phase of some Java applications from the SPECjvm98 benchmark suite. We used the Jalapeño JVM configured with a generational copying collector for this study.

Table 1 presents data for a *single* representative minor collection that occurred in the steady state of each application. There was relatively little runtime compilation activity during the steady state. The third column shows the number of prolific and non-prolific types (based on a threshold of 1% of allocated objects) for the given phase (between two minor collections) for each application. In most applications, relatively few prolific types account for most of the heap space allocated during this phase (column 7). The `compress` program allocates very few objects some of which are very large arrays of primitives. The `mpegaudio` program allocates very little heap space for its objects. Since it has very few minor collections (usually no more than one), we could not collect meaningful data between two minor collections. Hence, we have

[2]In our experiments, the threshold is simply set to 1% of the total number of objects created by an application. This threshold can be adjusted to control the size of the P-space.

Table 1: Allocation and surivival characteristics of objects for a steady state phase

			ALLOCATED				SURVIVED				DEAD	
Benchmark	TypeCateg.	Types	Inst.	AverSize	%AllInst.	%AllSpace	Inst.	AverSize	Rel.%Inst.	Rel.%Space	%DeadInst.	%DeadSpace
compress	Prolific	12	123	181392	87.20	99.99	17	58	13.821	0.004	85.50	99.99
	Non-Prolific	10	18	29	12.80	0.01	-	-	-	-	14.50	0.01
db	Prolific	2	512980	16	99.65	56	585	89	0.114	0.636	99.65	55.70
	Non-Prolific	27	1799	3608	0.35	44	10	48	0.555	0.007	0.35	44.30
jack	Prolific	12	230487	42	99	96	6096	84	2.644	5.301	98.90	96.10
	Non-Prolific	24	2719	140	1	4	161	23	5.921	0.966	1.10	3.90
javac	Prolific	13	338590	26	86	81	125996	25	37.211	35.334	89.5	85.40
	Non-Prolific	82	54915	38	14	19	29886	37	54.422	52.478	10.5	14.60
jess	Prolific	7	293970	36	99.75	99.60	1970	37	0.670	0.704	99.75	99.60
	Non-Prolific	8	750	61	0.25	0.40	43	135	5.733	12.681	0.25	0.40
mtrt	Prolific	8	391305	21	98	97	45	23	0.011	0.012	97.90	96.90
	Non-Prolific	2	8340	32	2	3	-	-	-	-	2.10	3.10

omitted data for `mpegaudio`. The `db` program also allocates a small number of large arrays of references consuming a large chunk of heap space and used to sort the data. Some of our conclusions may not apply to these atypical Java programs. However, we have included data for these benchmarks (except for `mpegaudio` for this specific table) for the sake of completeness.

Column 5 shows that except for `compress`, objects of prolific types are smaller on an average than objects of non-prolific types. We found that instances of scalar objects of prolific types tend to be smaller than 32 bytes. However, because many objects of prolific types are arrays, the average size of all prolific objects, in some cases, is greater than 32 bytes.

Column 10 (marked "Rel%Inst", short for relative percentage of instances) shows that the relative survival rates (among objects allocated during this phase) are usually lower for objects of prolific types than for objects of non-prolific types. This validates the prolific generational hypothesis. An interesting result is that for most benchmarks, the average size of objects of prolific types that survived a minor collection (column 9) is larger than that of objects of prolific types that were allocated (column 5). This is probably because arrays tend to live longer than scalar objects. An exception is `compress`, where most of allocated large arrays are garbage collected. Finally, most of the dead objects and most of the garbage collected in a minor collection comes from short-lived objects of prolific-types (columns 12 and 13).

2.4 Identifying Prolific Types

We now discuss various approaches that may be used to identify prolific types. These approaches vary in terms of their overhead and accuracy.

2.4.1 Off-Line Profiling

The simplest method of identifying prolific types is to use offline profiling. In an offline run, a modified runtime system monitors memory allocation requests issued by an application and counts the number of objects of each type that are allocated on behalf of an application. In our implementation, collecting allocation profile carries a 5-10% run-time overhead. When an application (or a JVM) exits, the collected allocation profile is saved into a file. During an actual run, the runtime system uses previously collected allocation profile to perform various optimizations. Thus, no monitoring overhead is incurred during the production run of the application.

A disadvantage of this approach, as with any offline profiling method, is that it requires a separate profiling run and the allocation profiles may vary depending on input data supplied to an application. In addition, an application may consists of several execution phases with drastically different allocation profiles. An allocation profile of such an application is going to reflect its "average" behavior during the entire run. Although imperfect, this "one size fits all" approach still works quite well for many applications.

2.4.2 Adaptive Identification of Prolific Types

An adaptive approach, in contrast to the offline profiling approach, attempts to identify prolific types during the actual production run of the program. An obvious adaptive strategy would be to monitor each memory allocation in the application. However, this has the disadvantage of relatively high runtime overhead (which we have measured at 5-10% for offline runs). It should be possible to reduce the overhead of monitoring object allocations by using sampling techniques, such as those presented in [1].

An alternate approach is to monitor the garbage collections rather than memory allocations to identify prolific types. We expect most dead objects to be of prolific types. Although examining dead objects could lead to a fairly accurate characterization of types as prolific or non-prolific, it would be an expensive operation. Again, sampling techniques using *weak references* [1] could help reduce the overhead, since only a small subset of the dead objects would then be examined.

3. TYPE-BASED MEMORY MANAGEMENT

In this section we discuss a type-based approach to memory management, based on the prolific generational hypothesis. We describe how to modify a generational garbage collector to obtain a type-based collector. We describe how this methodology applies to both copying and non-copying versions of the collector. We explain how partitioning types into prolific and non-prolific allows us eliminate many write barriers with efficient compile-time analysis, in spite of the issues related to polymorphism and dynamic class loading which complicate compiler analysis. We also describe some extensions to the basic approach. Finally, we discuss the strengths and weaknesses of our collector.

3.1 Basic Approach

Our basic approach is to distinguish between prolific and non-prolific objects in the heap, and direct the collection effort first towards prolific objects.

3.1.1 Type-based allocation

The type-based memory allocator partitions heap space into a *prolific space* and a *non-prolific space*: P-space and NP-space, respectively. Note that the actual allocation mechanism is related to the kind of collector used by the system. When used with a copying (type-based) collector, the allocator uses different regions of the memory for the P-space and NP-space. With a non-copying collector, the objects of prolific and non-prolific types are tagged differently, but not necessarily allocated in separate memory regions.

