Incremental Parallel Garbage Collection

Paul Thomas
(pt06@doc.ic.ac.uk)
Department of Computing
Imperial College London

Supervisor: Dr. Tony Field

June 2010

Abstract

This report details two parallel incremental garbage collectors and a framework for building incremental collectors for the object oriented language Java. It describes the implementation of the collectors and the framework and evaluates the two collectors through well known benchmarks and metrics. We extend Jikes a virtual machine designed for researchers with these collectors and the framework. The first of the collectors is an parallel incremental mark-sweep collector. The second is an parallel incremental copy collector based on previous work by Brooks[17]. These are the first incremental collectors made for Jikes. Experiments with the collectors indicate a large improvement in "real-time" characteristics of the collectors as parallelisation is increased.

Acknowledgments

I would like to thank my supervisor, Dr Tony Field for his constant support and motivation throughout the entire year. I would like to thank Andy Cheadle for his advice and insight into mysterious world of garbage collection. I would also like to thank the Jikes community for their support throughout the project. A big thanks goes to my friends for listening to my rantings about garbage collection for the last nine months.

Contents

1 Introduction **5**
1.1 Introduction . . . 5
1.1.1 Contributions . . . 5
1.1.1.1 A parallel incremental mark-sweep collector . . . 6
1.1.1.2 A parallel incremental copying collector . . . 6
1.1.1.3 A framework for incremental collectors . . . 6

2 Background **7**
2.1 Overview of garbage collection . . . 7
2.1.1 Reference counting . . . 7
2.1.2 Mark-sweep . . . 8
2.1.3 Mark-compact . . . 10
2.1.4 Copying . . . 10
2.1.5 Generational . . . 11
2.2 Garbage collector terminology . . . 12
2.2.1 Incremental, parallel an concurrent collection . . . 12
2.2.2 Real-Time collection . . . 13
2.2.3 Tri-colour marking abstraction . . . 13
2.2.4 Barriers . . . 14
2.2.4.1 Write Barriers . . . 14
Boundary . . . 14
Object . . . 14
Card . . . 15
Hybrid . . . 15
Brooks . . . 15
Yuasa . . . 15
2.2.4.2 Read Barriers . . . 15
Conditional . . . 15
Unconditional . . . 15
2.2.5 Real-time scheduling . . . 16
2.2.5.1 Time based . . . 16
2.2.5.2 Work based . . . 16
2.3 Real time algorithms . . . 17
2.3.1 Baker's algorithm . . . 17
2.3.2 Brooks collector . . . 17
2.3.3 Replicating garbage collection . . . 18

2.4 State-of-the-art Real-time collectors . . . 19
2.4.1 Cheng's collector . . . 19
2.4.2 Sapphire . . . 20
2.4.3 Metronome . . . 21
2.4.4 Stopless, Chicken and Clover . . . 22
2.4.4.1 Stopless . . . 22
2.4.4.2 Chicken . . . 23
2.4.4.3 Clover . . . 23
2.5 Java . . . 23
2.6 Jikes . . . 24
2.6.1 MMTk . . . 24
2.6.2 Magic . . . 26
2.6.3 The Jikes Compilers . . . 27
2.6.3.1 Baseline . . . 27
2.6.3.2 Optimising . . . 27

3 Design and Implementation **28**
3.1 Design and Implementation . . . 28
3.1.1 Indirection pointers . . . 28
3.1.2 Incremental Mark Sweep . . . 29
3.1.2.1 The write barrier . . . 30
3.1.2.2 Work calculations . . . 30
3.1.2.3 Triggering mechanisms . . . 33
3.1.2.4 The final phase . . . 33
3.1.2.5 Incremental Parallel Mark Sweep . . . 33
3.1.3 Brooks Style Incremental Copy Collector . . . 34
3.1.3.1 Tracing algorithm . . . 34
3.1.3.2 Barriers . . . 34
Write Barrier . . . 34
Read Barrier . . . 35
3.1.3.3 Fixing the read barrier . . . 36
3.1.3.4 Parallel Brooks Style Incremental Copy Collector . . . 36
3.1.4 Framework for incremental collectors . . . 38
3.1.5 Testing . . . 38
3.1.5.1 Jikes . . . 38
3.1.5.2 MMTk test harness . . . 42

4 Evaluation **43**
4.1 Benchmarking . . . 43
4.1.1 Indirection pointers . . . 44
4.1.2 Incremental Mark Sweep . . . 44
4.1.2.1 Experiment 1 . . . 45
Conclusion . . . 48
4.1.2.2 Experiment 2 . . . 48
Conclusion . . . 51
4.1.2.3 Experiment 3 . . . 51
Conclusion . . . 53
4.1.2.4 Experiment 4 . . . 53

Conclusion . . . 55
4.1.2.5 Experiment 5 . . . 57
Conclusion . . . 58
4.1.3 Incremental Parallel Mark-Sweep . . . 58
4.1.3.1 Experiment 1 . . . 59
Conclusion . . . 61
4.1.3.2 Experiment 2 . . . 61
4.1.3.3 Experiment 3 . . . 63
Conclusion . . . 66
4.1.3.4 Experiment 4 . . . 67
4.1.4 Incremental Copy Collector . . . 68
4.1.4.1 Experiment 1 . . . 69
4.1.4.2 Experiment 2 . . . 69
Conclusion . . . 73
4.1.4.3 Experiment 3 . . . 74
Conclusion . . . 76
4.1.4.4 Experiment 4 . . . 76
4.1.5 Parallel Brooks Style Incremental Copy Collector . . . 78
4.1.5.1 Development machine results . . . 78
4.1.5.2 Experiment 1 . . . 78
Conclusion . . . 82
4.1.5.3 Experiment 2 . . . 82
4.2 Conclusions . . . 85
4.2.1 The Collectors . . . 85
4.2.2 The Framework . . . 85
4.2.3 Future work . . . 85
4.2.3.1 Extensions to the current collectors . . . 85
4.2.3.2 Incremental Immix . . . 86
4.2.3.3 A Collector Similar To The Metronome . . . 86

