Not Just Yet - Giving Objects Time to Die for Garbage Collection

Kenneth W Fiduk
The University of Texas at Austin
fiduk@cs.utexas.edu

Kathryn S McKinley
The University of Texas at Austin
mckinley@cs.utexas.edu

Stephen M Blackburn
Australia National University
Steve.Blackburn@anu.edu.au

May 8, 2003

Abstract

Modern programming languages require garbage collection to automate memory management, but this feature still has a performance cost. Generational copying garbage collection improves performance by dividing old and young objects. It collects the young objects frequently and the old objects infrequently because young objects typically die at a high rate. Objects the collector copies into the older generation are likely to continue to survive for the near future. However, not all objects promoted possess this quality. Previous work shows that the majority of objects die within a short amount of time after allocation. Objects allocated right before collections generally are not allowed adequate time to die and subsequently are prematurely promoted to an older generation where they die shortly thereafter. These objects waste space (they quickly die in the old generation, but are not usually reclaimed until much later) and waste time (it takes time to copy the object). This work introduces the time-to-die nursery allocation algorithm for the fixed-sized generational collector. Objects allocated within a set time-to-die (TTD) bytes of a collection are not subjected to scanning; rather they become the start of the nursery following the collection. This delay assures that these TTD objects are given sufficient time to die, instead of being prematurely promoted to the older generation. By ignoring the TTD objects, fewer objects are both scanned and copied. Our results show that for the SPEC benchmarks, this TTD-Nursery reduces copying in garbage collection, but total performance suffers due to additional pointer tracking costs.

1 Introduction

The set of skills and techniques considered essential to every competent programmer has changed several times over the past several decades. Thirty years ago, memory management was the exclusive responsibility of the programmer with the creation and destruction of every object handled by the program itself. Today, with programming paradigms becoming increasingly higher level, memory management is now out of the programmer's hands. Garbage collection (GC) allows the programmer to nearly forget about memory management altogether, providing efficient collection of dead objects in memory. However, there are set-backs. The biggest of which, and the traditional primary complaint of GC, is the actual time spent in collection.

The *weak generational hypothesis* [7, 5] motivated the introduction of the generational garbage collector. By collecting the young generation, or *nursery*, more frequently than the old generation, the generational garbage collector cuts down on pause time by collecting only a fraction of the heap instead of the entire heap at once. Because the majority of objects die young, only a small amount of objects in the nursery actually survive between collections with the vast remaining objects no longer needed, allowing for their memory space to be reused. This behavior reduces collector work and improves throughput. This hypothesis is usually true for most of the nursery; however, it falls apart when looking at the very youngest objects in the nursery. Often times, there is a higher survival rate for the most recently allocated objects in the nursery compared to other objects. This property is not because the majority of recently allocated objects live longer, but rather

because they simply have not had time to die. What results is that the collector promotes a large percentage of this region into the older generation during a GC, where most of these objects quickly die thereafter. This behavior is undesirable for a couple of reasons. (1) The more copying done by the collector, the longer the pause time caused by the collection. The copying of surviving objects is a significant contributing factor to a GC's run time. (2) The purpose of the older generation is compromised. The older generation is meant for surviving objects whose behavior suggests (by surviving the nursery collection) that they will survive for a nontrivial amount of time, i.e. they are not quick-dying objects used for essential, but short, functions of the program. However, these prematurely promoted objects clutter the older generation. Because objects quickly die after being copied, we introduce garbage in the older generation and waste space.

This work introduces the time-to-die nursery (TTD-Nursery) allocation algorithm for the fixed-sized nursery generational collector. Objects allocated within a certain amount of bytes of a collection are not scanned by the collector and instead become the start of the nursery upon completion of the collection. By refraining from collecting this region, the objects are not prematurely copied into the older generation, but rather are left in the nursery till the next collection, giving short-lived objects enough time to die. This region is aptly called the *time-to-die* (TTD) region. This division introduces more pointers that need to be tracked compared to the traditional fixed-sized nursery generational collector: in addition to the usual pointers from the old generation to the nursery, both the pointers from the TTD region to the main region and from the old generation to the TTD region require tracking as well.

For this work, we use the Jikes RVM and GCTk, a memory management toolkit for Java [1, 2]. We first run experiments to observe the survival rates of various portions of the nursery to get an idea of what to expect from the TTD-Nursery algorithm. We then quantify the pointer costs and determine whether the algorithm improves or degrades overall GC performance. The TTD-Nursery collector modestly improves the number of objects copied, but suffers in overall performance time due to the increase in remembered pointers.

The paper is constructed as follows. Section 2 discusses background and related work that are essential to the thought process used in the TTD-Nursery algorithm, such as the weak generational hypothesis, oldest-first, giving objects time to die, and various write barrier methods. The actual implementation details are discussed in Section 3, including the definition of the TTD region, the nursery organization in virtual memory, and the write barrier. Section 4 lays out the methodology behind implementing the TTD-Nursery algorithm, including survival rates for various portions of the nursery, in addition to the increased pointer-tracking tradeoff that is made to reduce the amount of copying. Sections 5 and 6 discuss the results and conclusions, respectively.

2 Background & Related Work

The TTD-Nursery algorithm is a combination of the basic generational collector organization with several key ideas from Stefanović et al.'s Oldest-First work [6]. First, we will briefly explain the fixed-sized nursery generational collector followed by the ideas borrowed from the oldest-first algorithm.

2.1 Fixed-sized Nursery Generational Collector

As mentioned before, this work uses a fixed-sized nursery generational collector. The fixed-sized heap is partitioned into three separate areas: a permanent region and two generations, old and young. The generational collector never collects the permanent region because data essential to the RVM and independent from the application program resides there. The two generations contain objects used by the program with the nursery (the young generation) containing all new memory allocations. When the nursery is full, the collector triggers a GC for the nursery and all surviving objects are copied into the old generation. Reserving enough heap space is essential for copying collection of the older generation (dividing the virtual memory reserved for the older generation into two separate regions, high and low, allows for the copying collector to work). A full heap collection triggers when a nursery collection is made and the old generation is found to be full as well. All surviving old objects are copied to the other half of the generation. The process repeats itself back and forth between halves.

2.2 Key Concepts of Copying GC and Oldest-First GC

We exploit certain techniques and behaviors in the TTD-Nursery algorithm. The weak generational hypothesis is inherent in any generational collector and both the time-to-die concept and the mechanism used for the write barrier are concepts that exploit advantages of an age-based ordered heap. This heap organization was first explored thoroughly by Stefanović et al [6].

Weak Generational Hypothesis. Most objects die young [5, 7]. This tendency is the cornerstone and motivation behind generational copying collectors. By dividing memory into age-based generations, the generational collector collects the young generation more frequently because of its large percentage of dead objects. This organization allows for quick, concentrated collections of the younger objects and fewer collections of the longer-lived older objects. An age-ordered heap organization helps gives us an idea of what an object's age is and allows us to take advantage of this concept.

Time to Die. While it is true that the majority of objects die fairly young, it is necessary for them to survive for at least some amount of time; every object is alive if checked immediately after allocation. Therefore, it is important to give an object enough time to die before exploiting the weak generational hypothesis in a GC. Stefanović et al. took advantage of this property in their Oldest-First collector [6]. By organizing the heap in order of age, objects are given time-to-die because collection "follows" allocation. Collection is restricted to a window that travels through the heap from older to younger objects. When it bumps into the very youngest objects, it resets the collection window to the oldest end of the heap. The GC collects the dead objects and copies the old objects behind the collection window. In a steady-state the collection window follows, yet is slightly behind allocation, giving objects enough time to die. However, this activity results in the repeated copying of old objects. We try to achieve the same performance boost in the traditional generational collector without the increased copying of old objects.

Write Barriers in an Age Ordered Heap. By organizing the heap by object age, both the write barrier and the remembered sets can be simplified. Because of this ordering, only pointers from an older object to a younger object need to be recorded; during a collection, the collector collects all objects in a generation so pointers going the opposite direction (young to old) do not need to be remembered. This property allows for a quick, simple first write barrier test: do not remember the pointer if the source is younger than the target. Pointers that pass this first test are subject to additional tests that also take advantage of the age-ordering of the heap (discussed in Section 3.3). The collector ignores intra-generational pointers because collections are made on the scope of a generation.

3 The TTD-Nursery Collector

The TTD-Nursery collector builds off the fixed-sized nursery generational copying collector. For the most part, the ideology behind the generational collector is kept intact with a few changes made to the nursery organization. We will define the TTD section, show how its behavior is supported by the virtual memory layout of the nursery, and finally define the write barrier.

3.1 The TTD Region

The TTD region is the youngest *ttd* bytes of memory in the nursery. Given a nursery size of N, the older N-*ttd* bytes are considered for collection and the remaining *ttd* bytes remain in the nursery, unscanned by the GC. This delay is the main idea behind the algorithm; by providing more time to die, the objects in the TTD region are not prematurely promoted to the older generation.

Determining where the separation between the main and TTD regions lies is done during allocation. When an allocation is made, the GC first makes sure there is room in the nursery for the object. This boundary check is done by testing if the allocation of the new object causes the nursery to be full. If so, a GC occurs. If the addition of the current object to the nursery is greater than the nursery size minus the TTD size (N-*ttd*), the allocator starts allocating new objects in the TTD region. Otherwise, allocation has not yet entered the TTD region and continues in the main region. The TTD region is differentiated from the main

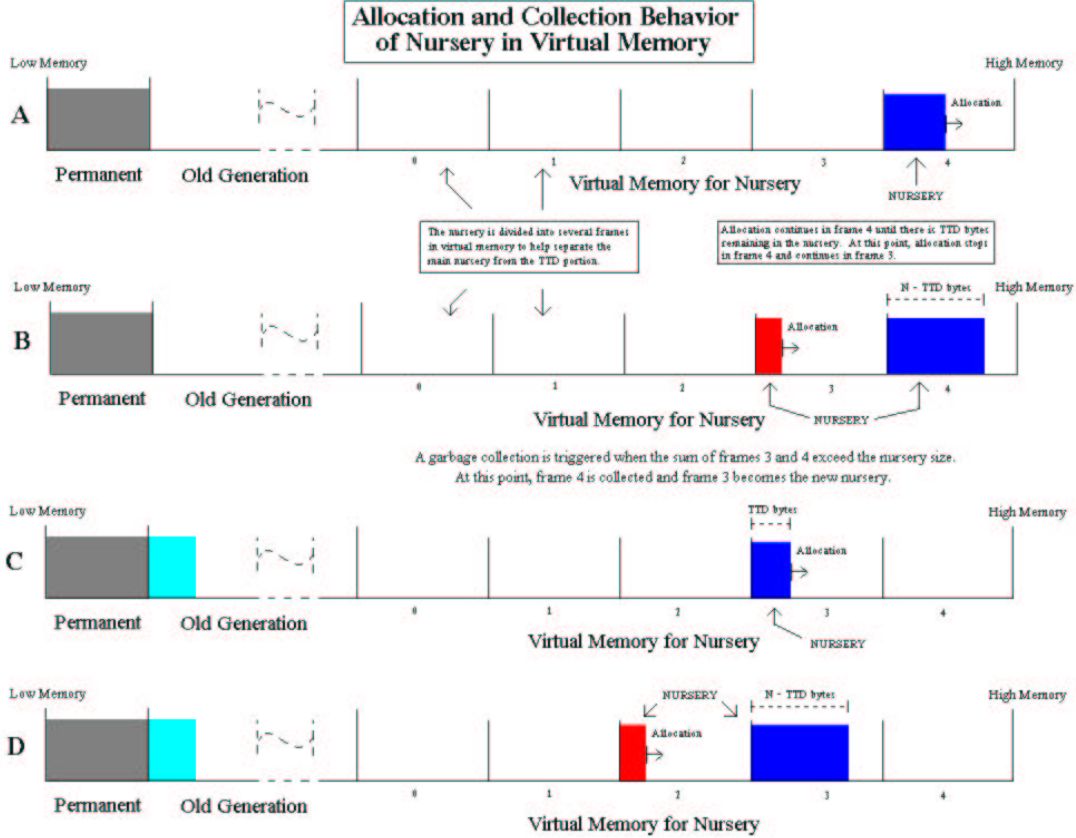


Figure 1: Nursery Layout in Virtual Memory

region by way of the organization of the nursery in virtual memory.

3.2 Nursery Organization in Virtual Memory

In order to separate the nursery into two distinct regions, certain alterations are made to its organization in virtual memory (Figure 1). Several smaller allocation frames are defined in memory for the nursery rather than one all-encompassing piece of memory. A *frame* is a division in virtual memory used to divide the nursery into multiple, small allocation areas and allows for more complicated nursery allocation methods. The large virtual space allotted for the nursery permits such a division. At the start of the RVM, the nursery is bound to the highest frame in memory. Allocation travels down virtual memory, moving from frame to frame at set triggering conditions.

When allocation enters the TTD region of the nursery, the GC releases the allocator from the current frame (F_i) and rebinds it to the next frame down the virtual memory (F_{i-1}). Allocation resumes in the new frame (F_{i-1}) and the nursery now spans two frames (F_i, F_{i-1}) in memory. At this point, the main and TTD regions are visibly separate, with the main being the old frame (F_i) containing N-*ttd* bytes and the TTD being the current frame (F_{i-1}) containing the remaining *ttd* bytes of the nursery. When the sum of the two regions equals or exceeds the nursery size, a GC is triggered for the main region. Collection proceeds as normal and upon completion, the remaining TTD region (F_{i-1}) becomes the new main region and allocation continues where it left off. This behavior continues throughout the virtual memory, with the next frame in line containing the TTD region of the nursery until a GC.