Given an allocation request for an object of a certain type, the allocator checks the allocation profile of the application (with information about whether or not the type is profilic) to decide whether to place the object in the P-space or NP-space. Hence, compared to a traditional memory allocator, the allocation path of the type-based allocator would have an extra step for checking the type of the object. However, since the prolificacy of types is known at compile-time, the compiler can avoid the overhead of the type check by simply calling (and possibly inlining) a specialized version of the allocator for prolific or non-prolific types.[3]

3.1.2 Type-Based Collection

Based on the prolific generational hypothesis, the type-based garbage collector assumes that most objects of prolific types die young, and performs minor collections only on the P-space. Since objects of prolific types account for most of heap space, we hope to collect enough garbage on each P-space collection. When a P-space collection does not yield a sufficient ammount of free space, a full collection of both P-space and NP-space is performed. Because enough unreachable objects should be uncovered during a P-collection, we hope that the full collections will be infrequent. Objects remain in their respective spaces after both P-space and full collections – i.e., unlike generational collection, objects that survive a P-space (minor) collection stay there and are not "promoted" to the NP-space. This enables the compiler to eliminate unnecessary write barriers with a relatively simple type check, as described in Section 3.1.3. Given the low survival rates of objects in the P-space, we do not expect the "pollution" of P-space due to longer lived objects to be a significant problem.

In order to ensure that during a P-space (minor) collection, any object reachable from an object in the NP-space is not collected, we have to keep track of all such pointers from an object in the NP-space to an object in the P-space. This is accomplished by executing a write barrier code for pointer assignments during program execution, which records such references and places them in a write buffer. The contents of the write buffer represents roots used in a P-space collection.

3.1.3 Eliminating Write Barriers with Compile-Time Analysis

In the type-based collection, we do not move objects from P-space to NP-space or vice versa. Hence, write barriers (which are supposed to keep track of references from NP-space to P-space) can be eliminated at compile time based on a simple type check. More specifically, given an assignment statement where the pointer of an object of type `source` is assigned a value corresponding to a pointer of an object of type `target`, we express the condition for eliminating a write barrier as:

$$\begin{aligned}\texttt{EliminateWB} &= \texttt{IsProlific}(\texttt{source}) \\ &\quad \textbf{or} \ \ \texttt{IsNotProlific}(\texttt{target}) \qquad (1)\end{aligned}$$

3.1.3.1 Potential Opportunity.

Table 2 shows the percentage of pointer assignments originating from an object of a prolific type, i.e., those for which `IsProlific(source)` is true at run time. This data was obtained by running SPECjvm98 benchmarks (with size 100) and the SPECjbb2000 benchmark with the Jalapeño VM, using the optimizing compiler and a non-copying generational collector. The high percentages for all programs, except for `db`, show that there is clearly a potential to eliminate a substantial percentage of write barriers. Note that the numbers presented in Table 2 only give an upper bound on how many of the write barriers can be eliminated (based on the `IsProlific(source)` part of the test). The actual numbers are likely to be lower due to language features like polymorphism and dynamic class loading that introduce conservativeness in the compiler analysis.

[3]Our current implementation does not yet perform this optimization.

Table 2: The fraction of dynamic assignments into the fields of objects of prolific types

Benchmark	% of $P \to \{P, NP\}$
compress	53
db	1
jack	99
javac	42
jess	99
mpegaudio	63
mtrt	80
jbb	73

3.1.3.2 Conservativeness due to Polymorphism.

Given that Java is an object-oriented language with polymorphism, the above test (1) requires analyses like *class hierarchy analysis* [11] or *rapid type analysis* [5], similar those needed for inlining or devirtualization of methods. We make this conservativeness explicit by redefining the test (1) to be:

$$\begin{aligned}\texttt{EliminateWB} &= \texttt{MustBeProlific}(\texttt{source}) \\ &\quad \textbf{or} \ \ \texttt{MustBeNonProlific}(\texttt{target}) \qquad (2)\end{aligned}$$

Given a declared type T of an object o, the compiler checks for $\texttt{MustBeProlific}(o)$ by checking that all children of T in the class hierarchy are prolific, and can similarly check for $\texttt{MustBeNonProlific}(o)$.

3.1.3.3 Conservativeness due to Dynamic Class Loading.

Dynamic class loading [14] is another feature of Java that forces us to do compile-time analysis more conservatively. Again, the problem is similar to the problem with inlining of virtual methods in the presence of dynamic class loading [12, 26].

Due to dynamic class loading, a program can load a new class that is non-prolific but subclasses a prolific class, unless the prolific class is declared `final`. Note that with a type prolificacy based methodology, a newly loaded class (which the compiler does not know about) would not be classified as prolific, since no instance of that class would have been seen. Therefore, while the check `MustBeNonProlific(target)` in (2) could be done by looking at the existing type hierarchy, the check `MustBeProlific(source)` would be conservatively set to `IsFinal(source)`, so that the code continues to work correctly after new classes are loaded. In this case, the condition under which a write barrier may be eliminated becomes:

$$\begin{aligned}\texttt{EliminateWB} &= \texttt{IsFinal}(\texttt{source}) \\ &\quad \textbf{or} \ \ \texttt{MustBeNonProlific}(\texttt{target}) \qquad (3)\end{aligned}$$

It is possible to use the techniques for inlining virtual methods in the presence of dynamic class loading, like *preexistence analysis* [12] and *extant analysis* [26], to improve the effectiveness of the above test. For example, using extant analysis, if we create a specialized method in which the `source` reference is known to be extant (i.e., pointing to an existing object) [26], the test `MustBeProlific(source)` can be performed based on the existing class hierarchy, without worrying about any new classes that might be loaded. We use a much simpler technique, described below, in our implementation.

3.1.3.4 *Simplified Solution.*

We postulate that the prolific types are likely to be leaves, or close to leaves, in a type hierarchy. The intermediate classes are typically used for defining a functionality that is common to all of their children classes. The subclasses then refine the behavior of their parent classes and are usually instantiated more frequently than their respective parent classes.