A Evaluation Results **88**
A.1 Incremental Mark Sweep Collector . . . 89
A.2 Parallel Incremental Mark Sweep Collector . . . 92
A.3 Incremental Copying Collector . . . 101

B Evaluation Graphs **104**
B.1 Overhead, Average and Max Pause Graphs . . . 105
B.1.1 Incremental Mark-Sweep . . . 105
B.1.2 Incremental Mark-Sweep Mark Bit . . . 108
B.1.3 Incremental Mark-Sweep time-work . . . 111
B.1.4 Parallel Incremental Mark-Sweep . . . 114
B.1.5 Parallel Incremental Mark-Sweep Work Time . . . 117
B.1.6 Incremental Copying Collector . . . 120
B.1.7 Incremental Copying Collector work-time . . . 123
B.2 MMU, Average CPU utilistaion and Pause Time Distribution . . . 126

Chapter 1

Introduction

1.1 Introduction

Garbage collection is the automatic process of reclaiming memory acquired by a program that it no longer needs. Many conventional garbage collectors cause the program to pause whilst garbage collection is in progress. These "stop-the-world" collectors are well suited to batch processing applications but are less appropriate for real-time applications, including interactive and safety-critical systems, where the ability to react to events within a specified time is either desirable or essential.

Incremental collectors seek to avoid the pause problem by interleaving the garbage collector and program. The idea is to replace the single stop-the-world pause with several smaller pauses within a single thread of execution. A key issue is ensuring that the collector makes sufficient progress between these shorter pauses in order to meet some specified quality of service requirements.

One way to reduce the duration of number of pauses is to perform the collection in parallel. By exploiting parallelism either more collection work can be performed during the same pause time, and the number of pauses thus reduced, or each pause time can be reduced whilst keeping the total number of pauses the same. Whilst a number of parallel stop-the-world collectors have been developed (see Section 2.1), very little work has be done on combining incremental collection with parallelism.

In this project we developed a parallel incremental garbage collector for the Java programming language, specifically for the IBM Jikes Research Virtual Machine (RVM). This is an implementation of a JVM written in Java itself and is thus allows all the abstractions of Java. A key aspect of the project is the development of additional tool support for incremental collection in the RVM, through a series of MMTk "plans". MMTk is a collection of Java class files for simplifying the building of garbage collectors, as detailed in Section 2.6.1.

1.1.1 Contributions

We have extended the Jikes RVM in the following ways:

1.1.1.1 A parallel incremental mark-sweep collector

We present a parallel incremental mark-sweep collector with the following key features:

- Parallel: Any number of collector threads can be run at the same time.
- Incremental: The collector will work in incremental steps interleaving with the programs threads. We added this to Jikes.
- Soft Real-Time: The collector is able to operate within some defined real-time bounds. This is new to Jikes.
- Incremental work calculations: A new way to calculate the amount of incremental work to be done in each incremental phase is presented based on work and time. This is new to the field and has its foundation in previous work based purely on time.
- Barriers: The collector utilises a slot based write barrier.

1.1.1.2 A parallel incremental copying collector

A parallel incremental copying collector has been made with the following features:

- Parallel: Any number of collector threads can be run at the same time.
- Incremental: The collector will work in incremental steps interleaving with the programs threads.
- Soft Real-Time: The collector is able to operate within some defined real-time bounds.
- Barriers: The collector utilises a slot based write barrier and an unconditional read barrier. The unconditional read barrier is a new feature added by us to Jikes.
- Defragmenting: The collector defragments memory.

1.1.1.3 A framework for incremental collectors

Jikes has a rich set of collectors currently implemented (see Section 2.6). It is predominantly aimed at classical stop-the-world collectors. This however does not reflect cutting edge research in garbage collection, more aimed towards real-time collectors. Jikes currently provides no explicit support for real-time style collectors. We have extended the MMTk by creating a new plan (see Section 2.6.1) that will be extendible by future programmers so that they can create their own incremental collectors.

Chapter 2

Background

2.1 Overview of garbage collection

Garbage collection is a way to automatically reclaim memory allocated by a computer program. For an excellent overview of garbage collection, see Garbage Collection: Algorithms for Automatic Dynamic Memory Management [35]. The first programming language to have a garbage collector was Lisp, written by McCarthy in 1960 [39]. The first Lisp system utilised a traditional mark-sweep collector (Section 2.1.2). This meant that programmers no longer had to manually manage memory.

Traditional garbage collectors have two main aims: Firstly to identify an object[1] that cannot be accessed by the program (reachability) and secondly to reclaim the memory used (collection). A separate part of the memory manager is responsible for allocating memory for new objects (mutation).

The reachability of an object is established in two main ways in garbage collection. The first is *tracing* (Figure 2.1), done by traversing the graph of all objects in memory. This graph begins from the *root* set. The root set comprises all objects which are directly accessible by the program. It will generally comprise global variables, static data and the control stack. An object that is reachable is *live* and thus will not to be collected. Once the graph is fully traversed any objects which are not reachable are *dead* and will be collected.

2.1.1 Reference counting

The reference counting algorithm actually does not use a trace. Instead it maintains a count for each object, of the number of objects which reference itself. When the count for an object has reached zero it is no longer reachable and is reclaimed. The mutator is responsible for maintaining these counts and requires a way to "trap" mutations of the memory graph. This is done with a write barrier (see Section 2.2.4.1).