A special case is made for the last frame in virtual memory. Obviously, there is no next frame to put the TTD

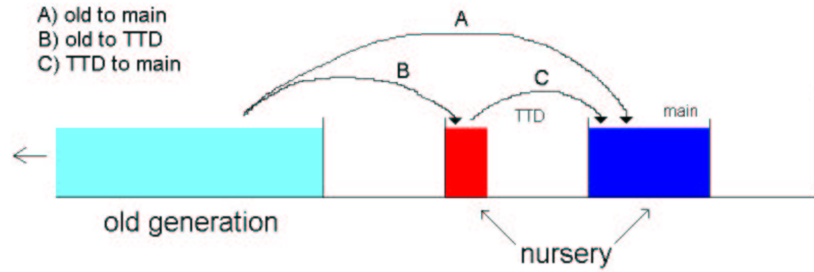


Figure 2: Remembered Pointers for TTD-Nursery

region, so the division of the nursery is simply ignored and a GC occurs for the entire nursery. Afterwards, the algorithm starts over in the first frame at high memory of virtual memory.

Collection of the older generation, which results in a full heap collection, also requires special attention. After a regular nursery collection, if the older generation is full, it is necessary to collect the TTD region prior to collection of the older generation. This full nursery collection is required because the TTD remembered set (Section 3.3) becomes invalid after an older generation collection. The source addresses are incorrect after the older generation's survivors are copied into to-space, rendering the TTD region tainted and unusable. To prevent this pointer invalidation, a separate GC of the TTD region is preemptively triggered before collecting the older generation.

3.3 Write Barriers and Remembered Sets

Figure 2 shows the three categories of pointers remembered in the TTD-Nursery algorithm. Because collection is limited to only a subset of the heap, pointers going into the collected region from the uncollected region must be remembered, including pointers from the TTD region to the main region of the nursery. These pointers are uniquely remembered for the TTD-Nursery collector and are normally ignored in the generational collector because they nursery is not divided; they are simply intra-generational pointers. If an object in an uncollected space points to a possible collection candidate, we assume the candidate is alive to assure correctness. During a GC, the collector traverses these remembered pointers in addition to those from the root set, which includes registers and stack contents, to determine all possible live objects.

The write barrier detects pointers into the nursery from the older generation, making a distinction between the main and TTD regions. Both regions require a remembered set because they are collected separately. When a GC collects the main region, it processes and releases the main remembered set, leaving the TTD remembered set intact. Just like the TTD region becomes the main region upon a GC completion, the TTD remembered set becomes the main remembered set. The TTD-Nursery algorithm never explicitly processes the TTD remembered set except in the case of a full heap collection (i.e. an older generation collection). The code for the write barrier follows. *HEAP_KPRIME* is the logarithmic size of a frame of virtual memory for the nursery (see Section 3.2). *mainBlock_* is the address of the start of the main region.

```
public static final void writeBarrier(ADDRESS source, ADDRESS target) {
    if (source < ((target>>>HEAP_KPRIME)<<HEAP_KPRIME)) {
        wbHB(source, target);
    }
}

public static final void wbHB(ADDRESS source, ADDRESS target) {
    if (target > mainBlock_)
        GCTk_RememberedSet.insert(intoMainRemset_, source);
    else
```

```
        GCTk_RememberedSet.insert(intoTtdRemset_, source);
}
```

We divide the write barrier into two parts: a fast and slow path. The fast path is an inlined test that involves a simple comparison to see if the source address resides in an older frame than the target address. We cannot simply check to see if the source is in the older generation and the target is in the nursery because the TTD region must be taken into account. Pointers from the TTD region to the main region of the nursery must also be remembered. This test fails for the majority of pointers [6] and because it is inlined, results in the fast completion of most write barrier checks. If the source is in an older frame than the target, the slow path determines in which remembered set the pointer is stored, this is done by determining where the target address resides. If the target is in the main region, then the pointer should be stored in the main remembered set; it makes no difference if the source is in either the older generation or the TTD region. Otherwise the target is in the TTD region and should be stored in the TTD remembered set.

4 Methodology

The nursery of a generational collector exhibits certain behaviors that motivate exploration of the use of a TTD region. This section briefly describes the Jikes RVM and GCTk used to implement the TTD-Nursery collector. We also describe the experimental setting used to run experiments and gather data. In addition, this section addresses the survival rate of objects in various areas of the nursery, specifically the youngest and oldest regions. Finally, we look at the pointer tracking cost that can be expected in the TTD-Nursery algorithm and discuss the implications on performance.

4.1 Jikes RVM and GCTk

We implemented this work in the Jikes RVM. Written in Java, the Jikes RVM is a high performance VM with an aggressive optimizing compiler [1]. We use the *optimizing* compiler and the *fast* build-time configuration. The optimizing compiler compiles all methods required for the execution of each benchmark and the fast build-time configuration removes assertion checking and pre-compiles as much as possible into the Jikes RVM boot image. We also use the GCTk, a garbage collector toolkit for the Jikes RVM [2]. The GCTk contains many popular garbage collectors and takes advantage of object oriented techniques found in Java. The TTD-Nursery collector extends the generation collector found in the GCTk to achieve the desired behavior.

4.2 Experimental Setting

We perform all of our experiments on 2 GHz Pentium 4, with 8KB L1 data cache, a 12K L1 instruction cache, a 512KB unified L2 on-chip cache, and 1GB of memory running Linux 2.4.18. We run each benchmark at a particular parameter setting five times and use the fastest of these. The variation between runs is low, and we believe this number is the least disturbed by other system factors.

4.3 Survival Rates of the Nursery

Figures 7 - 10 show the survival rates of the youngest, oldest, and remaining regions of the nursery using six SPEC Java programs. The x-axis spans several TTD region sizes and the y-axis represents the amount of living objects as a ratio of all objects in the entire nursery. In addition to raw survival rate numbers, Figures 3 - 6 represent the percentage of surviving bytes for a given region, with the x-axis again spanning TTD region sizes and the y-axis representing the percentage of survivals of the region. For instance, Figure 3(a) shows that 12.4% of the youngest 32KB bytes of the nursery survive when run with the _202_jess benchmark. Percentages for the other regions are calculated the same way. Figure 11 shows the differences in percentages of survival rates between the young TTD and old TTD regions. In order to get an idea of how large objects affect survival rate, we separate larger objects from smaller objects by directly allocating objects of a certain

size (1KB, 16KB, and 256KB) into the older generation, completely removing their presence from the nursery. This separation allows us to directly observe the survival rate of the smaller objects more likely to die shortly after allocation. Many collectors employ a separate large object space [3].

The youngest TTD bytes of the nursery consistently contain more survivors per byte than either the oldest TTD bytes or the remainder of the nursery. Compared directly to the oldest TTD bytes (Figure 11), we can observe the impact made by giving objects time to die. Depending on the benchmark and TTD size, there is a percent difference of 1% to 12% between the youngest and oldest TTD bytes. The TTD-Nursery collector abstains from collecting the youngest TTD bytes of the nursery, leaving it for the next GC, where it becomes the oldest TTD bytes. This difference suggests using a TTD region during collection will reduce the amount of objects copied.

For the majority of benchmarks and different large object promotion policies, there is a clear decreasing function for the percentage of survivors in both the youngest and oldest TTD regions. This behavior is most likely on account of two factors. First, the larger the TTD size, the more time objects are given to die; evidence of the weak generational hypothesis within the TTD regions themselves. Second, to maintain a high survivor percentage as TTD size increases would require a significant amount of surviving objects. It only takes the survival of 3K or 6K bytes to have a survival rate of 10% in a TTD region of 32KB or 64KB, whereas it would take the survival of 50KB or 100KB bytes for a 10% survival rate in a 512KB-sized or 1024KB-sized TTD region. Raw survival rates need to be taken in account to explain the decreasing surviving percentage due to increasing TTD size. Despite the decrease in survival percentage as TTD size grows, there still exists a gap of 1% to 6% more surviving bytes in the youngest TTD bytes, even in the case of the largest TTD size of 1024KB (Figure 11).

A look at the raw survival rates Figures 7 - 10 is also necessary to determine the likely performance impact of the TTD-Nursery collector. While the smaller TTD sizes result in the largest difference in survival percentages, they also represent the smallest amount of surviving bytes. Allowing 3KB worth of surviving bytes to shrink down to 2KB in a 32KB TTD region before copying them to the older generation will not produce much of an improvement in run time or the *mark/cons ratio* (the ratio of the amount copied to the amount freed by the GC) [4]. So while having a 10% survival rate in a smaller TTD region looks attractive, it may not necessarily reflect much of an improvement.

4.4 Pointer Tracking Cost

Figures 12 and 13 illustrate the distribution of pointers that the TTD-Nursery collector tracks, as discussed in Section 3.3. Both graphs have several TTD sizes for the x-axis. The y-axis of Figure 12 shows the ratio of the number of pointers for a given region versus all remembered pointers and the y-axis of Figure 13 is the number of pointers remembered.

As the TTD region grows in size, so too does the pointers remembered from TTD to main. For every benchmark, there is a consistent and clear increasing function as the TTD size increases in the percentage of remembered pointers coming from the TTD region; pointers that are not remembered in a normal generational collection. This behavior gives us an idea just how many intra-generational pointers exist within the nursery. For _202_jess, _205_raytrace, and _209_db, the presence of intra-generational is quite prevalent, with the TTD to Main pointers constituting nearly 50% of remembered pointers with a TTD size of only 256KB or 384KB. _202_jess reaches 50% with only a TTD size of 128KB. This trend continues with nearly 85% of remembered pointers in _202_jess originating in the TTD region and pointing to the main region with TTD sizes of 896KB and 1024KB (Figure 12(a)). Similar high percentages (nearly 70%) are also found with _205_raytrace and _209_db as TTD sizes increase as well (Figures 12(b) and 12(c)). However, other benchmarks do not retain such high percentages. _228_jack and, in particular, pseudojbb never have a majority of their remembered pointers originating from the TTD region, with pseudojbb never rising above 20% (Figures 12(e) and 12(f)). Pointer activity obviously depends greatly on the program.

Overall, the introduction of the new class of rememebered pointers results in a large increase in remembered pointers. The TTD to main pointers result in between a 24% (pseudojbb) to 604% (_202_jess) increase in pointers remembered.

Pointers from the old generation into the TTD region represent only a small percentage of the total remembered pointers, numbering only in the few hundreds. They are not processed during a TTD-Nursery GC (they are processed when the TTD region becomes the main region), so their influence on performance is trivial.

5 Results

This section looks at both the time spent in garbage collection and the mark/cons ratio for several different TTD sizes of the TTD-Nursery collector. Performance of the algorithm is compared to the performance of a generational collector, represented by the TTD-Nursery collector with a TTD size of zero. A TTD region with a size of zero does not split the nursery into two regions and behavior mirrors that of a generational collector. We then examine what role the survival rate and remembered set behaviors have in TTD-Nursery execution. All graphs have the heap size in a log scale on the x-axis, absolute values on the top of the graph and relative to minimum heap size on the bottom. Figure 14 shows the mark/cons ratio on the right y-axis and the relative values on the left y-axis. Figures 15 and 16 both show time on the y-axis; absolute seconds on right side and relative time on the left.

The TTD-Nursery collectors with sizes of 32KB and 64KB are omitted from this analysis. Their performance nearly mirrored that of the TTD-Nursery with a TTD size of 0 and proved trivial in their effectiveness. This ineffectiveness is most likely due to the fact that memory in the Jikes RVM and GCTk is handled as blocks whose size is 128KB. This division makes it difficult for TTD sizes of finer granularity than a block to function accurately because the method used to detect when the GC enters the TTD region operates in blocks rather than individual bytes. In addition, as mentioned in Section 4.3, the actual amount of surviving bytes in the smaller TTD sizes is relatively small and reflects little difference in overall performance.

5.1 Mark/Cons Ratio

Each benchmark experiences an improvement in their mark/cons ratios as a result of the TTD-Nursery collector, particularly for larger heap sizes, mirroring the measurements in 4.3. The benchmarks that exhibit the greatest percentage difference in survival rates between the oldest and youngest TTD bytes are _213_javac, _228_jack, and pseudojbb (Figures 14(d)(e) and (f)). When we exclude the large objects, the results improve further. The remaining benchmarks, _202_jess, _205_raytrace, and _209_db, also enjoy an improvement in mark/cons ratio, although the improvement is usually restricted to the larger heap sizes, above 1.5x the minimum. The previous benchmarks enjoy a more universal improvement over all heap sizes.

Particular TTD sizes do not seem to make much of a difference with no TTD size separating itself from the pack; instead it seems to vary greatly depending on heap size. However, the introduction of a TTD region to allow the youngest objects a chance to die does seem to have a positive, albeit small, impact on overall mark/cons ratio.

5.2 GC Time

Figure 15 shows the time spent in garbage collection for each of the benchmarks. The improvements in mark/cons ratio are negated in the end by the time spent processing the extra remembered pointers introduced by the presence of a TTD region. _202_jess and _205_raytrace, whose TTD to main pointers constitute, as TTD size increases, the majority of remembered pointers, clearly exhibit this property with the larger TTD sized configurations almost universally taking more time to execute than the smaller TTD sized configurations (Figures 15(a) and (b)). Conversely, the benchmarks that possess lower percentages of remembered TTD to main pointers, _228_jack and pseudojbb, demonstrate the best overall performance improvement with nearly all TTD configurations running faster than with no TTD region present (Figures 15(e) and (f)). Rarely do we see any time improvement in smaller heap sizes because of the increase number of full heap collections.

5.3 Total Time

Figure 16 shows the total time spent running the benchmarks. While different times are recorded for the overall runtime, the shape and behavior of each graph mirrors the GC time graphs, respective to the benchmarks. Again, the time spent processing the additional pointers introduced by the presence of a TTD region in the nursery hinders overall performance. The larger TTD sizes (red plots in Figure 16) almost universally take longer to run than smaller TTD sized configurations. This dichotomy is expected given the remembered pointers distribution disparity between the larger and smaller TTD sizes (Figures 12 and 13).

The time saved by reducing the number of objects copied into the old generation is offset by the increased pointer processing necessary because of the TTD region during a GC. This performance hit can possibly be avoided by preemptively triggering a GC when the TTD to main remembered set reaches a certain size. However, this idea lies outside the scope of this work.