While it is possible that a prolific class may have one or more subclasses that are not prolific, we have made a choice to treat all children of prolific types as prolific. This greatly simplifies the test to check if a type is definitely prolific. The test returns `true` if the declared type of the variable is prolific, and returns `false` otherwise (without looking any further at the class hierarchy). In particular, the test for eliminating redundant write barriers can be done without worrying about any new classes that may be dynamically loaded in the future (since we would regard any dynamically loaded class that subclasses a prolific type as prolific):

$$\begin{aligned} \texttt{EliminateWB} &= \texttt{IsDeclaredProlific}(\texttt{source}) \\ &\ \textbf{or}\ \ \texttt{MustBeNonProlific}(\texttt{target}) \end{aligned} \qquad (4)$$

Our decision to treat the children of a prolific type as prolific seems to work well in practice. We have profiled all SPECjvm98 applications and the SPECjbb2000 benchmark and discovered that (with a few exceptions) prolific types are indeed the leaves in a type hierarchy. We have seen only three cases in the SPECjvm98 benchmark suite where prolific types were not a leaves in a type hierarchy: only in two cases, prolific types were leaves in a class hierarchy and had sub-types that were not prolific; in one case, a prolific type had a subtype that was also prolific. Namely, in `javac`, a prolific class `Instruction` has a non-prolific subclass `Label`; in `mtrt`, a prolific class `IntersectPt` has a non-prolific subclass `CacheIntersectPt`; finally, also in `mtrt`, a prolific class `Point` has a subclass `Vector` that is also prolific:

3.2 Extensions

We now describe various extensions to the basic scheme, which can improve the performance of the system.

3.2.1 *Avoiding Scanning References to Type Information Blocks*

One of the fields in an object header usually points to a special object describing its type (or class) information. For example, in Jalapeño, this field points to a type information block (TIB) and is called a TIB field. Table 3 provides data[4]. Scanning TIB pointers for every reachable object does not seem to be necessary and can be avoided in the type-based scheme.

It is sufficient for only one object of a type to survive a collection to ensure that the TIB of that object is scanned and marked as live. The scanning of TIB fields of all other instances of that type are unnecessary, although the garbage collector will realize after reaching the TIB object that it has already been marked.

Since the number of distinct types is small, the number of objects representing them is also small. It follows that such objects can be classified as instances of a non-prolific type and placed in the NP-space. As a result, the TIB fields (which now point to the NP-space) do not have to be scanned during a P-space collection.

[4]Each SPECjvm98 ran through three iterations showing that a large fraction of scanned pointers (28%-55% depending on the benchmark) are TIB pointers. After loading a database, SPECjbb2000 ran for two minutes.

Table 3: Many pointers scanned during garbage collection are reference fields in object headers such as those pointing to the type information block (TIB) objects

	References Scanned		
Benchmark	# of TIB refs.	# of all refs.	% of TIB refs.
compress	8885294	28923650	30.719
db	1561864	2795719	55.866
jack	1446534	3796136	38.105
javac	4563270	14301008	31.908
jess	1940900	6551758	29.624
mpegaudio	409520	1421784	28.803
mtrt	2139610	3873905	55.231
jbb	2008508	5582408	35.979

3.2.2 *Processing Fewer Pointers*

The idea of not scanning TIB objects during a P-space collection can be taken further by observing that only objects of prolific types need to be scanned in the P-space collection. We will now show that in our type-based scheme, the number of pointers processed can be reduced even further during a P-space collection.

During the P-space scanning process, for each object, the garbage collector requests the list of fields that are references. This list is created when a class is loaded. Normally, the returned list contains all such reference fields. Consequently, all such references are first processed and then some of them (e.g. those pointing to young objects) are scanned.

However, in the type-based collector, there is no need to return a complete list of reference fields to the collector during a P-space collection. Only the references pointing to objects of prolific types have to be returned (because object residing in the NP-space are only scanned during a full collection). To support this extension, the class loader needs to provide to the collector with two different sets of methods returning the lists of reference fields: one (returning a partial list) for a P-space collection and one (returning a full list) for a full collection.

3.2.3 *Multiple "Generations"*

By defining several degrees of prolificacy of types, we can create several P-spaces. The prolific generational hypothesis predicts that objects corresponding to different levels of prolificacy will have different lifetimes. Each such space may be collected with a different frequency. This is analogous to generational garbage collectors employing multiple generations and promoting objects with different ages to different generations.

3.2.4 *Partitioning of P-space and NP-space*

It would be useful to have a separate sub-space within the NP-space for objects of non-prolific types that cannot point to objects in the P-space directly. By partitioning the NP-space in this manner, we can eliminate the need to maintain card marking data structures for this sub-space because a write barrier cannot be executed on any object in this sub-space.

3.2.5 *Lifetime Analysis*

Combining type-based memory management with lifetime analysis of objects may lead to further performance improvements. Although, most short-lived objects are instances of prolific types, there may be some non-prolific types whose instances are always short-lived. It may be advantageous to colocate such objects of non-prolific types with objects of prolific types in the P-space (and to treat them as if they were instances of prolific types) thereby reducing the pollution of NP-space, albeit only slightly. However, if such short-lived objects can point directly to a P-space, then this colocation can also reduce the pollution of P-space after a P-space collection.

3.3 Discussion

The type-based approach, while similar to the generational approach in spirit, has some important differences. First, it involves pretenuring objects of non-prolific types into the heap partition which is collected less often. These objects do not need to be scanned during P-space (minor) collections. However, we expect those savings to be limited because non-prolific types, by their very nature, would not have occupied a lot of space in the nursery.

Second, objects of prolific types are never promoted to the heap partition which is collected less often. This can be a double-edged sword. If objects of prolific types live for a long time, they can pollute the P-space, causing the scanning time to increase during future minor collections.[5] However, this approach can also help avoid the negative side effects of premature promotion of young objects which are going to die soon anyway (namely, dragging more objects via write buffers into the old generation; and requiring more major collections).

Third, the separation between the heap partitions based on the types of objects allows redundant write barriers to be eliminated with a simple (and well-known in the context of dealing with virtual methods) compile-time analysis. This, apart from saving the direct overhead of executing write barriers, can also help avoid adding unnecessary objects to the write buffer, thus leading to fewer roots for minor collections, and potentially, more effective garbage collection.

4. PRELIMINARY IMPLEMENTATION

We will now describe an initial implementation of the type-based approach in the Jalapeño VM [2]. Except for some low level components, Jalapeño is itself written in Java. Jalapeño supports a number of different garbage collectors, which can be used in different configurations of the VM. We implemented our type-based scheme by making a number of modifications to the non-copying generational garbage collector, which we found the simplest to start with.

4.1 Execution Modes

We modified the JVM to introduce two modes of execution, with the selection controlled by a command line parameter. In one (profiling) mode, the allocation profile is collected; in the other (production) mode, the allocation profile collected in the profiling run is used by the memory allocator and garbage collector to implement type-based heap management, and by the compiler to optimize away redundant write barrier code. These modes are similar to the compilation run (*write* mode) and the production run (*read* mode) of the quasi-static compiler [25].