The main strength of this approach is that overheads are distributed throughout execution of the program. Unlike more traditional schemes such as mark-sweep which must pause the program whilst collection takes place. However this cost is considered somewhat "lumpy" as the cost of removing pointers is dependent on the size of the subgraph it points

[1] Object is used to denote a continuous range of bytes in memory as well as the classic language level object. Note that cell and object essentially are the same thing.

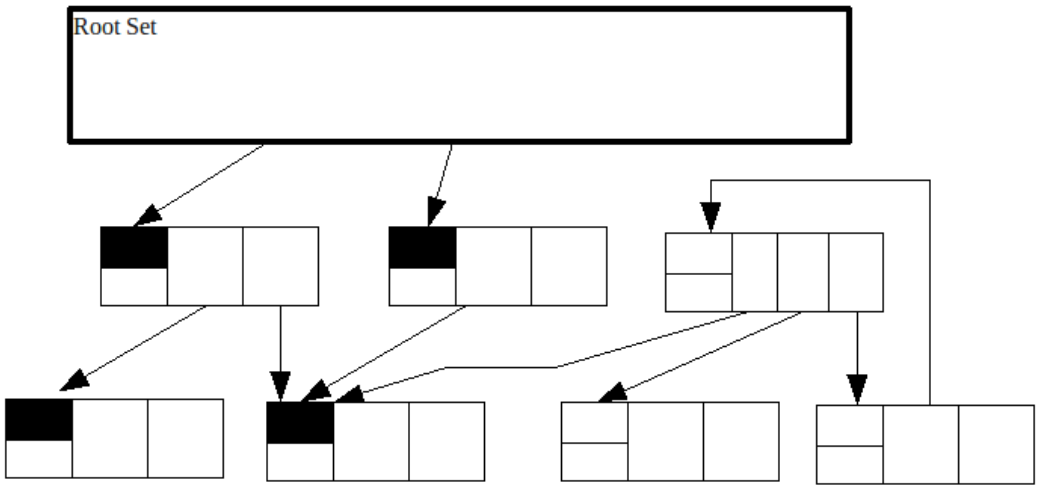


Figure 2.1: Image demonstrating a completed trace of a memory graph. Note that live objects have a cell marked black

to. From this approach it is easier to collect unused cells due to the references being maintained during program run-time. This differs from other traditional tracing collectors where dead objects reside in memory until collection is run.

There are two main disadvantages to reference counting collection. Firstly each object requires additional space needed to store the counter. This is stored in the object header. Due to numeric overflow (caused by incrementing this number over its bounds) the correctness of the algorithm can be compromised. Secondly reference counting algorithms cannot handle cyclic data structures.

A data structure is cyclic if an object can be reached by following a path a pointers from itself. Cyclic data structure are common features of programs including: doubly linked lists, circular buffers and graphs with cycles in them. If the cycle is unlinked from the main memory graph its members will still have a positive reference count even though they are dead demonstrated in Figure 2.2.

2.1.2 Mark-sweep

Mark-sweep [39] requires all objects to have an extra bit in their object header for the mark-phase. It is traditionally a stop-the-world collector meaning the program completely stops when the garbage collection routine begins. Garbage collection is triggered when an object allocation to the heap fails. At this point all mutators are paused[2], the tracing routine begins marking all live objects. After this "mark" phase is complete, the collector begins scanning from the start of the heap for any objects which are not marked (dead objects) and frees that memory. When the second phase gets to the end of the heap the mutators are resumed.

[2]Note that there may be several muator threads running at any one time

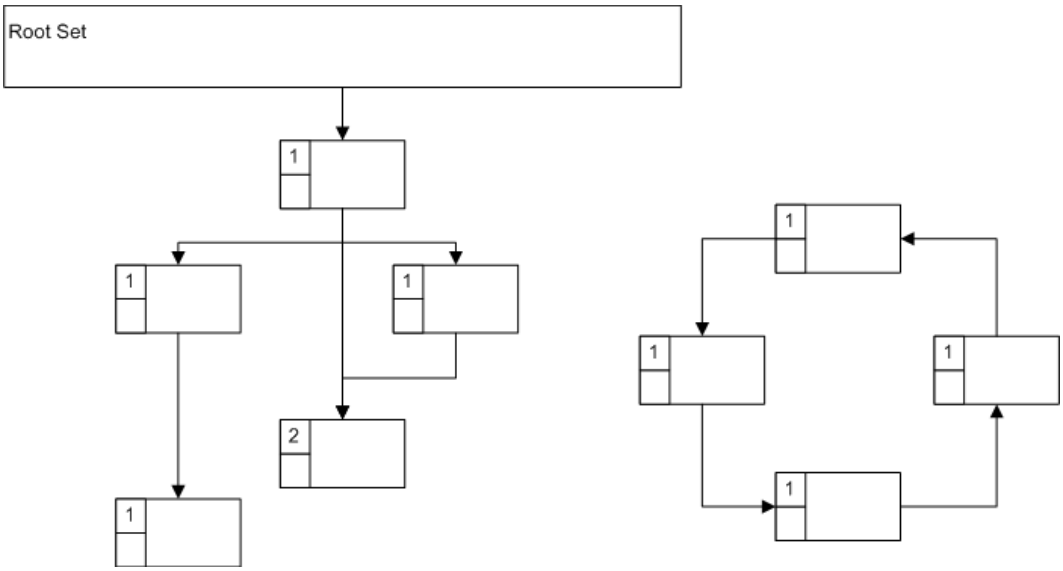


Figure 2.2: Image demonstrating the issues with cycles in reference counting algorithms

Allocation in this algorithm is done via a *freelist*. A freelist works by linking together unallocated regions of memory together in a linked list. When an object is reclaimed the newly freed memory is simply added to the list. The problem arises when a particularly large object needs to be allocated and there is not a large enough chunk to accommodate it. Subsequently a garbage collection must occur or consolidation of blocks that are contiguous, both of which are computationally expensive.