6 Conclusions

Giving the youngest bytes in the nursery time to die slightly reduces the amount of copying done for the generational collector. However, the additional pointer tracking proves to be too big of a cost to improve overall pause time on average. As mentioned before, this cost could possibly be improved upon by preemptively triggering a garbage collection when the number of TTD to main remembered pointers exceeds a given threshold. However, this preemptive trigger may result in too many GCs to provide any good. In addition, the TTD size could be limited to be no larger than a specific size, as the number of TTD to main pointers increase with TTD size.

7 Acknowledgements

Personally, I would like to thank Kathryn McKinley for having the confidence in my abilities and taking me under her wing for my first venture into the realm of research. Her guidance and encouragement is greatly appreciated. Thanks also goes out to Steve Blackburn for his technical expertise, suggestions, bug-hunting skills, and overall camaraderie. I would not have made it past week three if it was not for the patience, understanding, kindness, and willingness to help of Maria Jump. Her advice and ability to make even the most daunting tasks seem doable has been invaluable to me throughout this endeavor. I most certainly would not have finished without her. Finally, thanks and appreciation go to both family and close friends for their patience with me throughout this entire endeavor. Thank you all from the bottom of my heart.

References

[1] B. Alpern, C. R. Attanasio, J. J. Barton, 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. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The jalapeno virtual machine. *IBM System Journal*, 39(1):211–238, February 2000.

[2] Stephen M Blackburn, Richard Jones, Kathryn S McKinley, and J Eliot B Moss. Beltway: Getting around garbage collection gridlock. In *Proceeding of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 153–164. ACM Press, 2002.

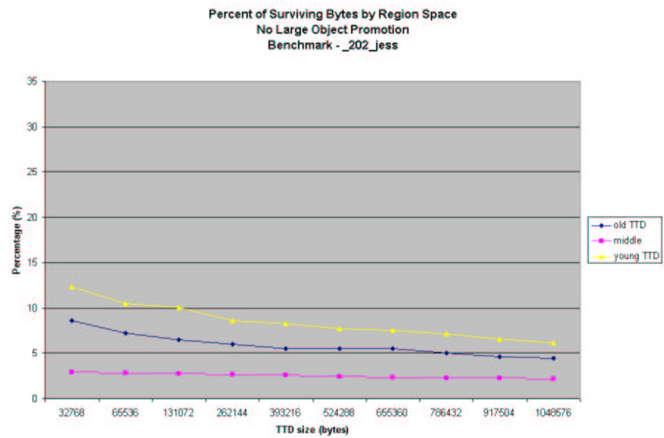
[3] M. Hicks, L. Hornof, J.T. Moore, and S. Nettles. A study of large object spaces. In *ISMM'98 Proceedings of the First International Symposium on Memory Management*, pages 138–145. ACM, 1998.

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

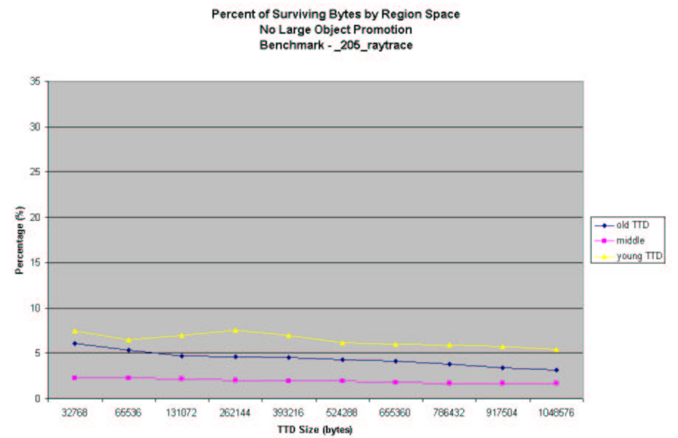
[5] H. Liberman and C. Hewitt. A Real-Time Garbage Collector Based on the Lifetimes of Objects. *Communications of the ACM*, 26(6):419–429, June 1983.

[6] Darko Stefanović, Kathryn S McKinley, and J. Eliot B. Moss. Age-based garbage collection. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, OOPSLA '99*, volume 34, pages 370–381. ACM Press, October 1999.

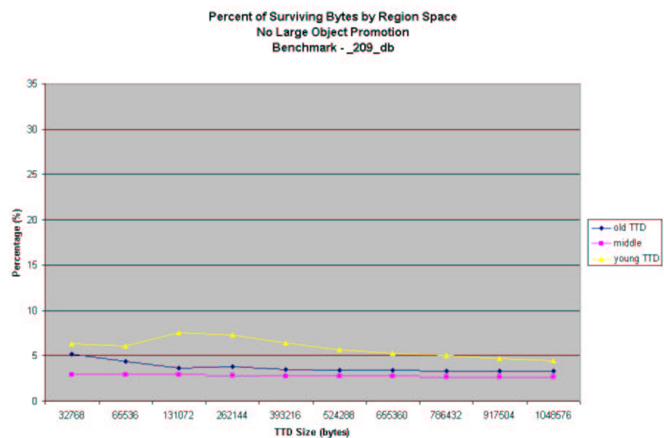
[7] D.M. Ungar. Generation scavenging: A non-distruptive high performance storage reclamation algorithm. volume 19, pages 157–167, April 1999.



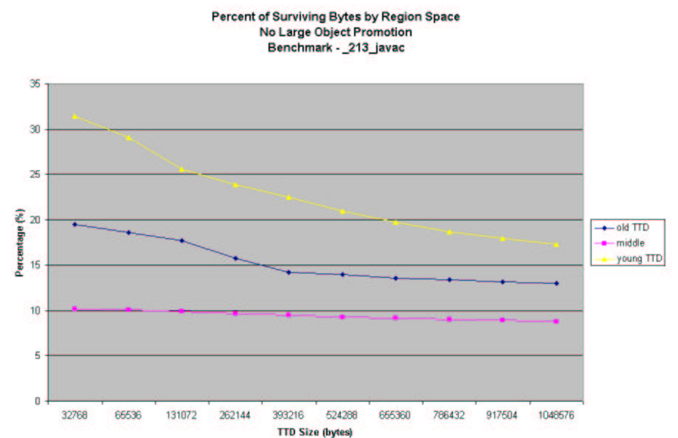
(a)



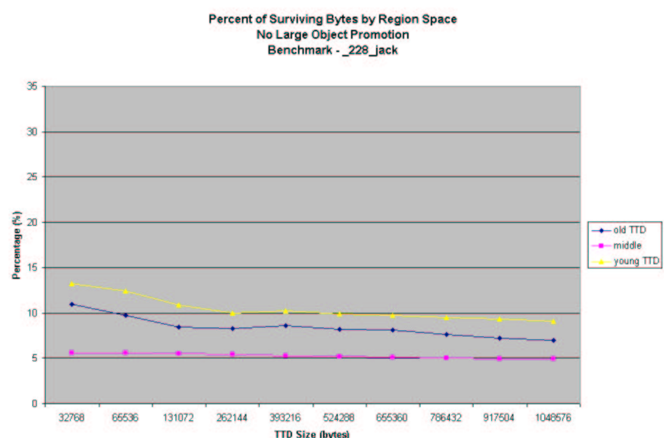
(b)



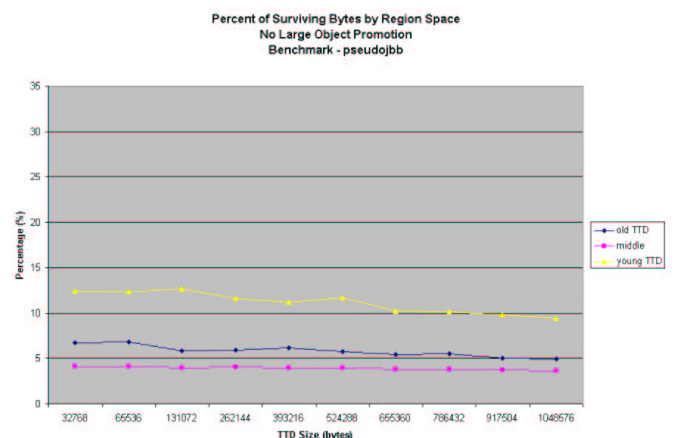
(c)



(d)

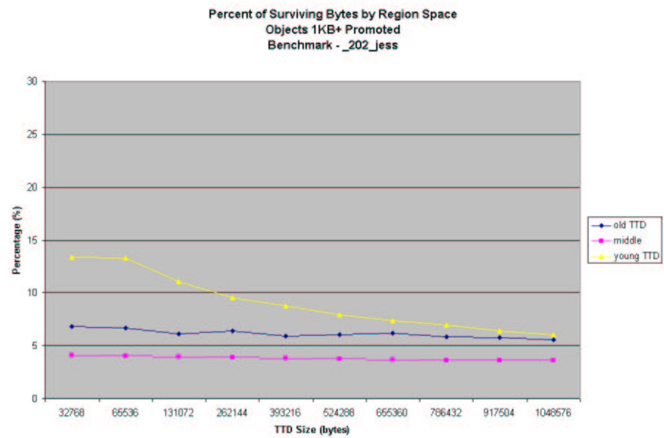


(e)

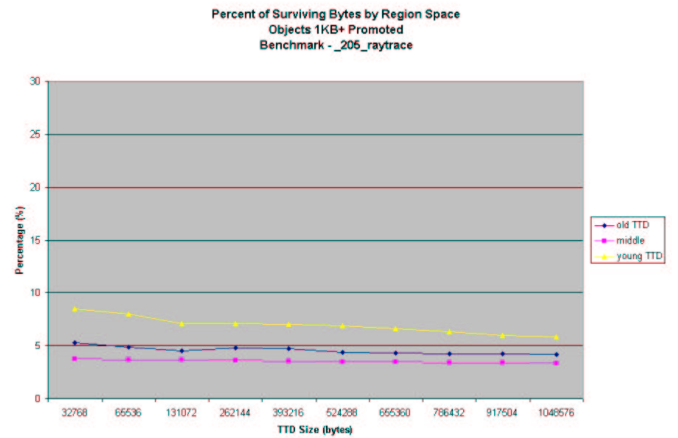


(f)

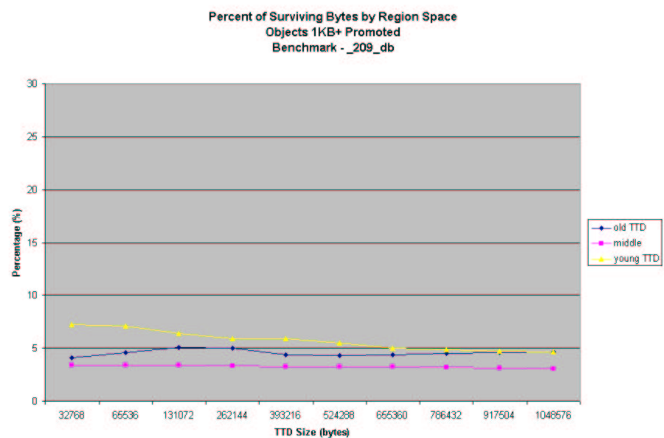
Figure 3: Survival Rate Percentages - No Large Object Promotion



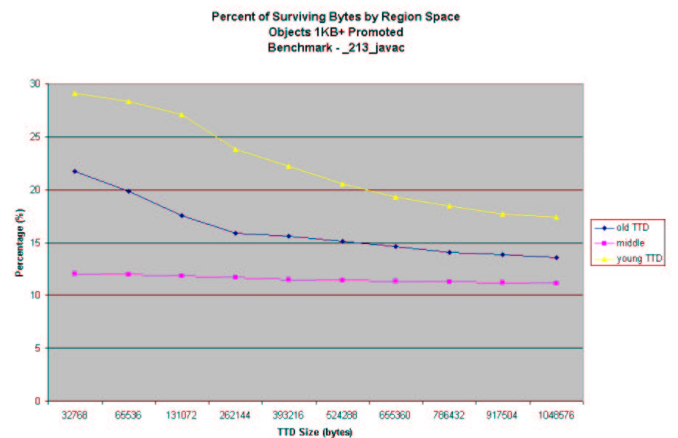
(a)



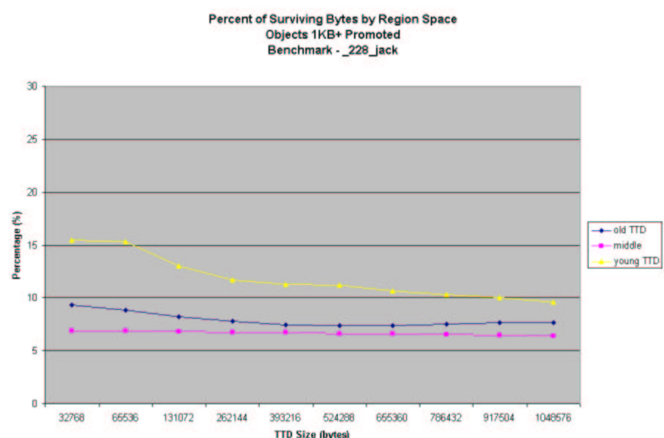
(b)



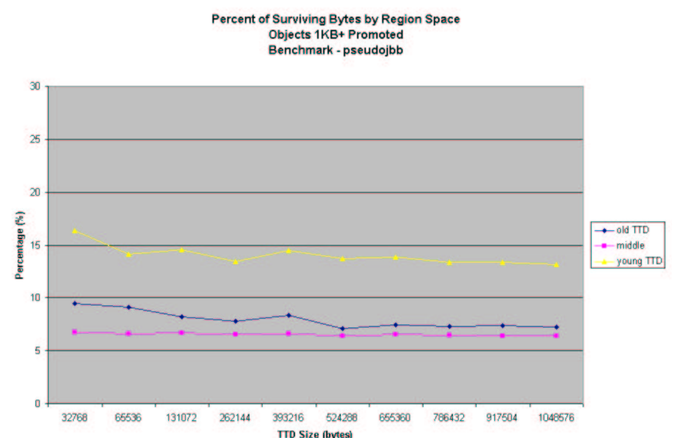
(c)