4.2 Allocator and Collector

In the profiling mode, the memory allocator monitors all allocation requests, collect its type profile, and produces a file with the profile information, including class hierarchy information. In the production mode, the memory allocator uses the previously collected type profile to make allocation decisions. Also, in the production mode, the garbage collector repeatedly collects space occupied by dead objects of prolific types. When only a small portion of memory is freed, it collects the entire heap.

[5] Our experimental results show that the times for minor collections are, in fact, lower for our type-based collector.

4.3 Write Barriers

Jalapeño uses one of the bits in the object header for write barriers. When an object is created, this bit is cleared. When an object survives a minor collection, the bit is set while an object is promoted to the old generation. When a reference assignment operation (such as `putfield` or `aastore`) is executed, the write barrier code checks whether this bit is set in the object into which a new reference is stored (i.e. the source object). If it is cleared, nothing needs to be done. If the bit is set, a references to the source object is added to a write buffer, and the write barrier bit is cleared to ensure that no more than one reference to the source object is added to the write buffer. The references stored in the write buffer (in addition to the references on the stack of all threads and the references in the JTOC array) become roots of minor collections and are processed accordingly. Once a reference stored in the write buffer is processed, the write barrier bit for the object corresponding to the reference is set again.

In Jalapeño, no write barrier code is executed for `putstatic` operations, because all static variables appear in a global table called JTOC, which is scanned on each collection.

We have made modifications to this code to ensure that the write barriers work appropriately for our type-based approach. While the write barrier bit is cleared by default when objects are created, we set it for objects of non-prolific type at allocation time (reflecting our decision to "pre-tenure" these objects in NP-space). Furthermore, rather than setting this bit for all objects surviving a minor collection, we do not set this bit for objects of prolific type, because those objects are not "promoted" to the NP-space.

We modified the Jalapeño optimizing compiler to eliminate redundant write barriers during the production run of a program. The compiler analysis to identify redundant write barriers is based on the simplified solution described in Section *3.1.3.4*. However, the analysis has to deal with an additional, Jalapeño-specific issue of *boot-image*, which makes the analysis more conservative, as described below.

4.4 Jalapeño-Specific Issues Related to the Boot-Image

We now describe the boot-image issue, which arises in the context of the Jalapeño JVM, and is likely to be relevant for any JVM implemented in Java.

4.4.1 *The Boot-Image Problem*

A number of Jalapeño components (implemented in Java) necessary to bring up the JVM are first compiled into machine code by a compiler. The compiled code is then written into a file called the *boot-image*. Unlike the rest of the heap space, the portion of the heap occupied by the boot-image is not garbage collected. In addition, some of the boot-image objects cannot be moved out of the boot-image because pieces of code implemented in C (such as interrupt handling routines) point inside of Java objects (namely, arrays of integers used to store compiled machine code).

As we discussed earlier, the type-based scheme allows us to eliminate write barriers for references that are written into instances of prolific types (which ought to reside in the P-space). The problem arises when the boot-image contains an instance of a prolific type for an application. This type is likely to be one of the types defined in the Java class libraries, such as `String`, which is actively used by applications.

Given a reference to an object of a prolific type, the compiler cannot always infer whether this reference (pointing to a source object) will only point to an object that does not reside in the boot-image.[6] In such cases, if the write barrier is eliminated, the target object of a prolific type pointed to by the boot-image object may be mistakenly collected during the P-space collection (unless it is also pointed to from one of the objects residing in the NP-space which is collected only during a full collection).

4.4.2 *Unsuitable Solutions to the Boot-Image Problem*

Several obvious approaches that could alleviate this problem are not suitable. One possible solution, to move instances of prolific types from boot-image into the P-space, may not be done easily due to the restrictions on moving objects out of the boot-image, discussed above. Another solution is not to use the potentially prolific types for applications as such in the JVM implementation (and hence, ensure that they do not appear in the boot-image). Although it may be accomplished by implementing an internal version of classes that have the same functionality as those in Java libraries but different names, this solution requires a fairly cumbersome effort. Finally, if we force the garbage collector to scan the boot-image objects on each collection, we would no longer need write barriers for assignments out of objects in the boot-image. However, because of the large size of the boot-image, we ruled out this approach as well.

4.4.3 *The Adopted Solution*

Clearly, the compiler cannot eliminate a write barrier if the source reference may be pointing to an object residing in a boot-image. Consequently, we adopted the following conservative strategy. We modified the test (4) to check for the type of `source` being in the boot-image:

$$\begin{aligned} \texttt{EliminateWB} \quad = \quad & (\texttt{MustBeProlific}(\texttt{source}) \\ & \textbf{and } \texttt{MustNotBeInBootImage}(\texttt{source})) \\ \textbf{or} \quad & \texttt{MustBeNonProlific}(\texttt{target}) \qquad (5) \end{aligned}$$

The information about all of the classes in the boot-image is available for each configuration of Jalapeño.

5. EXPERIMENTAL RESULTS

In this section we present the results of our experiments which were performed on an RS/6000 system with a 333 Mhz PowerPC 604e processor and 768 MB of memory. We studied applications from the industry standard SPECjvm98 benchmark suite [27] and the SPECjbb2000 benchmark (Release 1.0) [28]. The SPECjbb2000 benchmark, which we refer to as `jbb`, is based on the pBOB (portable business object benchmark) [8], which follows the business model of the TPC-C benchmark [33] and measures the scalability and throughput of the JVM. We used the largest data set size (set to 100) to run SPECjvm98 applications. For all SPECjvm98 benchmarks, we used a 64 MB heap, except for javac (80 MB). The jbb program ran with a 128 MB heap.

5.1 Reducing the Overhead of Write Barriers

Table 4 demonstrates the effect of our write barrier elimination technique on reducing the overhead associated with execution of write barriers. Each SPECjvm98 program was executed with $-M3-m3$ flags and run through three iterations. The `jbb` benchmark ran for two minutes after the initial ramp up stage.

[6]Although, the location of an object is always known at run-time, we are interested in a compile-time solution. In some cases, the compiler may be able to trace the source reference to the site where the object it points to is instantiated.