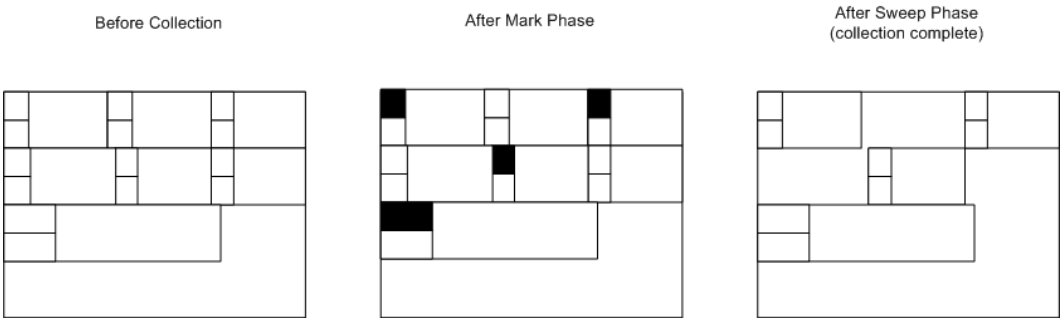


Figure 2.3: Image demonstrating the mark-sweep collection algorithm and its fragmenting nature

This algorithm correctly handles cyclic structures due to it scanning the memory graph as a whole in one phase. However mark-sweep is a fragmenting collector. This means that when it collects it causes memory to become fragmented (Figure 2.3). It is also a stop-the-world collector which makes it unsuitable for real time systems.

2.1.3 Mark-compact

Memory fragmentation means that fresh object allocation can be expensive, as a search must be done over the freelist in order to locate a free slot that can accommodate newly allocated object. This is normally done with one of the classic fits algorithms such as: first-fit, best-fit, next-fit and worse-fit. This can lead to an allocation performance hit for the mutator. The mark-compact collector solves this by compacting memory during collection.

All the mark-compact algorithms have the same mark phase as the mark-sweep collector. They employ a different form of allocation known as *bump-pointer* allocation. In this there is a pointer to the next place to allocate an object. When objects are allocated this bump pointer is increased by the size of the new object. This means there is always the same cost to find a place for a newly allocated object. Instead of the sweep phase there are two or more phases where objects must be moved and their pointers fixed accordingly.

The two finger algorithm [47] uses two sweeps in order to complete its compaction. It assumes all cells are the same fixed size. In the first phase it moves all live objects at the top of the heap to empty slots at the start of the heap. This works with two pointers (the two fingers) which begin at either end of the heap, the one on the left pausing when it finds a dead object and the one on the right stopping when it finds a live object. When both are paused the dead object is then removed and the live object copied in its place and a forwarding pointer is left in its old place. The algorithm repeats this until both "fingers" meet. After this a second phase starts from the beginning of the heap. Here pointers to forwarded objects are re-linked to the correct object by replacing the original pointer with the forwarding pointer.

The Lisp 2 algorithm [38] also known as the sliding algorithm allows for variable sized objects and preserves the order of the objects. This algorithm is split into three phases and requires each object to have an additional pointer-sized field to store the address of where it will be moved. The first pass, beginning from the head of the heap, sets the field of all live objects to the new address they will be moved to. This is calculated by taking the sum of the sizes of all live objects encountered thus far and adding that to the start of heap pointer address. In the second phase all objects with pointers to other objects are updated to their new location. The third phase relocates the object to its new address and clears the address stored in the object header.

Threading [28] overcomes the issue with the extra pointer-sized space per object Lisp 2 requires. In threading the object graph is temporarily changed so that all objects referring to some object P, are contained in a linked list within P. P is then relocated and all objects in the list updated with the new location of P. After this P is returned to how it was originally. This is a two phase process and is actually a description of the widely adopted Jonkers [36] variant of threading. The original by fisher [28] had the object P with a pointer to the linked list stored elsewhere in memory. This is expensive due to the extra overhead needed to store the lists.

2.1.4 Copying

The Copying collector is another tracing collector [20]. Originally called the *Cheney* collection algorithm it is also known as the semi-space collector and is similar to compactors in that allocation is cheap, handled by a bump pointer. It also causes compaction to occur naturally through the copying of objects. This, in combination with bump pointer allocation, means variable sized objects are allocated easily.

The collector divides the heap into two "semi-spaces". The first of these (from-space) contains old data and the second (to-space) contains all live data. The collector begins by "flipping" to-space and from-space so that the current live objects are now in from-space and to-space is empty[3]. The collector then traverses the memory graph of objects in from-space copying each live cell into to-space. When the graph has been fully traversed all the live objects are in to-space. At this point the from-space is reset; cleared and the pointers reset. This collector ensures that mutators only ever see objects in to-space which is also known as the to-space invariant.

This collector has several advantages over other collectors. Allocation is very cheap due to the compactifying nature of the copying. It has a naturally defragmenting nature through the copying of objects at each collection (see Figure 2.4). The out of memory check is just a simple pointer comparison[4]. New memory is easily acquired by increasing the value of the free space pointer. Finally similar to mark-sweep due to the tracing nature of the algorithm it deals with cyclic data structures.

The collector is not without its faults. Firstly it is a stop-the-world collector meaning the program must be stopped for collection. It automatically utilises twice the amount of memory as that of all other non copying collectors due to the two semi-spaces. The counter argument to this extra use of space is that more space is gained from the loss of fragmentation. Also if object lifetimes are long then the collector has to continually copy these objects which also causes the flip operation to become more expensive.