(d)

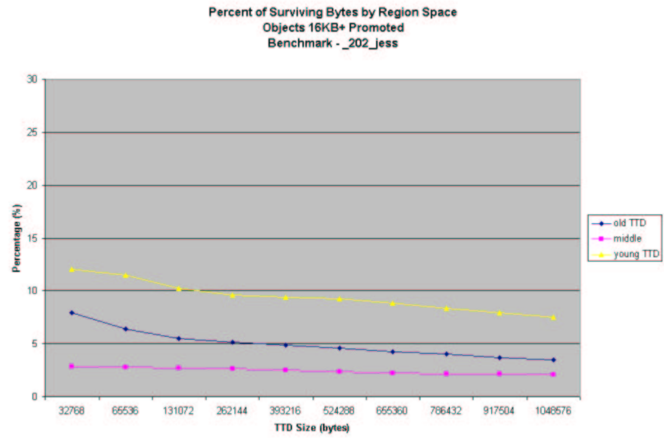


(e)

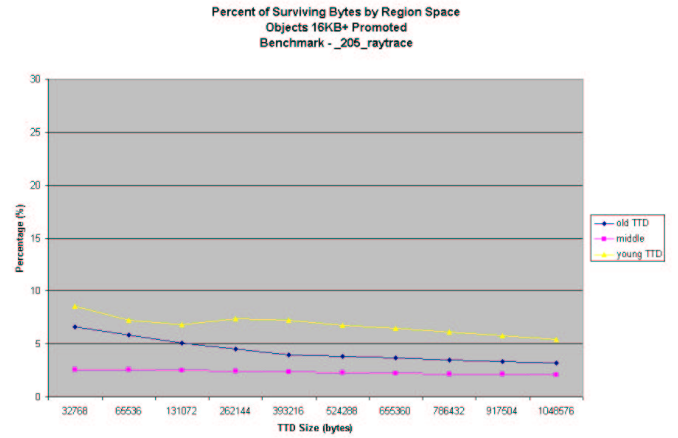


(f)

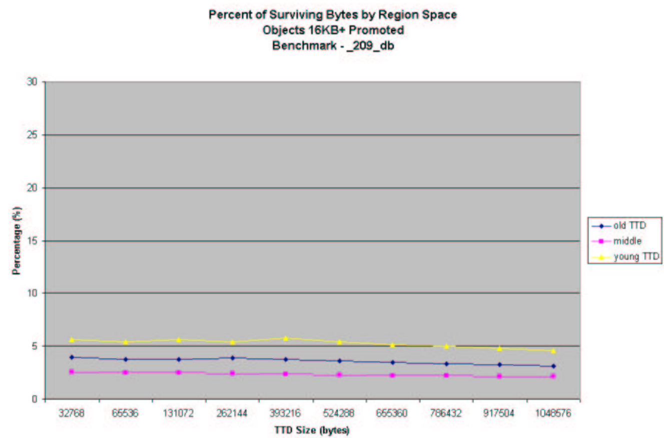
Figure 4: Survival Rate Percentages - 1KB+ Objects Promoted



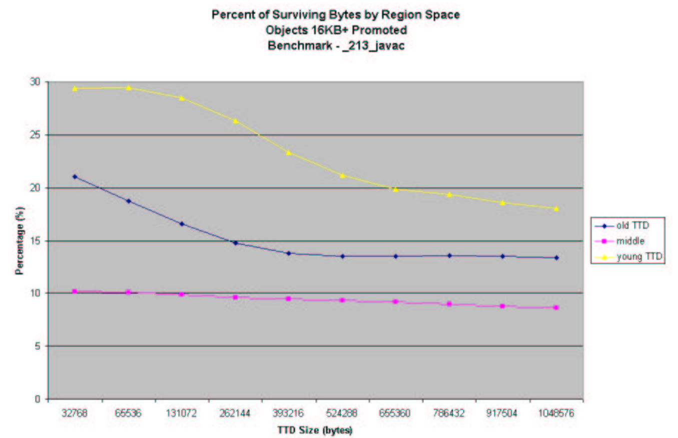
(a)



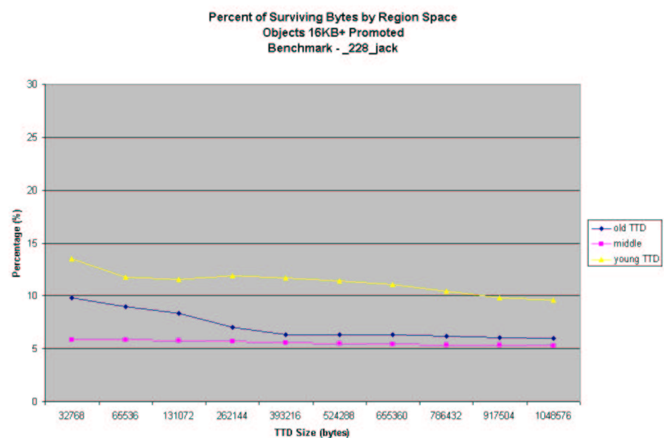
(b)



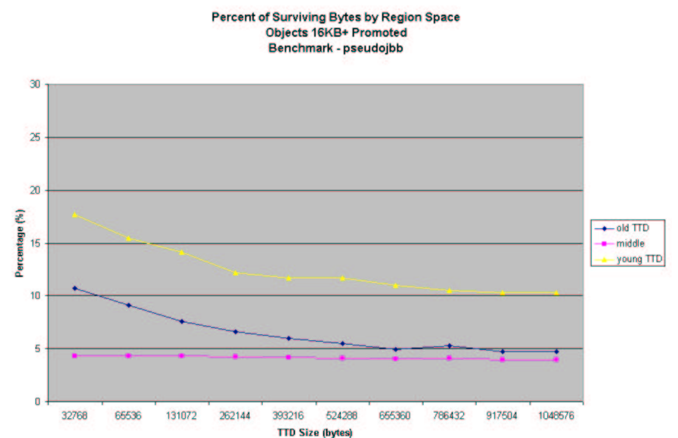
(c)



(d)

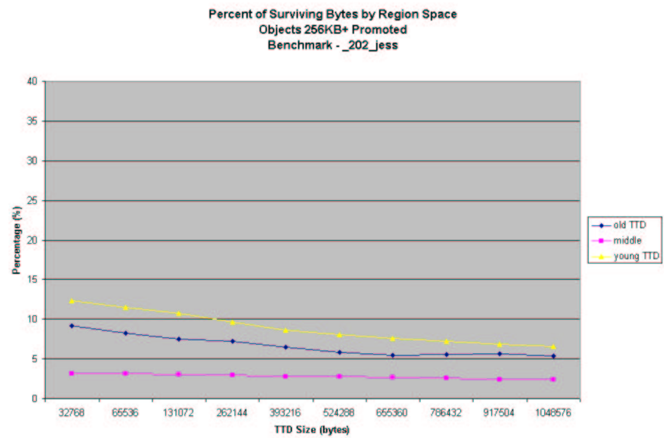


(e)

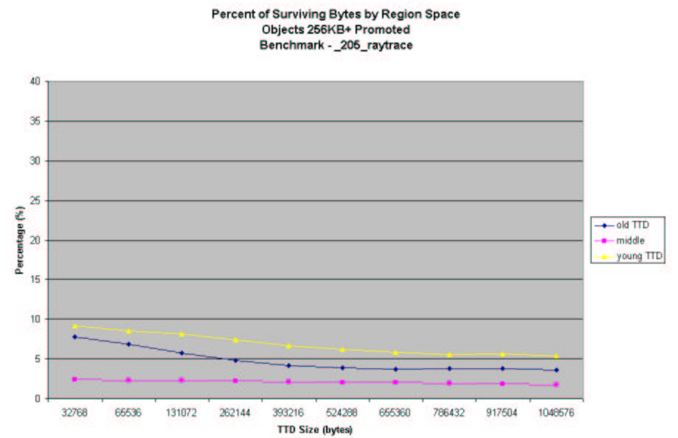


(f)

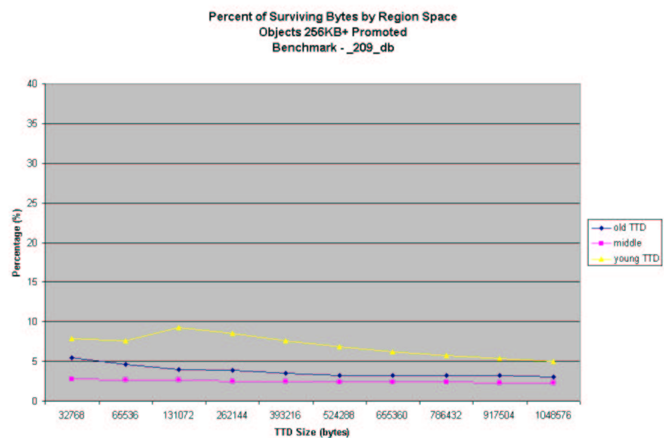
Figure 5: Survival Rate Percentages - 16KB+ Objects Promoted



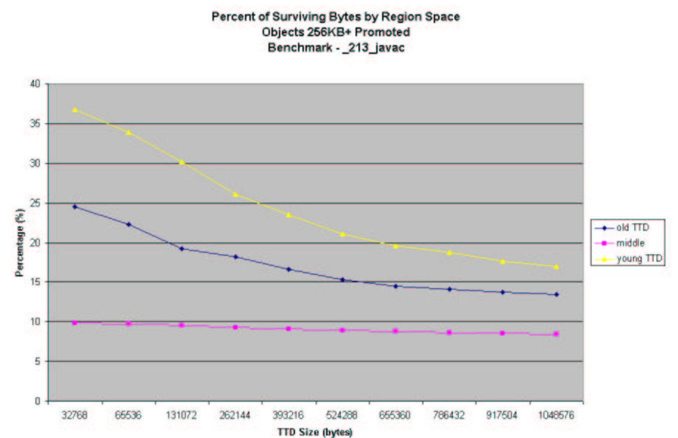
(a)



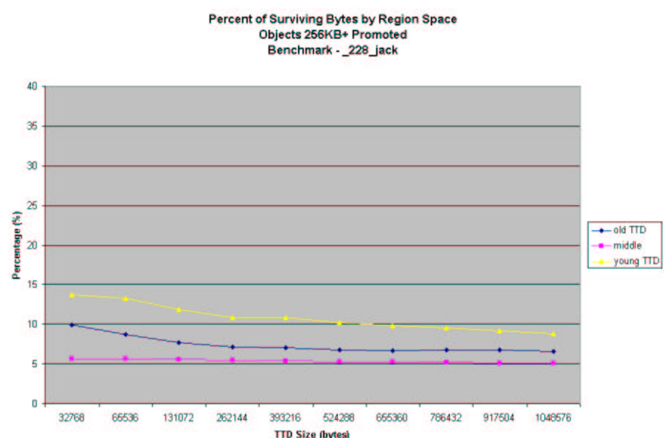
(b)



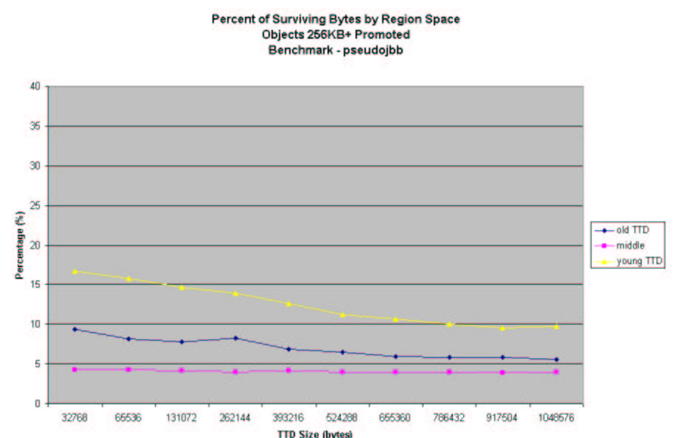
(c)



(d)

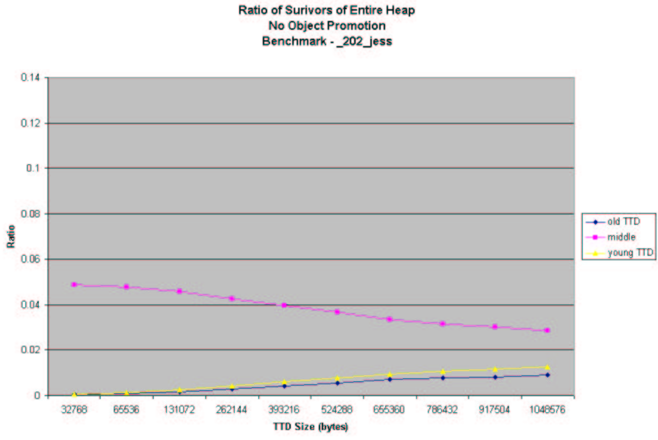


(e)

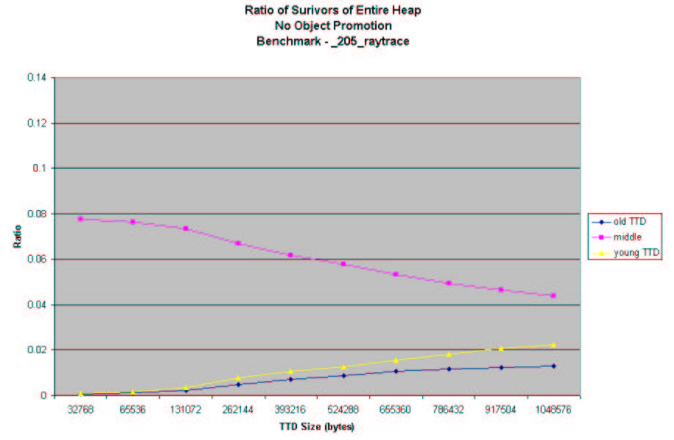


(f)