For some benchmarks, the fraction of sites at which write barriers have been eliminated is fairly small (column 2). For others, it is quite significant and ranges from 15% to 32%. Note that the number of write barriers eliminated at compile time does not translate linearly to the number of write barriers eliminated at run time (column 5). It can be seen that the programs that are amenable to our optimization execute 18%-74% fewer write barriers which improves the throughput of these programs. Interestingly, a comparison with data presented in Table 2 suggests that there is a considerable potential for eliminating more write barriers by using more precise compiler analysis. In those programs, 3%-84% fewer entries are added to the write buffer for processing, which reduces the GC pauses and reduces the pollution of the heap. Benchmarks like `compress` and `mpegaudio` do not allocate a lot of objects which could be classified as instances of prolific types. Consequently, on these benchmarks, we are not able to eliminate write barriers. In db, most of the pointer assignments were to a few large arrays of references (which were classified as non-prolific) used for sorting data . As a result, the reduction in of write barrier overhead is insignificant.

5.2 Performance and Throughput of Applications

Table 5 shows the throughput of SPECjvm98 and SPECjbb200 benchmarks under the Jalapeño JVM built with different garbage collectors (GC): our non-copying type-based GC, the non-copying generational GC, and the non-copying (non-generational) GC. The best execution times (SPECjvm98) and the throughput (SPECjbb2000) for each benchmark are highlighted in Table 5 and then presented again for comparison in Table 6. Interestingly, our scheme yields the highest throughput numbers on three benchmarks, most notably `jack` and `jbb`.

Each SPECjvm98 program was run with $-M6-m6$ flags and was executed six times. The execution time in the initial iteration was the highest while throughput was the lowest due to compilation performed by an optimizing compiler. In each of the remaining five steady state iterations, there was very little compilation activity. After a ramp up, the `jbb` benchmark runs for two minutes. We ran this benchmark twice to ensure the representativeness of timing data.

Compared to the generational GC, our scheme performs up to 3% better on all benchmarks, except for `mtrt` where the throughput is 2% worse. (We believe this anomaly is due to data locality effects, and are investigating this further.) This is an encouraging result, suggesting that our scheme can improve the throughput of applications in systems where the generational GC is used.

Compared to the non-generational non-copying GC, our scheme performs noticeably better (by almost 12%) on `jess`, and somewhat better on `jack`, `mtrt` and `jbb` on which it shows a 1.5%-4.7% improvement. The non-generational GC performs better than our scheme on `compress`, `db`, and `mpegaudio`, probably because these programs do not generate a lot of objects. This program behavior limits the opportunities where our optimizations can apply.

5.3 Garbage Collection Statistics

Table 7 shows the benefits of our technique for garbage collection. The data was collected during performance runs for which we presented the data in Section 5.2.

Compared to the non-copying generational collector, our scheme has fewer garbage collections during the execution of `javac` and `jess`. Interestingly, on `javac`, the number of both P-space and full collections (compared to minor and major collections in the

Table 4: The reduction of write barrier overhead.

	Static count	Dynamic count					
	% of WB eliminated	# of write barriers executed			# of ref. added to the write buffer		
Benchmark	optimized	original	optimized	% elim.	original	optimized	% elim
compress	0.000	3514	3514	0	149	149	0
db	3.048	81440517	81407032	0	63	54	14.2
jack	0.840	29326975	20849265	28.9	3912	3775	3.5
javac	31.967	41026080	33728942	17.8	714812	603352	15.5
jess	15.838	30777556	8106063	73.7	1765	1602	10.2
mpegaudio	0.000	17436480	17436480	0	310	310	0
mtrt	17.187	9832002	3513591	64.3	1073	167	84.4
jbb	0.622	26740265	19967459	25.3	46707	8803	71.1

Table 5: Throughput (collected in the GC timing run). Execution times are given in secs. for SPECjvm98 programs (smaller is better) and the throughput is given in ops/sec. for SPECjbb2000 (larger is better). The best numbers are highlighted.

	Iteration					
Benchmark	0	1	2	3	4	5
Non-copying GC Type-Based						
compress	72.792	57.121	57.065	**56.994**	57.008	56.999
db	108.502	88.283	88.814	89.520	88.576	**88.249**
jack	108.169	51.003	50.757	50.879	50.850	**50.732**
javac	174.706	52.281	**52.171**	52.277	52.639	52.261
jess	88.501	31.215	**31.122**	31.969	31.992	31.251
mpegaudio	66.568	26.808	27.000	28.193	27.540	**26.785**
mtrt	55.584	23.203	**22.975**	23.566	23.071	23.455
jbb	n/a	1165.18	**1171.35**	-	-	-
Non-copying GC Generational						
compress	73.574	58.697	**58.598**	58.644	58.685	59.371
db	107.468	88.771	88.788	89.772	**88.546**	88.817
jack	110.041	51.922	51.651	51.649	**51.619**	51.670
javac	190.356	54.644	54.692	54.682	**53.369**	54.032
jess	93.093	32.464	32.725	**31.728**	32.139	32.927
mpegaudio	66.03	27.328	**27.121**	27.353	28.366	27.814
mtrt	56.732	23.403	**22.485**	24.137	22.568	23.510
jbb	n/a	1137.05	**1137.59**	-	-	-
Non-copying GC						
compress	69.191	55.289	55.294	55.311	55.311	**55.272**
db	105.818	**87.979**	88.727	88.277	88.745	88.267
jack	102.593	52.639	52.560	**52.478**	52.538	52.559
javac	144.788	**47.349**	47.533	47.499	47.586	47.577
jess	86.607	35.649	**34.813**	34.884	35.060	35.210
mpegaudio	62.103	27.233	27.040	**26.974**	27.587	28.205
mtrt	53.877	23.347	23.313	23.242	**23.239**	23.289
jbb	n/a	**1119.24**	1118.30	-	-	-

Table 6: Comparison of the best throughput results (collected in the GC timing run). Execution times are given in secs. for SPECjvm98 programs (smaller is better) and throughput is given in ops/sec. for SPECjbb2000 (larger is better). The best numbers for each benchmark are highlighted.

	Non-Copying GC				
	Generational	Non-Generational		Type-Based	
Benchmark	Raw	Raw	Normalized w.r.t Generational GC	Raw	Normalized w.r.t. Generational GC
compress	58.598	55.272	106.017	56.994	102.814
db	88.546	87.979	100.644	88.249	100.336
jack	51.619	52.478	98.363	50.732	101.748
javac	53.369	47.349	112.714	52.171	102.296
jess	31.728	34.813	91.138	31.122	101.947
mpegaudio	27.121	26.974	100.544	26.785	101.254
mtrt	22.485	23.239	96.755	22.975	97.867
jbb	1137.59	1119.24	98.386.	1171.35	102.967

generational scheme) is reduced. On `mtrt`, the number of P-space collections went up slightly. On all other benchmarks, the number of collections is the same. Overall, except for the "atypical" Java benchmarks like `compress` and `mpegaudio`, both the type-based and the generational collectors have a higher number of collections than the non-generational GC (which is expected). However the number of expensive full collections is significantly smaller in the type-based scheme (and in the generational scheme) compared to the non-generational GC. This is done at the "expense" of performing more frequent, less-costly P-space (minor) collections.