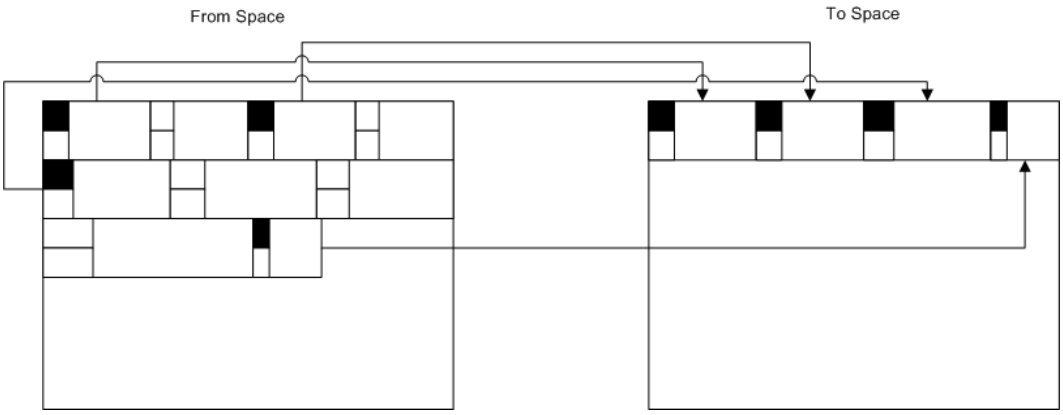


Figure 2.4: Picture demonstrating the copying collectors defragmentation abilities

2.1.5 Generational

Generational collectors work by splitting the heap into two or more sections called generations. Objects begin life in the youngest generation and are promoted to an older generation. Collection is done on a generation basis so if the youngest generation is full then it is collected and if an older one is full it, and all generations younger than it, are collected. This tends to give shorter pause times than a standard collector due to the weak generational hypothesis [48].

[3] the reason for this is explained shortly

[4] Comparing addresses of the bump allocation pointer and the free space pointer

This states that "most objects die young". This means that objects in the younger generation will be the most likely to die and as such this space tends to be smaller than the other generations. In turn this causes lots of smaller collections to occur frequently and tends to free a lot of the dead objects. Several other researchers [12, 30, 46, 53] back up Ungar's claim whilst little evidence to the converse seems to exist.

The *promotion policy* determines when an object in a young generation is moved to an older generation. There are several different approaches to promotion. The time based approach, whereby an object is promoted if it is live for a certain length of wall clock time. The problem is that program execution rate is machine dependent so the policy becomes non portable. There is the option of monitoring how much memory has been allocated since the object was allocated, known as memory time. This is the most accurate[5] and consistent across systems. Finally there are approaches such as number of GC cycles since allocation and when the younger generation is full objects are promoted.

There is an issue with determining each generation's root set. At the beginning of each collection the root set for the generation to be collected must be calculated. This means when a collection on an older generation occurs the tracing algorithm must also consider all pointers from younger objects (which are also live) that point to the older generation as roots, these are known as "inter-generational" pointer. This is why, when collection of an older generation occurs, all younger generations must also be collected. This means the collector only need know about old to young pointers and have them as part of the root set for collection of younger generations. As these are much rarer this is favorable to remembering young to old pointers. In order to track these a write barrier must be utilised.

There are two versions of the write barrier to track and store the old to young pointers: a remembered set barrier with sequential store buffers [33] or a card marking barrier. These are outlined in more detail in Section 2.2.4.1. There are also schemes for a hybrid of the two which tends to offer improvements according the results in [32] and also outlined in Section 2.2.4.1.

Generational collection is considered a "half-way" house between a stop-the-world collector and an incremental collector. It can offer the benefits of multiple collectors as it can have a different kind of collector to collect one generation than another. Generally a mark-sweep style collector is used for younger generations and a copy collector for later generations. The issue comes with the write barrier. Studies have been very varied on the impact this has on program execution time. Estimates of the barrier over-head vary from, 2% for Ungars collector for Berkely Smalltalk [48] and 3% for SOAR [49] to, 4%-27% for Chambers et al. reports in the optimising compiler for SELF[6] [18, 19] and 5%-10% for Appel [6]. This variation in figures shows that this is very language/compiler dependent.

2.2 Garbage collector terminology

2.2.1 Incremental, parallel an concurrent collection

The terms incremental, parallel and concurrent are sometimes used inconsistently in the garbage collection literature. We will use the following terminology:

- Incremental: An incremental collector does small chunks of work and in between allows the mutator to resume. This differs from traditional tracing collectors which

[5] Accuracy being the most appropriate time to promote an object
[6] SELF is a smalltalk like language

must finish a complete GC cycle or stop and forget the work they have done if the mutator needs to resume.

- Parallel: A parallel collector utilises multiple threads simultaneously to do GC work. This can be at the same time as the mutator or when the mutator is stopped.
- Concurrent: A concurrent collector runs concurrently on a separate thread to the mutator. This means that on a multi-core machine the GC would run at the same time as the mutator.

2.2.2 Real-Time collection

A "real-time" collector ensures it does not encroach on a real-time application bounds. Most GC literature aims to minimise the maximum pause time of a collector so that it does not pause the program for longer than the real-time application can tolerate. The issue with this is that no thought is given to the window which the application needs to run for it to do some useful work (Figure 2.5).