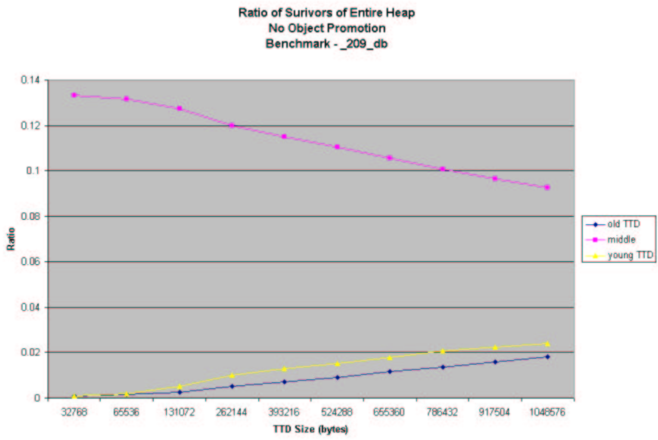
Figure 6: Survival Rate Percentages - 256KB+ Objects Promoted



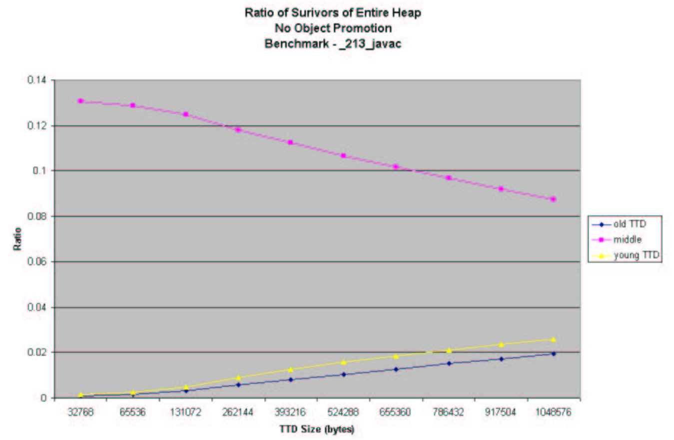
(a)



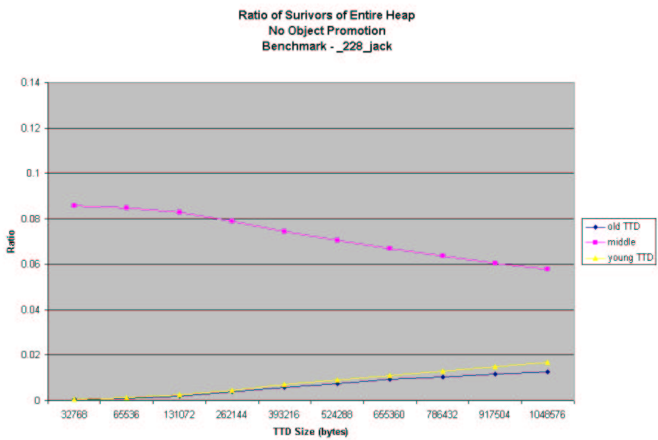
(b)



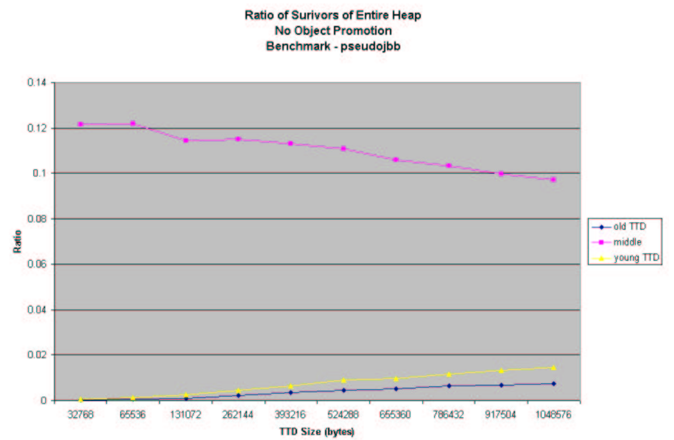
(c)



(d)

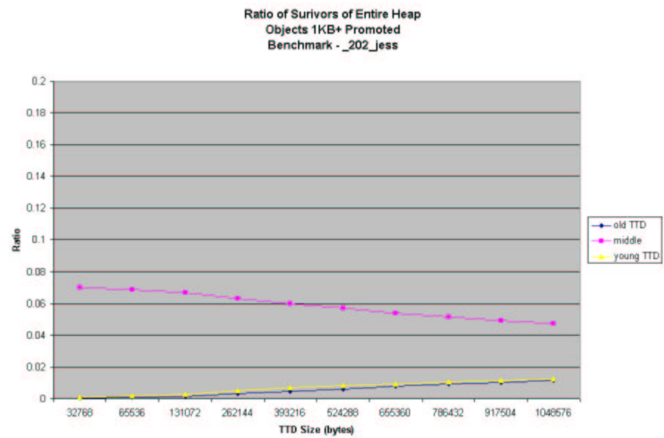


(e)

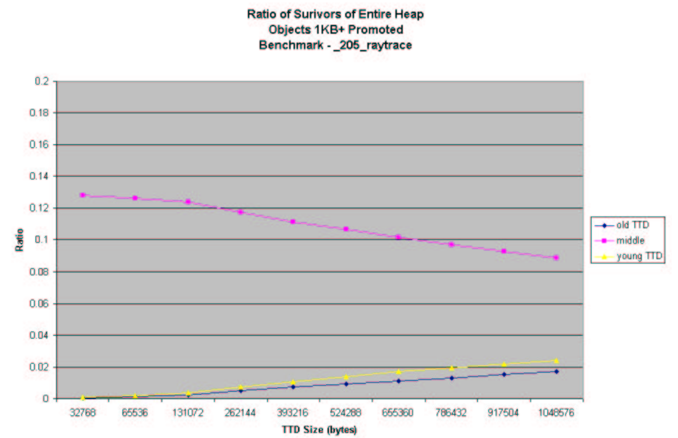


(f)

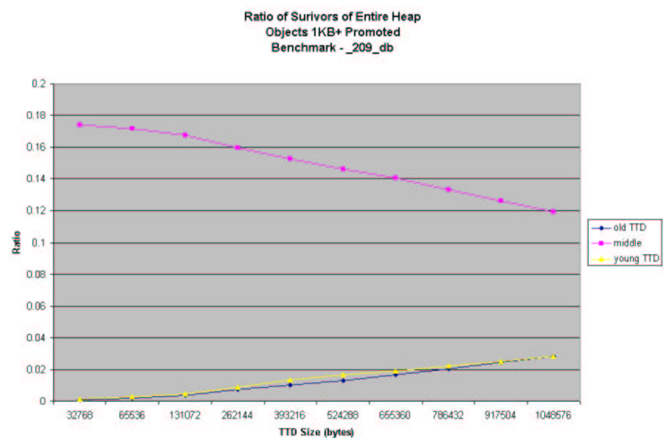
Figure 7: Survival Rate Raw Numbers - No Large Object Promotion



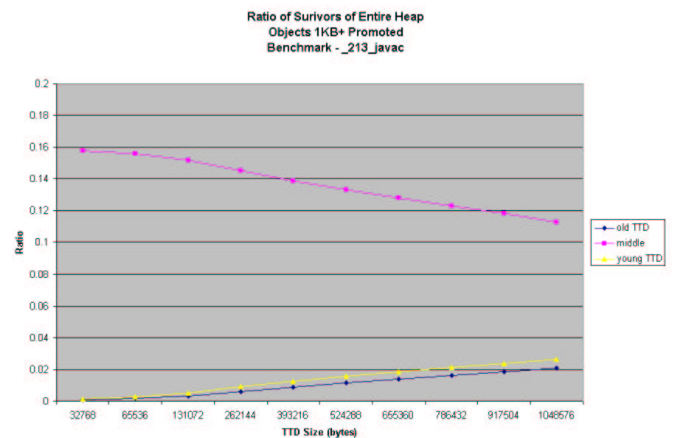
(a)



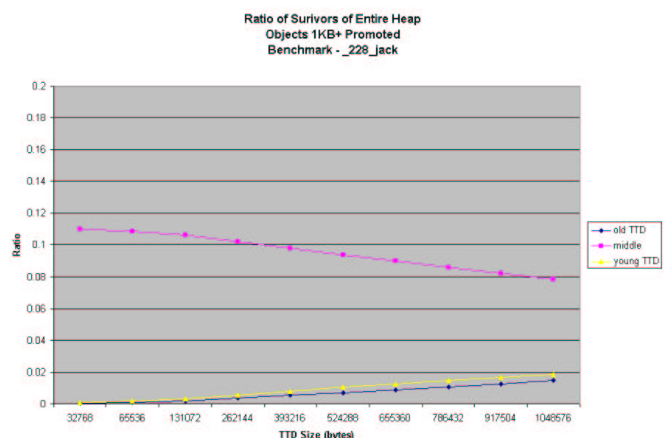
(b)



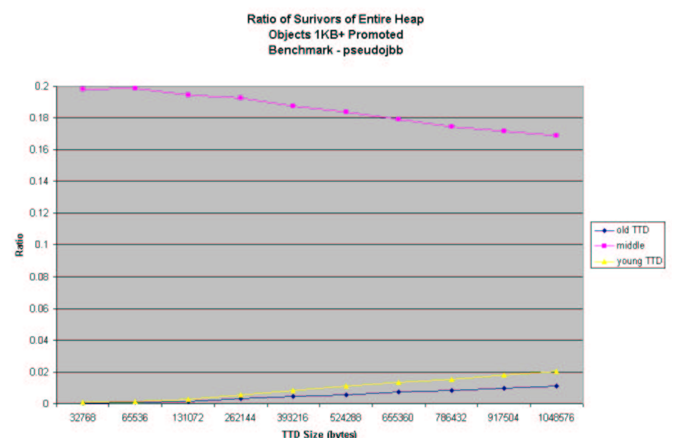
(c)



(d)

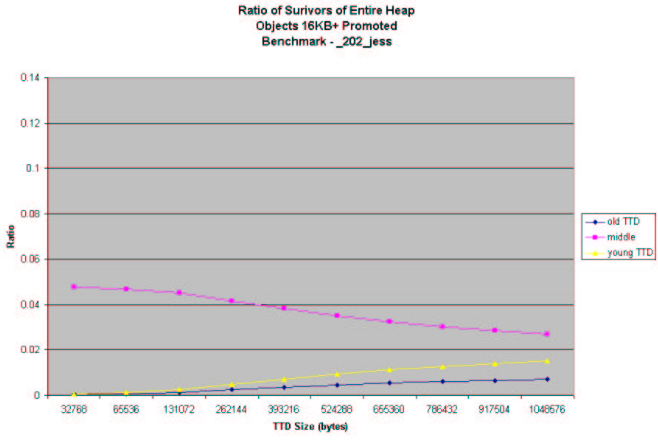


(e)

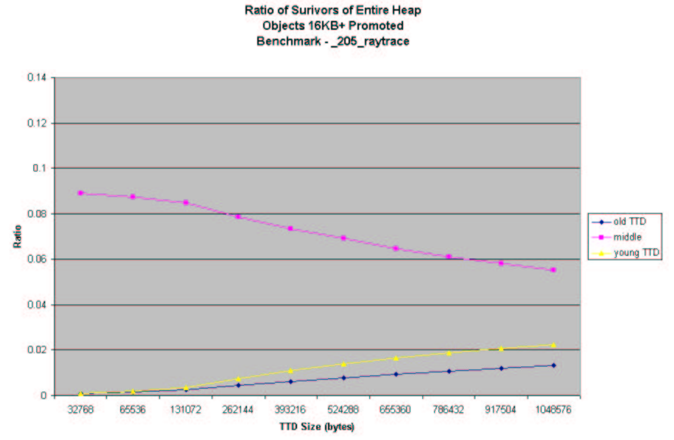


(f)

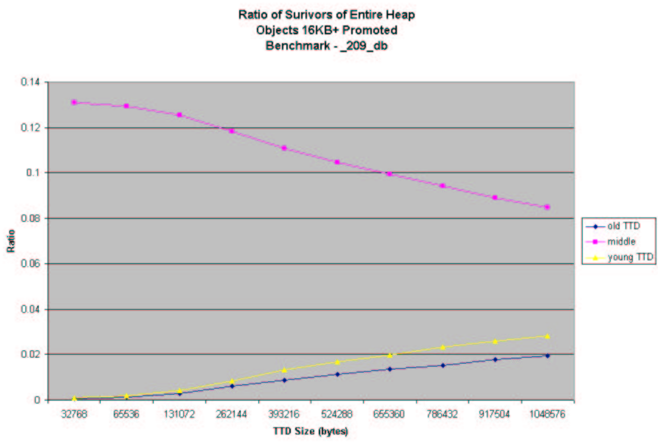
Figure 8: Survival Rate Raw Numbers - 1KB+ Objects Promoted



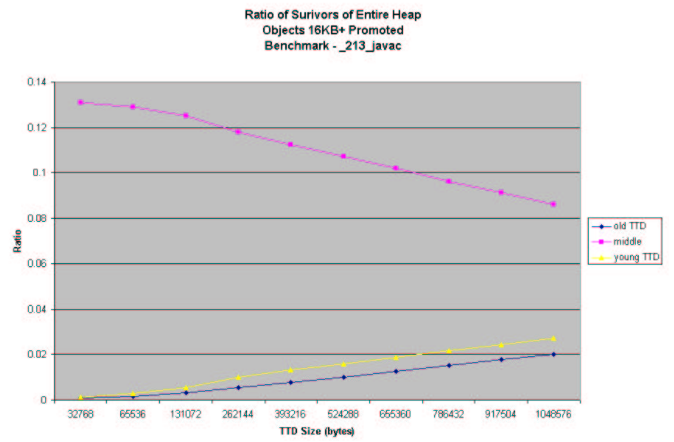
(a)