Compared to the generational scheme, the type-based GC spends less time collecting the P-space. The `javac` program is an exception. For this benchmark, the average time spent on collecting the P-space is approximately twice as large as the time spent on collecting the nursery in the generational scheme. At the same time, the average time spent on collecting the whole heap is noticeably smaller (25%). Although the type based-scheme executes more inexpensive collections than the generational scheme during the execution of the `mtrt` benchmark, its average time for collecting young objects is smaller. With exception of `mpegaudio`, the total time spent on collecting the whole heap is smaller in the type-based scheme than in the generational GC. Finally, for all benchmarks, the total time spent on garbage collection is smaller in the type-based scheme compared to the generational GC. The improvements range from 1.2% to 15.2%, with an average improvement of 7.4%. Short average GC times is an attractive characteristic of the type based scheme.

Both minimum and maximum GC pauses during collection of young and all objects are shorter in the type-based scheme than those exhibited by the generational GC. This observation is important since short GC pauses is a critical requirement for some systems. The `mtrt` program is an exception where the maximum pause time for a full collection is slightly longer.

5.4 The Impact of the Jalapeño BootImage

For most of the benchmarks, the bootimage problem was not very important and had an insignificant impact on performance. However, for `jess`, 25% of write barriers executed dynamically could have been eliminated if `java.lang.String` objects were not a part of the Jalapeño bootimage. Similarly, in `jbb`, 7% and 25% of write barriers could have been optimized away if the bootimage did not contain strings or arrays of objects, respectively.

6. RELATED WORK

Jones and Lins [21] present a comprehensive overview of various memory allocation and garbage collection strategies. Surveys by Wilson [37], and Wilson, Johnstone, and others [38], discuss uniprocessor garbage collection and dynamic memory allocation algorithms, respectively.

Exploitation of object lifetimes for garbage collection has been investigated extensively. The key observation that the lifetime of many objects is short was reported as early as 1976 [13]. This insight then led to the formulation of the *weak generational hypothesis* [34] which states that most objects die young. This hypothesis forms the basis of generational garbage collection [23, 6]: focus on reclaming objects that are most likely to die, i.e., young objects.

Stefanovic et al. have investigated alternatives to the traditional *young-first* generational collectors [29]. They propose *age-based* garbage collection [30] algorithms, some of which use an *older-first* collector, that collects older objects before the younger ones. This approach primarily reduces copying costs over the traditional generation collectors. Both the age-based and traditional generational collectors follow the same philosphy: both use age as a criterion for identifying objects for collection (young objects in generational and old objects in age-based collectors). On the contrary, the prolific garbage collector, uses the prolificacy of object types as the criterion for identifying prospective moribund objects.

Experimental studies by Zorn [39], and Tarditi and Diwan [32] have shown that the cost of generational garbage collection is between 5% to 20%. Generational collectors usually segragate heap objects by their age. However, substantial performance improvement can be achieved by allocating large objects in a separate non-copy region, usually termed as *large object space*(LOS) [10]. Identification of large objects can be an absolute measure (e.g., more than 1024 bytes [35] or 256 bytes [20]) or a relative one (i.e., identify the object type whose instances occupy substantial space [18]). Many recent generation collection implementations use the LOS for storing large objects [16, 2]. The cost of write barriers is also significant, especially for pointer-intensive applications [32]. While there have been several efforts for improving the write barrier performance [17, 18], we did not come across any work that eliminates write barriers via static compile-time analysis.

Previous studies have investigated off-line feedback-driven approaches for segragating objects using criteria such as object lifetime and reference behavior. Barrett and Zorn [7] use full-run profiles on allocation-intensive C programs to predict short-lived objects, place them contiguously and delay their deallocation until large 4KB batches become free. Seidl and Zorn [24] propose partitioning heap for storing objects according to their reference behavior (frequently vs. infrequently referred objects) and lifetimes (short-lived vs. long-lived) objects. Blackburn et al. describe

Table 7: Garbage collection statistics.

Non-copying GC Type-Based

	# of GCs			GC Time			P-sp. col. pause time		Full col. pause time	
Benchmark	P-space	Full	Total	P-space	Full	Total	Min	Max	Min	Max
compress	2	94	96	0.184	50.935	51.120	0.068	0.121	0.531	0.631
db	14	3	17	5.041	2.682	7.724	0.150	0.172	0.590	1.061
jack	64	9	73	8.682	5.724	14.406	0.101	0.455	0.592	0.674
javac	28	31	59	20.845	46.723	67.569	0.232	0.815	0.595	1.935
jess	56	11	67	7.006	7.667	14.674	0.111	0.208	0.639	0.769
mpegaudio	1	2	3	0.177	1.222	1.400	0.179	0.179	0.592	0.634
mtrt	30	5	35	8.419	4.378	12.798	0.126	0.674	0.877	1.170
jbb	10	5	15	3.497	4.911	8.409	0.266	0.572	0.658	1.344