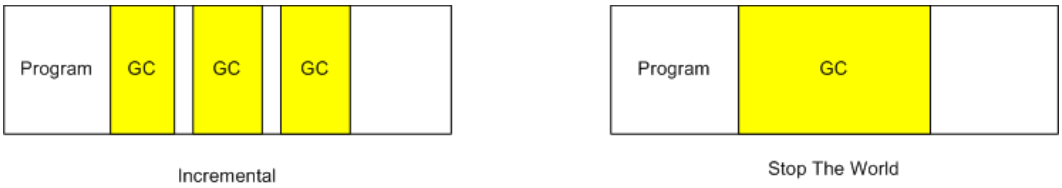


Figure 2.5: Picture demonstrating the problems with a burst of collection compared to a stop-the-world collection.

It is important to consider utilization introduced in Cheng and Belloch's paper [22]. Both maximum pause time and minimum mutator utilisation (MMU) are considered. MMU is the minimum window of time required by a real-time application to make sufficient progress. Consider a nuclear reactor monitoring system which can tolerate a maximum pause time of 50ms and it requires a 5ms window to make sufficient progress. This means there must be an MMU of 10% with maximum pause times of 50ms in order for the program to work correctly.

There is also hard verses soft real-time which most papers do not distinguish between. Soft real-time is a system whereby real-time deadline violations are tolerable and thus the program can wait until the workload is completed. The system may, however, operate correctly in a degraded state. Hard real-time is strict in that if the real-time deadlines are violated the program will fail.

2.2.3 Tri-colour marking abstraction

The tri-colour marking abstraction [23] is used to reason about the correctness of reachability GC algorithms. The object graph is represented with each node being one of three colours: white, black or grey. Black nodes are ones which have been visited by the collector and all object pointers in that node traced. Grey nodes are ones which have been marked for processing and must have all their own object pointers traced and subsequently the grey node is turned black. Once there are no grey nodes remaining (so they are all black) the GC

terminates. When a pointer to a white node is traced it is turned grey. The white nodes currently have no links found to them. If they stay white when there are no grey nodes left these are considered dead objects. This abstraction has two separate invariants associated with it the strong tri-colour invariant and the weak tri-colour invariant

The strong tri-colour invariant [23] states that no black node points to a white node. It can be demonstrated in the classic copying collector whereby all nodes in to-space are black nodes, every node these point to in from-space are grey and all other nodes are white. The weak tri-colour invariant is aimed more at incremental "snapshot-at-the-beginning" collectors [42]. It states that objects coloured white pointed to by a black node are also reachable from some grey node through a chain of white nodes. This ensures if a white object is pointed to by a black node it will eventually be traced due to the grey node. Both invariants ensure that no reachable object will be collected.

2.2.4 Barriers

Barriers have two main roles in garbage collection. The first is in generational/regional collectors. Here barriers are used to keep track of references that cross generations/regions. Secondly they are used to synchronise mutator and collectors in real-time schemes. Barriers come in two different forms: *static* and *non-static*. Non-static barriers only cover the heap; static barriers cover the heap as well as registers, stacks and other data outside the heap. The impact of barriers on performance has been widely studied [32, 33, 31, 48, 17, 52, 15, 9] with varying measurements. Zorn [52] reported write barrier overheads of around 2-6% and worst case read barrier overheads of around 20%; Bacon et al. [9] reported costs for their Brooks style [17] read barrier as 4% average overhead on SPECjvm98 [3]. Probably the most useful statistics for us are that of Blackburn et al. [15], here they implement several well known barriers in the Jikes RVM. With write barriers having a low average overhead of 2% and less than 6% in worst case and read barriers having a low of 0.85% for an unconditional (see Section 2.2.4.2) read barrier on a PPC and 5% on a P4.

2.2.4.1 Write Barriers

Write barriers allow the collector to trap every write to memory. They are used throughout garbage collection in generational, incremental and concurrent collectors.

Boundary The boundary barrier is used in generational collectors. It tests to see if the source and the target of a write lie in different generations (and as such cross the generation boundary). If this is the case the source location to which the target references is stored. This can be utilised in different ways but generally is used for recording old to young generational pointers.

Object The object barrier is aimed at generational collection. It works by storing the source of every write in a sequential store buffer (SSB). Originally this allowed duplicate entries to be in the SSB, although there are plans whereby each time an object is put into the SSB a tag bit is set so that it only appears once. When a GC occurs it traces all the objects in the SSB in order to discover any interesting pointers such as cross generational pointers.

Card The card barrier is a unconditional[7] write barrier. It is aimed at generational collectors and assumes that the heap is divided into 2^k-byte logical cards. If a write occurs on one of these cards the card is marked as dirty by the barrier. When a GC is triggered the dirty cards are scanned in-case it contains a pointer of interest (i.e. an old to young pointer). The size of the cards determines the speed verses accuracy of this approach.

Hybrid A hybrid barrier is a combination of the above barriers. In the MMTk there is a hybrid implementation combining the object and the boundary barrier. It uses the boundary barrier for arrays and the object barrier otherwise. This provides minor improvements in mutator overhead to that of either barrier on their own [15].

The other version created by Hoskin and Hudson [32] combines the object barrier and the card marking barrier. Similar to the MMTk barrier this provided lower average barrier overhead than that of either barrier by itself.

Brooks The Brooks write barrier [17] is used in the incremental copy collector. It is there to ensure the strong tri-colour invariant holds. Every time there is a write to an object, if the object is in from-space it is copied to to-space. This, in combination with the Brooks read barrier (see Section 2.2.4.2), ensures that the mutator never sees a white or grey object.

Yuasa This barrier is utilised in the incremental mark-sweep snapshot-at-the-beginning collector [51]. This barrier ensures the weak tri-colour invariant is upheld by storing all write accesses. This essentially colours the objects grey. The stored set of objects are re-traversed at the end of collection as mutations to the memory graph may have caused once dead objects to become live.