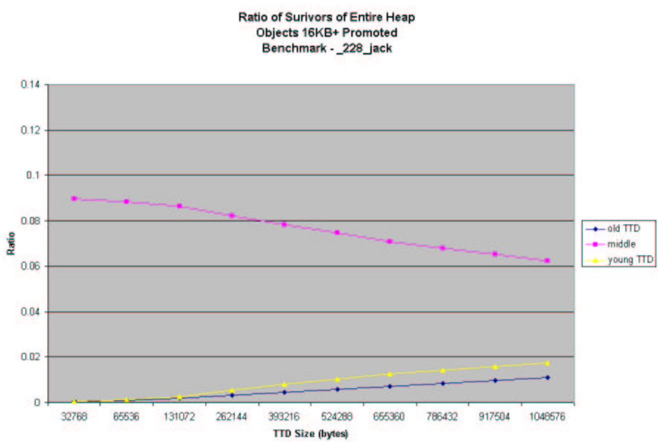
(b)



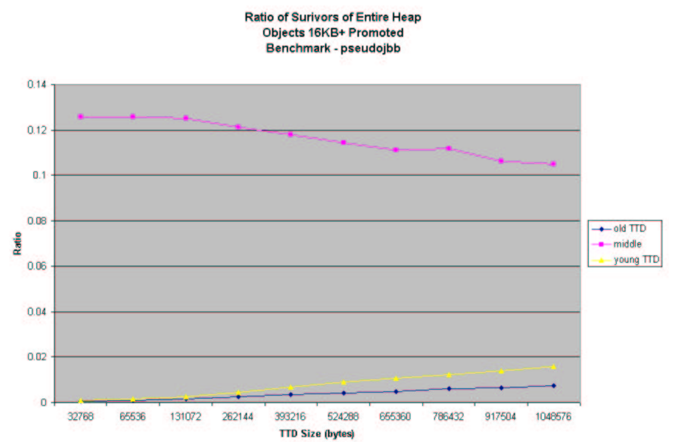
(c)



(d)

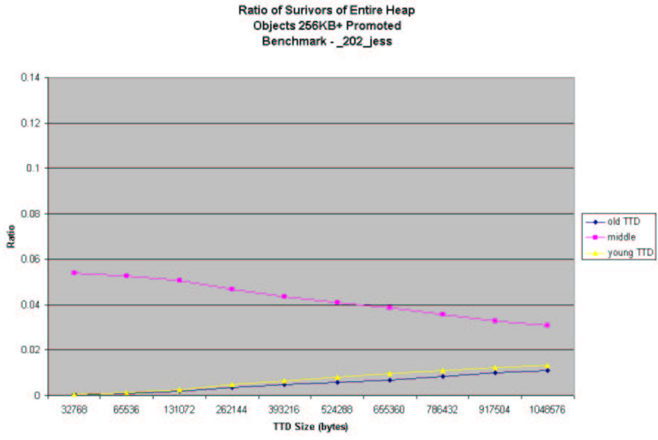


(e)

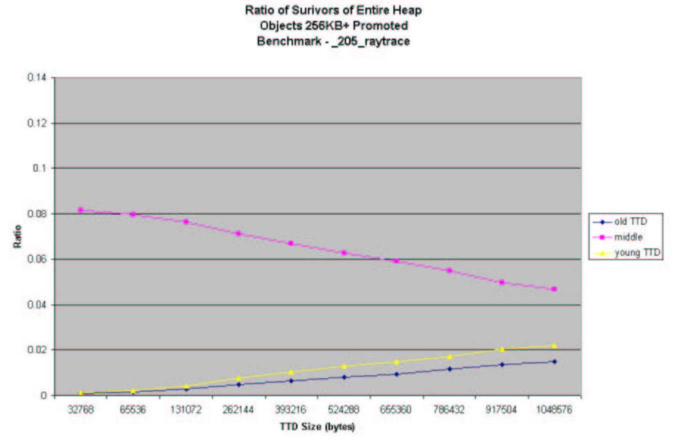


(f)

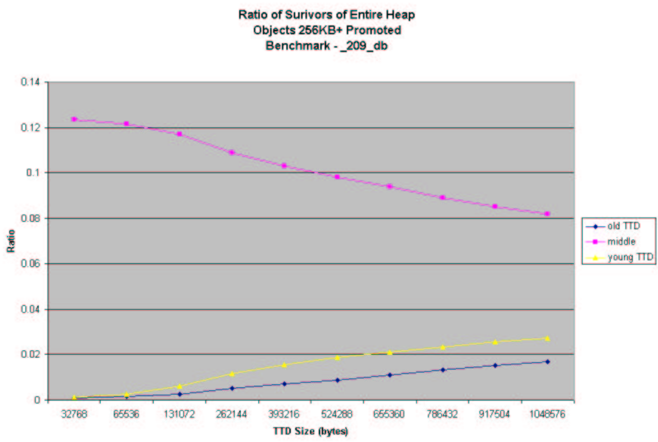
Figure 9: Survival Rate Raw Numbers - 16KB+ Objects Promoted



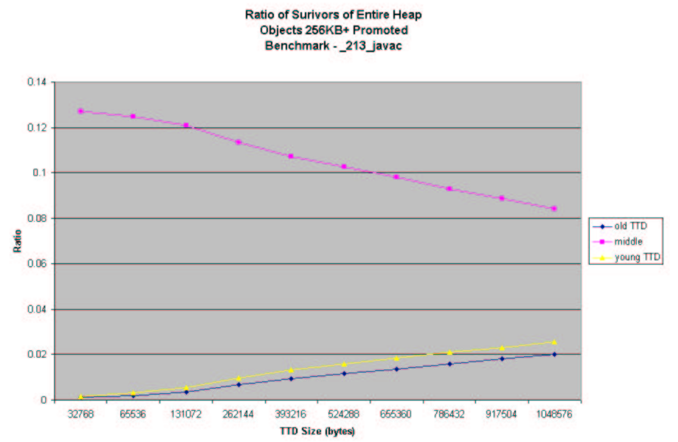
(a)



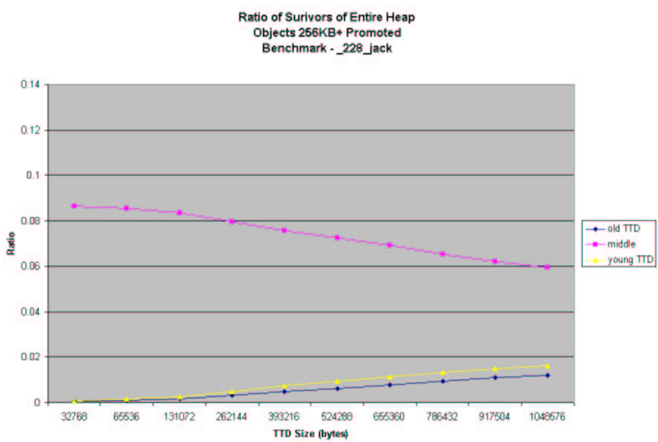
(b)



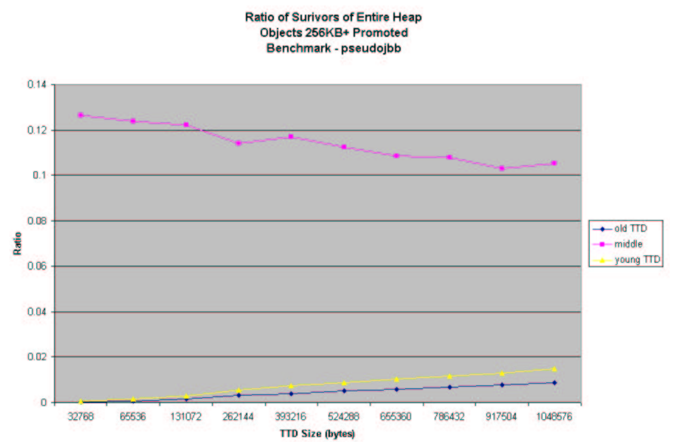
(c)



(d)

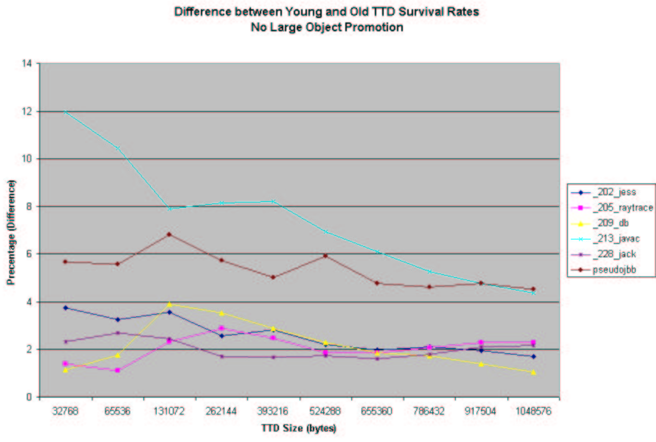


(e)

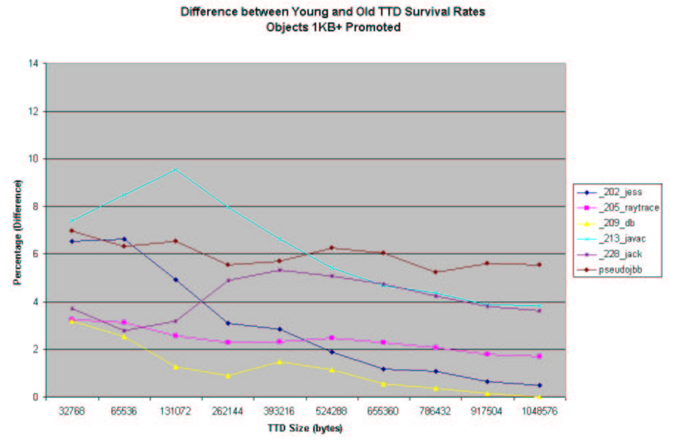


(f)

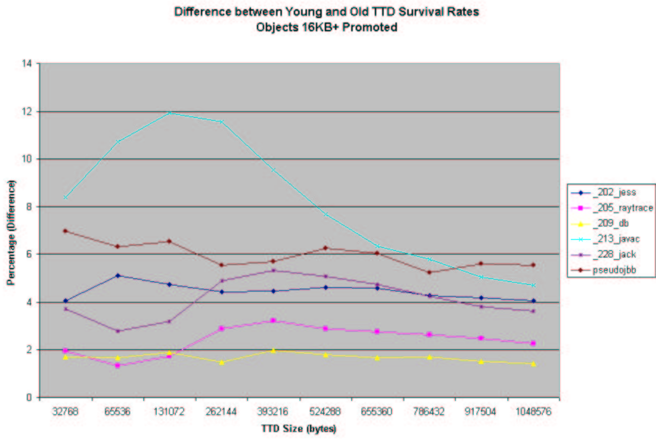
Figure 10: Survival Rate Raw Numbers - 256KB+ Objects Promoted



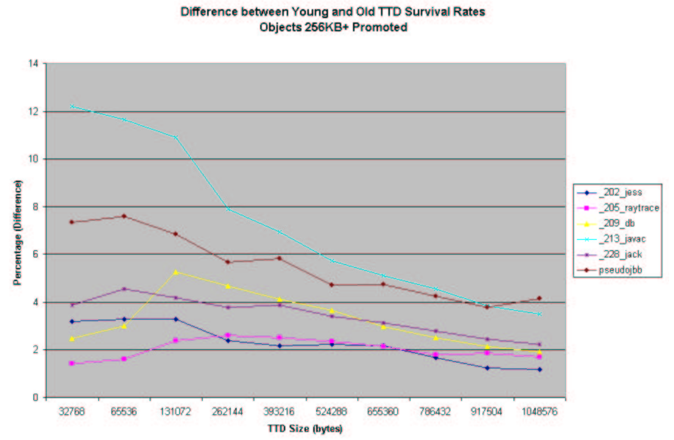
(a)



(b)

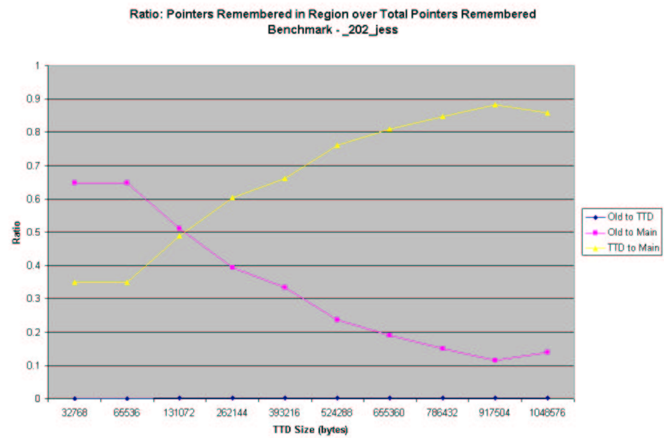


(c)

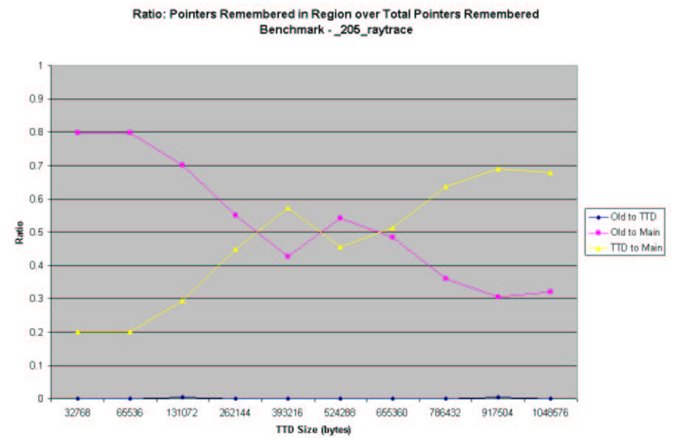


(d)