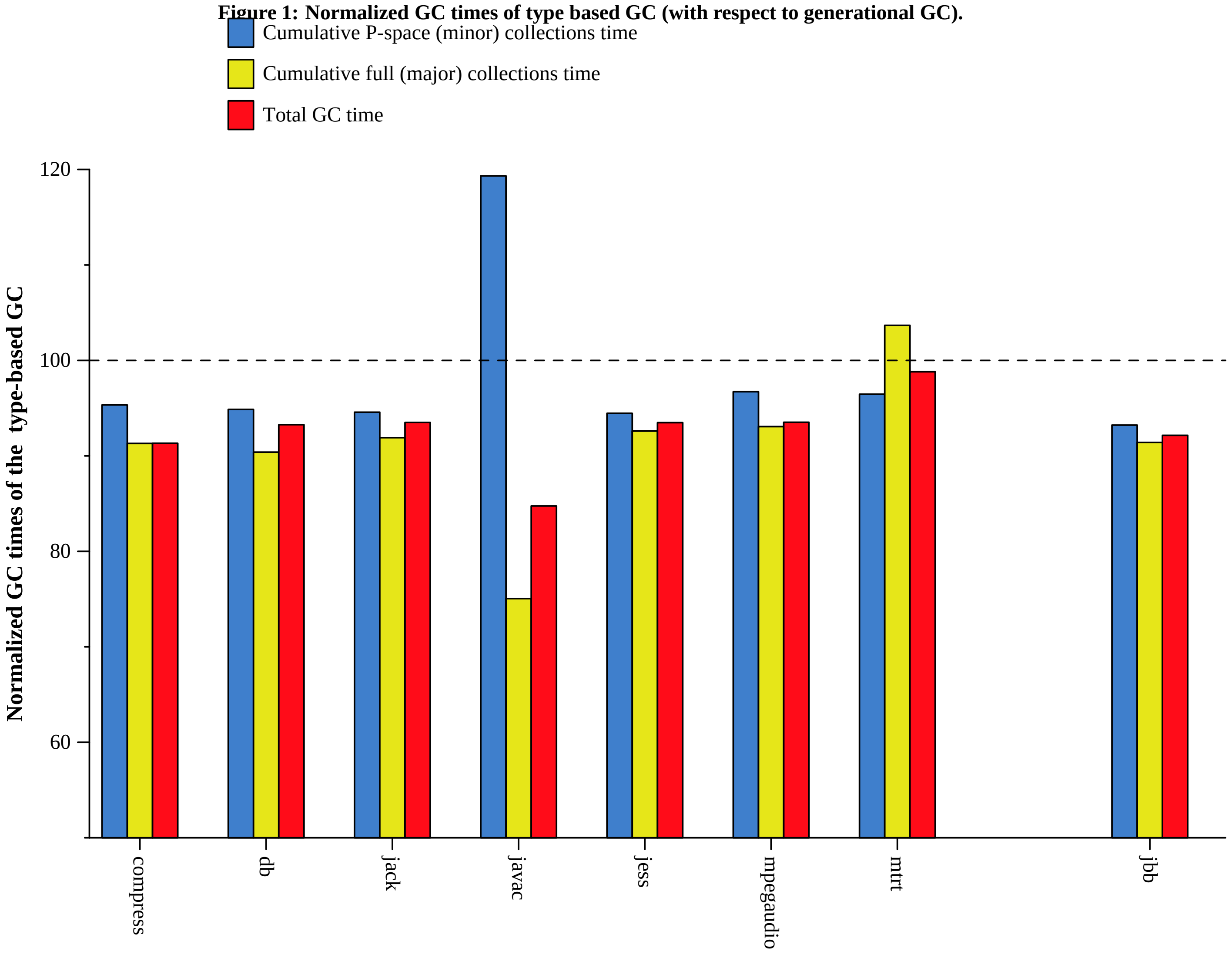
Non-copying GC Generational

	# of GCs			GC Time			Minor col. pause time		Major col. pause time	
Benchmark	Minor	Major	Total	Minor	Major	Total	Min	Max	Min	Max
compress	2	94	96	0.193	55.786	55.979	0.102	0.142	0.604	0.756
db	14	3	17	5.314	2.967	8.282	0.180	0.186	0.661	1.202
jack	64	9	73	9.180	6.228	15.408	0.130	0.490	0.661	0.743
javac	53	38	91	17.468	62.253	79.721	0.250	0.913	0.672	2.119
jess	57	11	68	7.417	8.280	15.697	0.137	0.234	0.717	0.843
mpegaudio	1	2	3	0.183	1.313	1.497	0.209	0.209	0.660	0.703
mtrt	27	5	32	8.728	4.223	12.952	0.169	0.766	0.660	1.043
jbb	10	5	15	3.751	5.373	9.125	0.304	0.641	0.726	1.600

Non-copying GC

	# of GCs			GC Time			Minor col. pause time		Major col. pause time	
Benchmark	Minor	Major	Total	Minor	Major	Total	Min	Max	Min	Max
compress	.	96	96	.	45.563	45.563	.	.	0.436	0.505
db	.	14	14	.	10.710	10.710	.	.	0.448	0.826
jack	.	62	62	.	31.271	31.271	.	.	0.436	0.619
javac	.	58	58	.	46.948	46.948	.	.	0.436	1.080
jess	.	64	64	.	36.630	36.630	.	.	0.439	0.689
mpegaudio	.	3	3	.	1.398	1.398	.	.	0.435	0.488
mtrt	.	26	26	.	19.018	19.018	.	.	0.435	0.874
jbb	.	11	11	.	9.137	9.137	.	.	0.447	0.963

Figure 1: Normalized GC times of type based GC (with respect to generational GC).



profile-driven technique for reducing copying by *pre-tenuring* long-lived objects, i.e., storing long-lived objects in an uncollected region [9]. Recently, Harris [15] has presented a dynamic technique in which selection of objects for pre-tenuring is performed at run-time.

Stefanovic et al. [31] describe analytical models for object lifetimes in object-oriented programs. Appel [3] has proposed that a plausible object lifetime distribution should use the following property: the expected future lifetime of an object is proportional to its current age.

7. CONCLUSIONS

We have presented a new approach to memory management. It is inspired by the observation that a few prolific types in an application usually account for a large majority of objects that are cumulatively allocated; furthermore, objects of these prolific types tend to have short lifetimes. Therefore, our approach to garbage collection directs the frequent collections towards objects of prolific types, much like generational collection directs them towards young objects. This approach leads to some important advantages over generational collection – it leads to fewer write barriers, potentially more effective collections, and lower copying costs (not verified yet, because our current implementation only performs non-copying collection).

With a preliminary implementation of this approach in the Jalapeño VM, we have observed significant improvements over the generational collector. For the SPECjvm98 and SPECjbb2000 benchmarks, the number of dynamically executed write barriers is reduced by 18% to 74% (except for three programs, for which there is no reduction). The total garbage collection times are reduced by an average of 7.4% over all benchmark programs. The overall performance improves modestly for most programs.

This work opens up a number of interesting possibilities for future research. We plan to develop a copying version of our type-based collector. This would also involve allocating objects of prolific and non-prolific types in distinct regions of memory. It would be interesting to study the impact of such an allocation policy and copying collection on the data locality characteristics.

Another interesting direction would be to investigate a more dynamic version of our approach, where the detection of prolific types is done during program execution in an adaptive manner. Changing the status of a type from prolific to non-prolific, or vice versa, would require selective recompilation of sections of code where write barriers may have been eliminated based on the older classification of types (we do not anticipate a need to undo the *effect* of a previously eliminated write barrier, as long as the existing objects are not moved across their respective spaces).

We also plan to implement some of the extensions described in Section 3.2. In particular, with the techniques that allow fewer pointers to be processed and fewer objects to be scanned, we hope to show even greater benefits from our approach.