2.2.4.2 Read Barriers

Read barriers are generally considered the most expensive of the two types of barrier. This is due to a program doing more reads than writes during normal execution. There have been many implementations of read barriers and some even created at the hardware level [40] in order to minimise the performance hit. We shall focus on software read barriers in this report.

Conditional A conditional read barrier has a test in it that checks if a tag bit is set. If this tag bit is set then the object has a forwarding pointer and that pointer is followed and the forwarded object will be returned; otherwise it will just return the object. There are variations of this [11] whereby if the tag bit is not set and the collector is running then the object is forwarded and the tag bit is set.

Unconditional An unconditional barrier returns the location of where the forwarding pointer points to. This means that all objects must always have a forwarding pointer. An excellent example of this can be seen in the Brooks read barrier. In this fresh objects are created with a forwarding pointer pointing back to the object itself and when a to-space copy is made this pointer is redirected to the to-space version.

[7] An unconditional barrier is one where there is no branching statements in it

2.2.5 Real-time scheduling

Scheduling for "real-time" collectors has two major goals. First it ensures that a collector collects enough dead objects so that the program will not halt. Secondly scheduling must ensure that the mutator makes sufficient progress. Scheduling is aimed at incremental collectors that do chunks of work at a time.

2.2.5.1 Time based

Time based scheduling [9] uses fixed time quanta to ensure even CPU utilisation. The mutator is scheduled to run for Q_T seconds and the collector for C_T seconds. This interleaved execution continues until a collection cycle is finished. Assuming a perfect scheduler for any given time interval $\triangle T$ as the minimum CPU utilization by the mutator over all time intervals of time width $\triangle T$, the MMU, $u_T(\triangle t)$, can be defined as:

$$u_T(\triangle t) = \frac{Q_T \cdot \left\lfloor \frac{\triangle t}{Q_T + C_T} \right\rfloor + x}{\triangle t}$$

The first part of the numerator is the number of whole mutator quanta in the interval and x corresponds to the size of the remaining partial mutator quantum:

$$x = max\left(0, \triangle t - (Q_T + C_T) \cdot \left\lfloor \frac{\triangle t}{Q_T + C_T} \right\rfloor - C_T\right)$$

As the size of the interval ($\triangle t$) increases the first equation simplifies to:

$$\lim_{\triangle t \to \infty} u_T(\triangle t) = \frac{Q_T}{Q_T + C_T}$$

This ensures collectors have constant mutator utilisation rate unlike work based approaches (Section 2.2.5.2). However against adversarial mutators the program can run out of memory as the time quanta dedicated to the collector is not enough. There were two suggestions to alleviate this. First force a full stop-the-world collection when memory was about to be exhausted. However this means the collector is no longer hard "real-time". A better solution suggested was to move the time based scheduler to a work based approach when memory became scarce. However this was never implemented as it also could stop the collector being hard "real-time".

2.2.5.2 Work based

A work based scheduler was first used in Baker's incremental collector [11]. It directly couples the amount of work the collector does with the rate at which the allocator allocates memory. For every unit of memory the mutator allocates the collector must do k units of collection. k is calculated when collection is started which ensures that the memory graph is fully traversed before exhaustion occurs. The formula for a tracing collector with a tracing rate m is:

$$m = \frac{H - L}{L}$$

where L words are traced during the allocation of $H - L$ new words. In reality L is not known and must be estimated at the start of a collection based on previous collections.

Work based scheduling may seem like a simple yet elegant solution to the scheduling problem. However there are issues. While it ensures that enough memory is collected, it is directly tied to the allocation rate of the mutator. This means that when the allocation rate of a program increases so will the time chunk spent in collection. If the mutator is adversarial, in that it purposely allocates memory in large amounts over small time intervals, then the MMU will begin to fall. Unfortunately this "bursty" nature of allocation is actually quite common in real life programs and while pause times still stay low the sum of pause times over a particularly "bursty" moment will possibly be worse than that of a stop-the-world collector.

2.3 Real time algorithms

In this section fundamental real-time collectors are outlined. These are core algorithms that most modern day collectors build upon.

2.3.1 Baker's algorithm

Baker's collector [11] builds on the copying collector of Cheney, making it incremental. It utilises a conditional read barrier to trap reads of objects in from-space. It also uses work based scheduling.

Memory is split into two regions, like the semi-space collector, from-space and to-space. From-space now contains live and dead objects at the same time. Like the semi-space collector collection is triggered when memory is exhausted. At this point it evacuates[8] the root set to to-space and then incrementally scans the rest of the memory graph. During this allocation is done at the opposite end of to-space meaning that all newly allocated objects are coloured black. This can lead to "floating garbage"; dead objects that will not be collected until the next GC cycle.

This collector starts it's collection cycle when memory has been exhausted. This means that these are a lower number of surviving objects, due to them having a longer opportunity to die. However there is also a greater amount of heap allocation during collection therefore there is a greater chance for page faults to occur. There is a trade-off between the overhead caused by the page faults and the increase in collector time that would be caused if collection had been triggered at more frequent intervals, due to more interleaving between the collector and the mutator.

The mutators always see the collection as complete after it is first triggered. This is achieved through the use of a conditional read barrier. In this barrier there is a check if the object is coloured white[9]. If this is the case then the barrier copies this object to to-space and mark it black. This was inherently expensive and is the main issue with Baker's collector.

2.3.2 Brooks collector

The Brooks [17] collector extends the Baker collector by reducing the cost of the read barrier, at the cost of some additional memory per object. This is done via the use of an unconditional read barrier with "indirection" pointers (Figure 2.6). These pointers point

[8] Evacuation encompasses the action of the copy

[9] In from-space

to their respective object in to-space (if one has been created); otherwise they point to themselves. This means that the read barrier code need only de-reference this pointer in order to return the latest version of the object. They are created by adding an extra pointer-sized field to the object model which holds the indirection pointer. Essentially this barrier is trading space for speed.