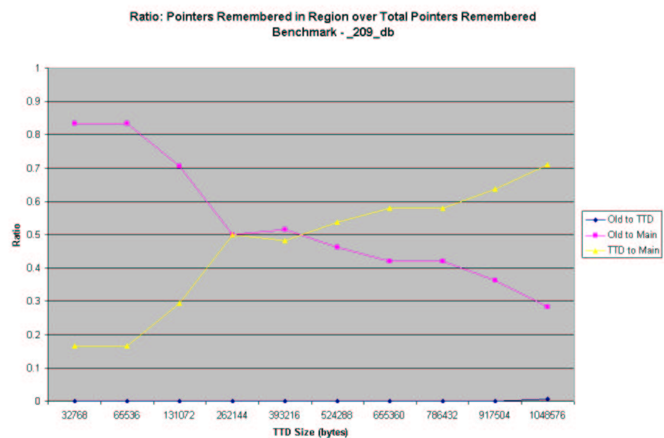
Figure 11: Survival Rate Percentage Differences



(a)

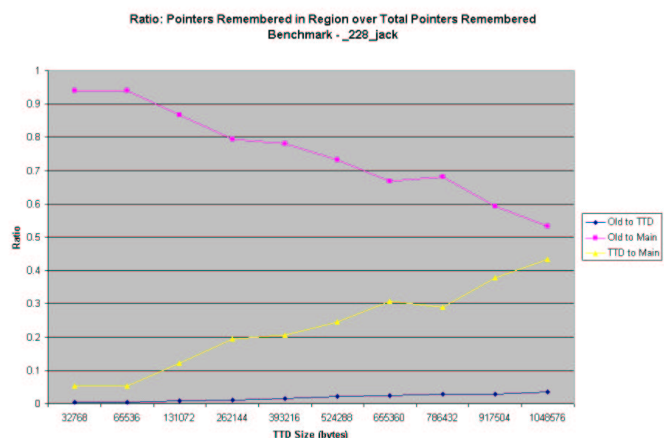


(b)

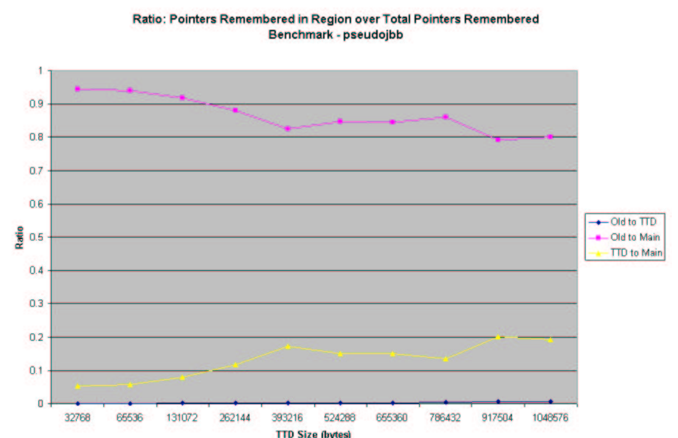


(c)

(d)

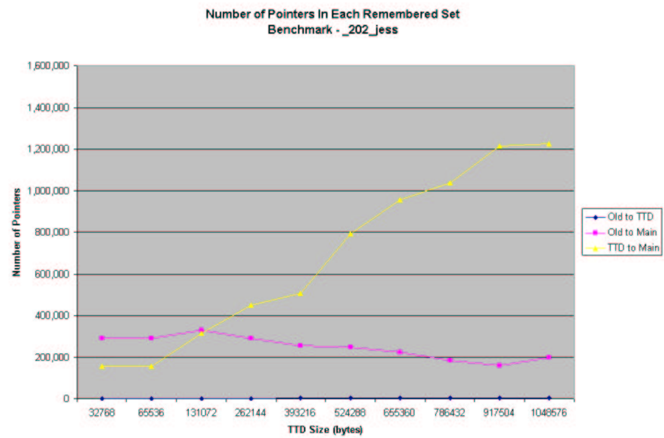


(e)

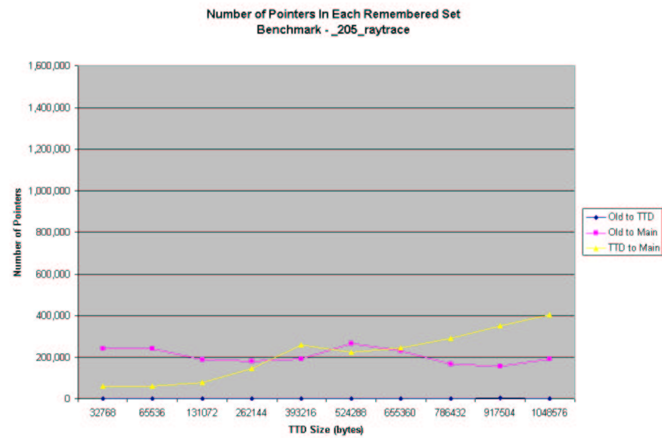


(f)

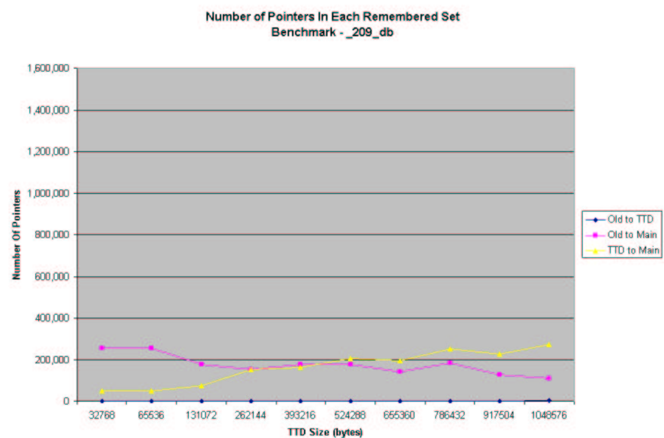
Figure 12: Remembered Set Sizes - Percentages by Region



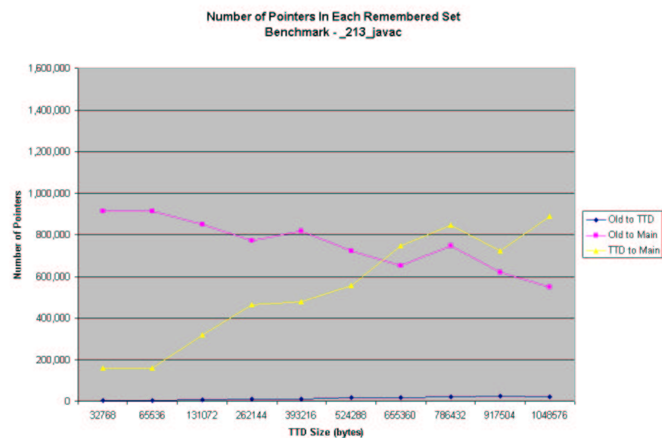
(a)



(b)

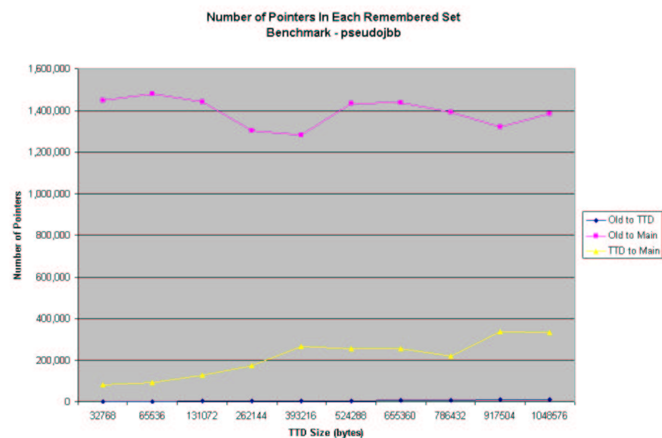


(c)



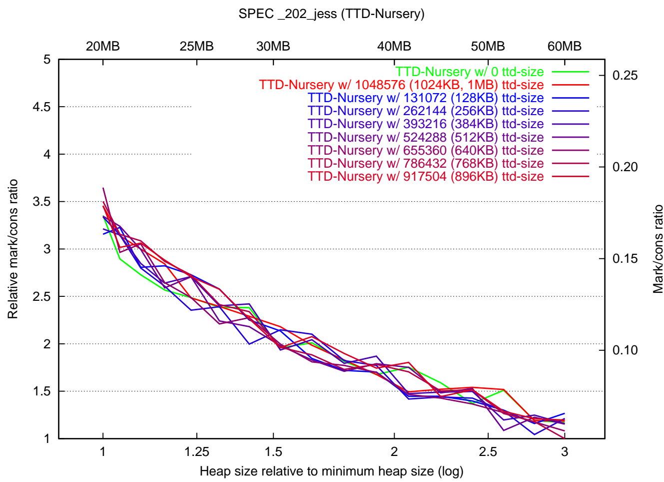
(d)

(e)

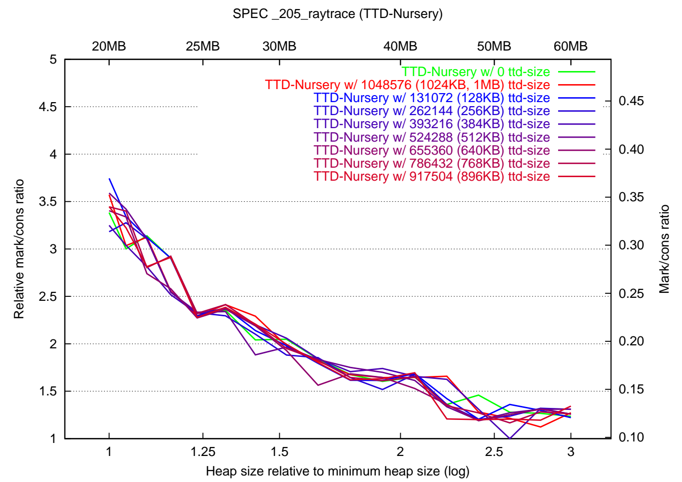


(f)

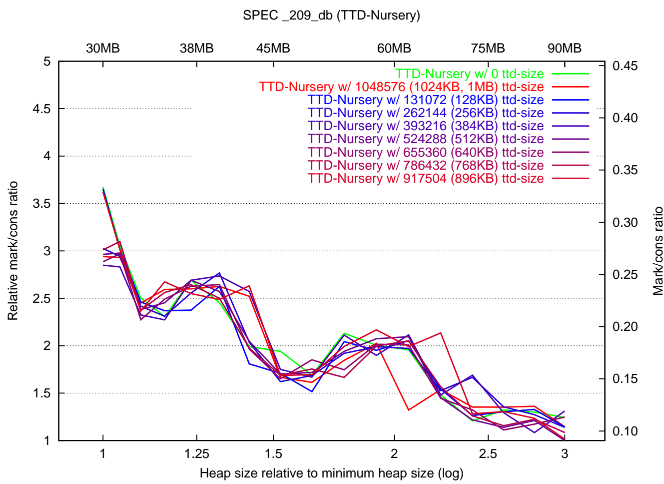
Figure 13: Remembered Set Sizes - Raw Numbers by Region



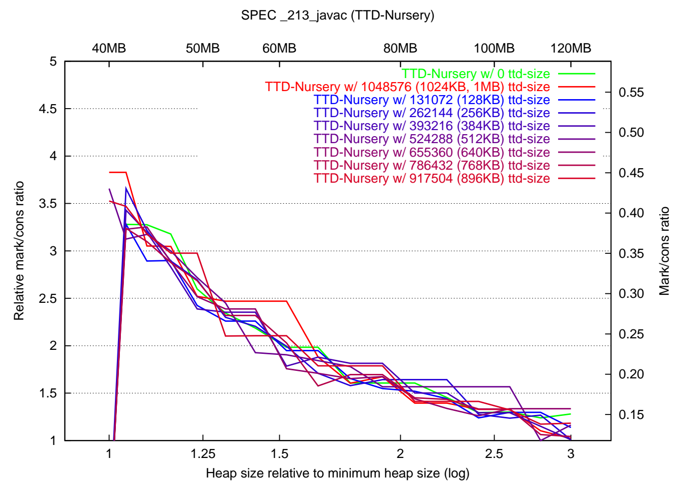
(a)



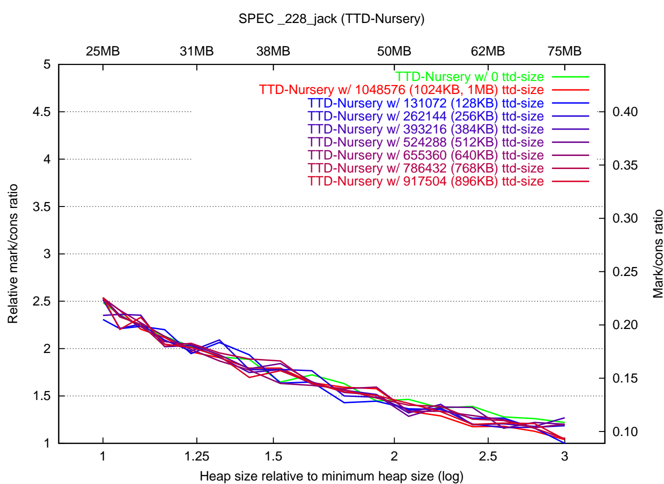
(b)