Acknowledgments

We would like to thank the members of the Jalapeño team for providing us with an infrastructure for this work. We would also like to thank Pratap Pattnaik for valuable technical discussions.

8. REFERENCES

[1] O. Agesen and A. Garthwaite. Efficient object sampling via weak references. In *Proc. International Symposium on Memory Management*, pages 127–136, 2000.

[2] B. Alpern, C. R. Attanasio, J. J. Burton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shephard, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeno virtual machine. *IBM Systems Journal*, 39(1):194–211, 2000.

[3] A. Appel. A better analytical model for the strong generational hypothesis, November 1997.

[4] M. Arnold, S. Fink, D. Grove, M. Hind, and P. Sweeney. Adaptive optimization in the Jalapeño JVM. In *Proc. of OOPSLA 2000*, Minneapolis, MN, October 2000.

[5] D. Bacon and P. Sweeney. Fast static analysis of C++ virtual function calls. In *Proc. Conference on Object-Oriented Programming Systems, languages and Applications*, pages 324–341, 1996.

[6] H. G. Baker. Infant mortality and generational garbage collection. *SIGPLAN Notices*, 28(4):55–57, 1993.

[7] D. A. Barrett and B. G. Zorn. Using lifetime predictors to improve memory allocation performance. *ACM SIGPLAN Notices*, 28(6):187–196, June 1993.

[8] S. J. Baylor, M. Devarakonda, S. Fink, E. Gluzberg, M. Kalantar, P. Muttineni, E. Barsness, R. Arora, R. Dimpsey, and S. J. Munroe. Java server benchmarks. *IBM Systems Journal*, 39(1):57–81, 2000.

[9] S. Blackburn, J. Cavazos, S. Singhai, A. Khan, K. McKinley, and J. E. B. Moss. Profile-driven pretenuring for java. Poster in OOPSLA 2000, October 2000.

[10] P. J. Caudill and A. Wirfs-Brock. A third-generation smalltalk-80 implementation. In *Proceedings of ACM Conference on Object-Oriented Systems, Languages and Applications*, pages 119–130, October 1986.

[11] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy. In *Proc. 9th European Conference on Object-Oriented Programming*, pages 77–101, 1995.

[12] D. Detlefs and O. Agesen. Inlining of virtual methods. In *Proc. 13th European Conference on Object-Oriented Programming*, pages 258–278, 1999.

[13] L. P. Deutch and D. Bobrow. An efficient incremental automatic garbage collector. *Communications of the ACM*, 19(7), July 1976.

[14] J. Gosling, B. Joy, and G. Steele. *The Java$^{(TM)}$ Language Specification*. Addison-Wesley, 1996.

[15] T. Harris. Dynamic adaptive pre-tenuring. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management(ISMM'00)*, pages 127–137, October 2000.

[16] M. Hicks, L. Hornof, J. T. Moore, and S. M. Nettles. A study of large object spaces. In *Proceedings of the International Symposium on Memory Management*, pages 138–146, 1998.

[17] U. Hoelzle. A fast write barrier for generational garbage collectors. In *Proceedings of OOPSLA'93 Workshop on Garbage Collection*, 1993.

[18] A. L. Hosking, J. E. B. Moss, and D. Stefanović. A comparativr performance evaluation of write barrier implementations. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 92–109, October 1992.

[19] The Java hotspot performance engine architecture. `http://java.sun.com/products/hotspot/whitepaper.html.`

[20] R. Hudson, J. E. B. Moss, A. Diwan, and C. Weight. A language-independent garbage collector toolkit. Technical Report TR91-47, University of Massachusetts at Amherst, September 1991.

[21] R. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons, 1996.

[22] J.-S. Kim and Y. Hsu. Memory system behavior of Java programs: Methodology and analysis. In *Proc. 2000 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 264 – 274, Santa Clara, CA USA, June 2000.

[23] H. Lieberman and C. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, June 1983.

[24] M. L. Seidl and B. G. Zorn. Segragating heap objects by reference behavior and lifetime. In *Proceedings of Architectural Support for Programming Languages and Operating Systems*, pages 12–23, 1998.

[25] M. Serrano, R. Bordawekar, S. Midkiff, and M. Gupta. Quicksilver: A quasi-static compiler for Java. In *Proc. of OOPSLA 2000*, Minneapolis, Minnesota USA, October 2000.

[26] V. Sreedhar, M. Burke, and J.-D. Choi. A framework for interprocedural optimization in the presence of dynamic class loading. In *SIGPLAN 2000 Conference on Programming Language Design and Implementation*. ACM SIGPLAN, 2000.

[27] Standard Performance Evaluation Council. *SPEC JVM98 Benchmarks*, 1998. `http://www.spec.org/osg/jvm98/`.

[28] Standard Performance Evaluation Council. *SPEC JBB2000 Benchmark*, 2000. `http://www.spec.org/osg/jbb2000/`.

[29] D. Stefanović. *Properties of Age-based Automatic Memory Reclamation Algorithms*. PhD thesis, University of Massachusetts, Amherst, MA, February 1999.

[30] D. Stefanović, K. McKinley, and J. E. B. Moss. Age-based garbage collection. In *Proceedings of the ACM Conference on Object-Oriented Systems, Languages and Systems*, pages 370–381, October 1999.

[31] D. Stefanovic, K. S. McKinley, and J. E. B. Moss. On

models for object lifetime distributions. In *Proceedings of the International Symposium on Memory Management*, October 2000.

[32] D. Tarditi and A. Diwan. The full cost of a generational copying garbage collection implementation. In *Proceedings of the OOPSLA93 Workshop on Memory Management and Garbage Collection*, October 1993.

[33] Transaction Processing Performance Council. *TPC-C Benchmark*, 2000. `http://www.tpc.org/cspec.html` .

[34] D. M. Ungar. Generational scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, April 1984.

[35] D. M. Ungar and F. Jackson. Tenuring policies for generation based storage reclamation. *ACM Transactions on Programming Languages and Systems*, 14(1):1–27, 1992.

[36] A. R. Wallace. On the tendency of varieties to depart indefinitely from the original type. In Letters to the Royal Society, Feb. 1858.

[37] P. Wilson. Uniprocessor garbage collection techniques. Technical report, University of Texas at Austin, 1994. To appear in ACM Computing Surveys.

[38] P. Wilson, M. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In *Proceedings of International Workshop on Memory Management*, September 1995.

[39] B. Zorn. *Comparative Performance Evaluation of Garbage Collection Algorithms*. PhD thesis, EECS Department, University of California at Berkeley, 1989.