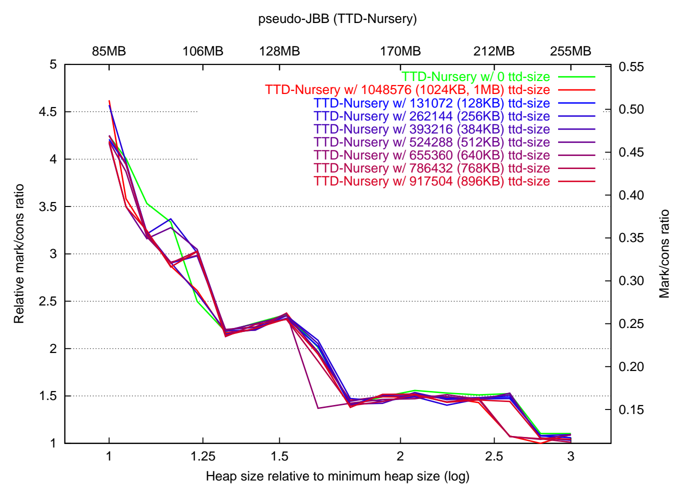
(c)



(d)

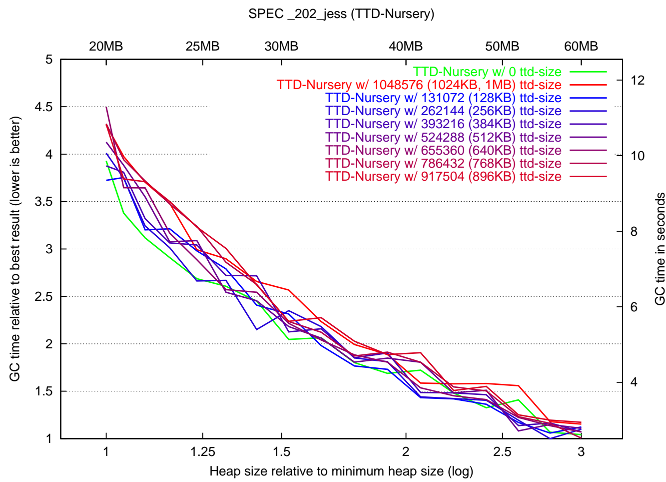


(e)

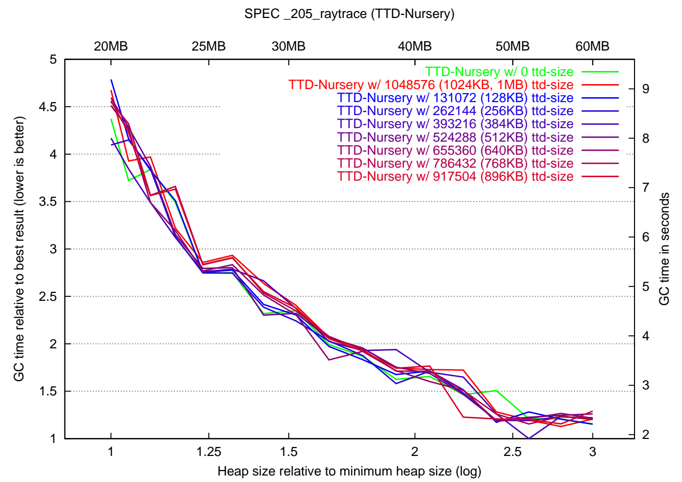


(f)

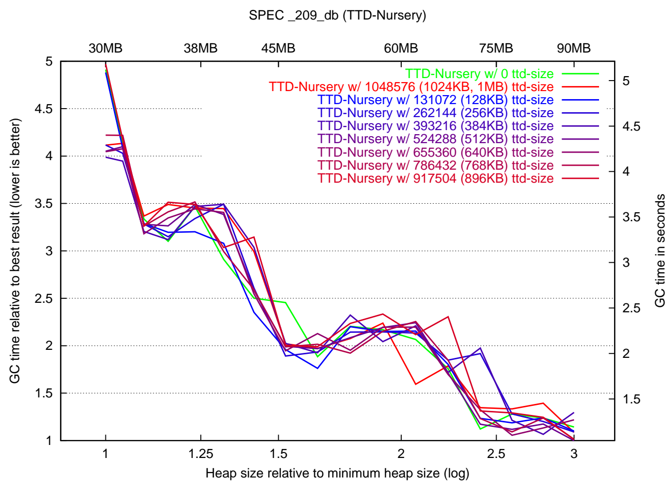
Figure 14: Mark/Cons Ratio



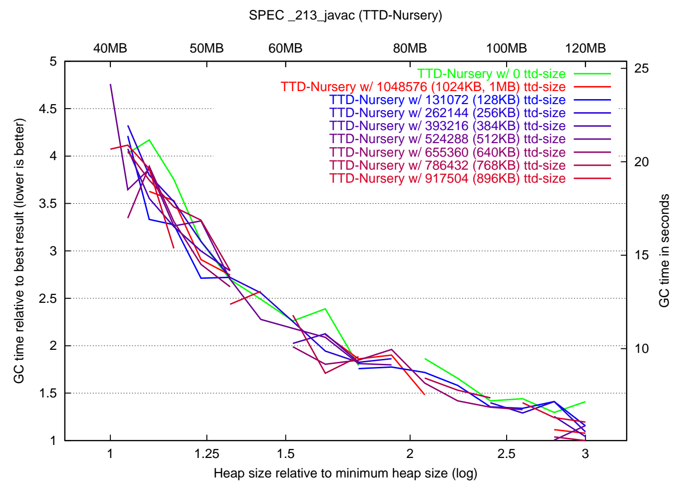
(a)



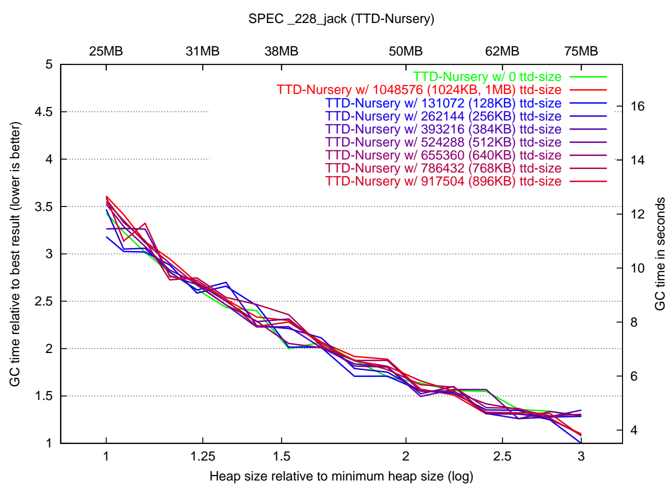
(b)



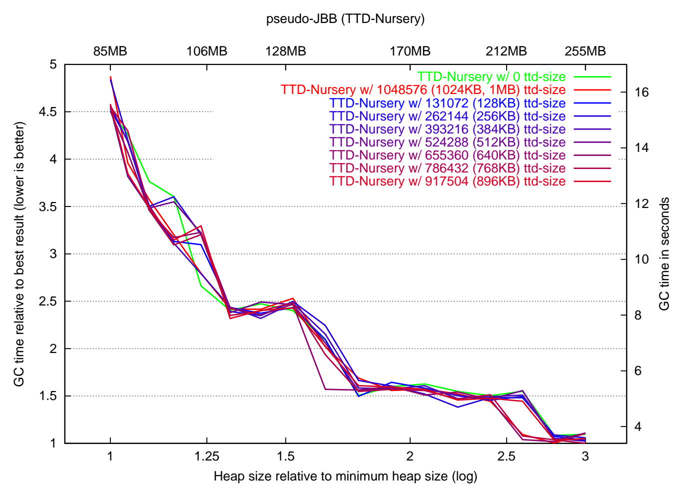
(c)



(d)

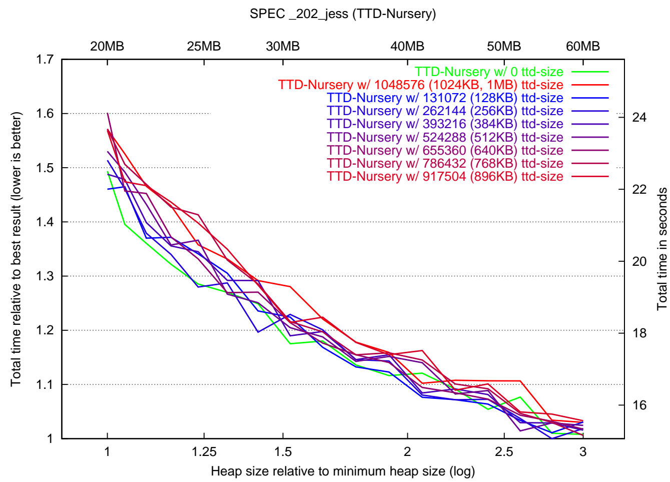


(e)

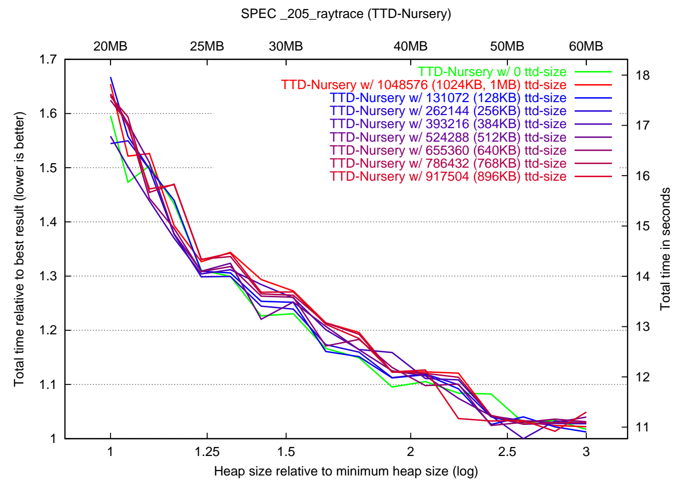


(f)

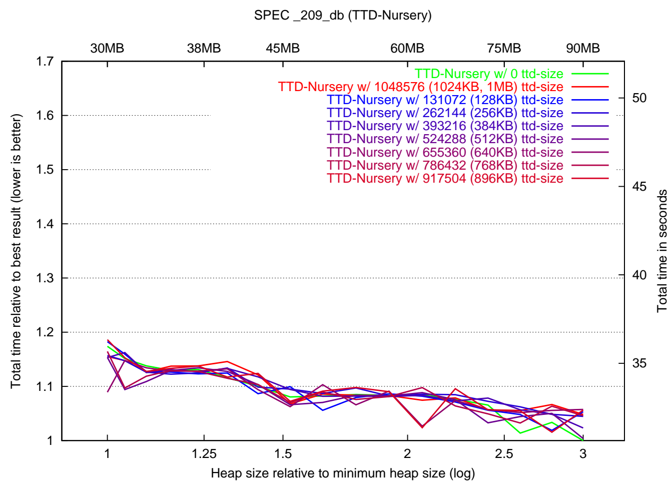
Figure 15: Time Spent in Garbage Collection



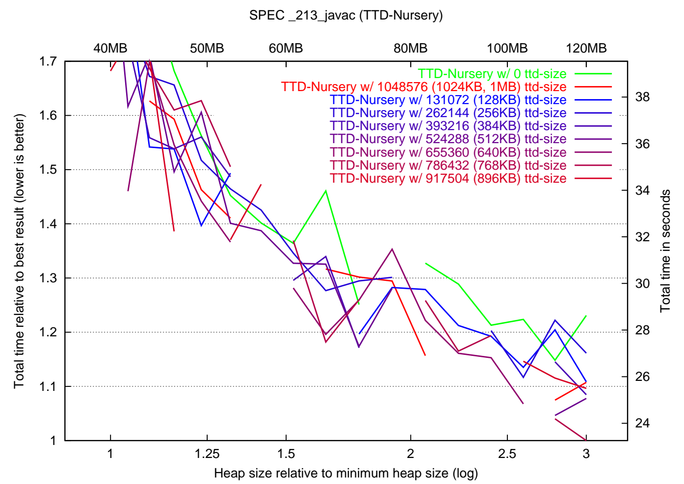
(a)



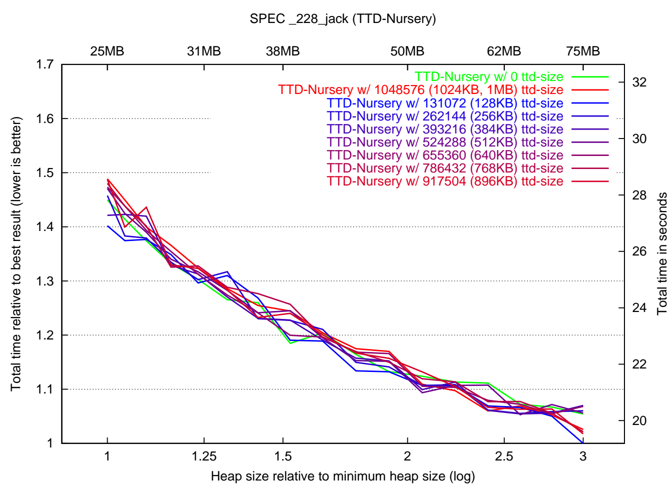
(b)



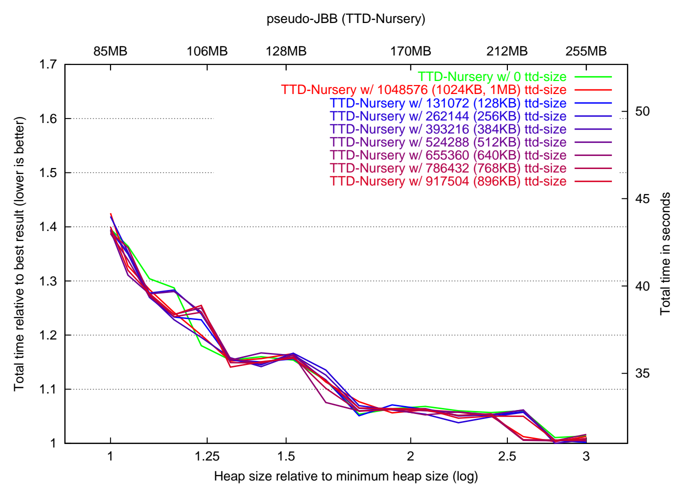
(c)



(d)



(e)



(f)

Figure 16: Total Time