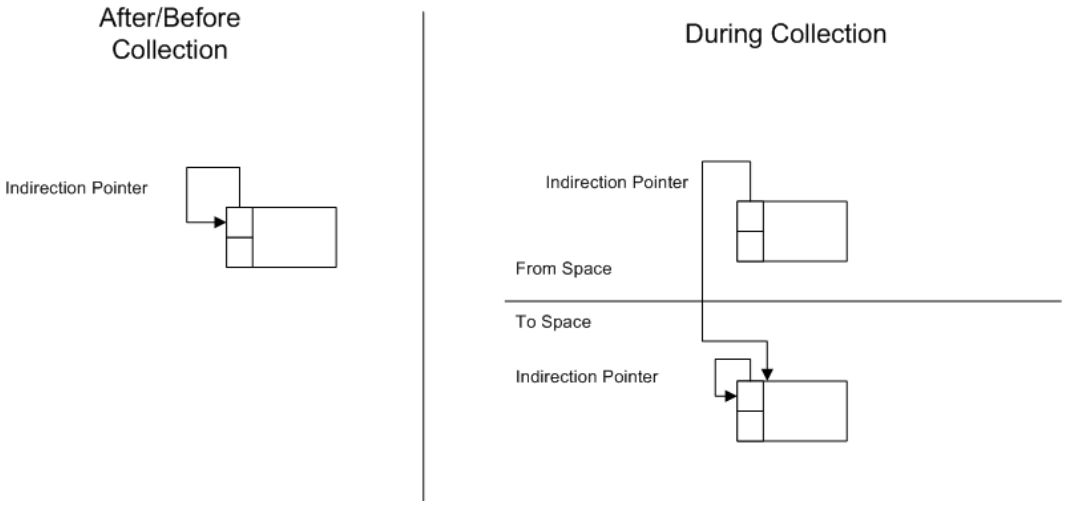


Figure 2.6: Picture illustrating Brooks indirection pointers

Without Baker's read barrier the strong tri-colour invariant is broken as objects are not automatically forwarded by the Brooks read barrier. In order to restore this Brooks utilises an update write barrier. This barrier means that whenever a write is made to the heap the barrier determines if the object has been forwarded or not. If it has it will forward the object and thus colour it black; otherwise it will just return the object.

The last modification Brooks proposes is an incremental scanning of the stack. As the top layers of the stack will be the most frequent to mutate, the incremental scanner starts from the bottom up. A pointer is used to record how far up the stack has been traced. The problem is when the collector is operating against an adversarial program that does many pops and then pushes in succession.

2.3.3 Replicating garbage collection

The replicating collector is an incremental collector written by Nettles et al. [41]. There is no Read barrier and the to-space invariant is ignored. Instead mutators see from-space objects until a collection cycle is complete. The collector is based on the Appel's Generational Collector [6].

In order to keep links from the object in from-space to their to-space counterparts an extra pointer-sized field is needed. This field stores a pointer to the object's to-space counterpart. The algorithm incrementally creates a graph of live objects in to-space. However these objects can be mutated after they have been copied. This is resolved by the use of a logging write barrier. Every time an object is mutated it is stored and then after the copying of all live objects is complete the log must be analysed and any updates required made to the to-space objects.

There is an interesting question here about the performance implication of using a write barrier as opposed to a read barrier. According to the findings of Nettels et al. it would appear that results are quite good for their implementation language (ML), having the write fixup taking only 3% of the collector's total time. There would be issues with a language with a higher level of destructive writes. There is also the issue of the extra bookkeeping involved in the mutator log as this may grow to be quite large. Whilst this makes it less suited to incremental collectors it is still well suited for concurrent collectors as less low level synchronisation is needed between threads.

2.4 State-of-the-art Real-time collectors

There is an extensive literature on garbage collection with many papers addressing concurrent and incremental collectors. However many of the papers do not have fully implemented algorithms and/or have little information on their real-time characteristics. The collectors below are the ones with the most interesting additions to garbage collection and clear evaluations of fully implemented algorithms. Interestingly there are only two collectors which can be truly classified as "real-time"; The Metronome and Garbage-First and even then these are only soft real-time.

2.4.1 Cheng's collector

Cheng's collector [21, 22] is a replicating, incremental, concurrent and parallel collector. It builds on work from Nettles and O'Toole's [41] incremental replicating collector which leaves objects in from-space and then updated the objects to-space counter parts with a mutation log maintained with a write barrier. This collector requires no read barriers, instead opting for a mutator logging barrier.

The collector overcomes several shortcomings of the Nettles-O'Toole collector. First it utilises stacklets which are a representation of the stack broken up into small parts and then each of these parts linked together. This means there is no static write barrier to monitor the stack. It also aids parallelism as it enables processors to work on one stacklet at a time, monitor its pushes/pops and synchronise accordingly. It also runs collector threads in parallel which enables the collector to process a greater workload. Cheng's collector is a fully replicating collector as opposed to that of Nettles and O'Toole which was a fully replicating collector during major collection cycles. This means that Cheng's collector requires more space due to objects being doubly allocated[10]. This is alleviated somewhat with a "2-phase" scheme: an aggressive phase, in which no objects are double-allocated and a conservative phase, in which the root set is recomputed and a second, shorter, collection which doubly-allocates objects. This causes survival rate to increase by 0.7%.

Several interesting methods are used in order to achieve parallelism. A centralised work stack is used in which all grey objects are pushed onto and collectors pop their work off. There is an issue with pushes and pops occurring simultaneously. In order to combat this Cheng employs "Rooms". Rooms can essentially be thought of as critical sections in which no two processes can be in at the same time. The shared stack has two rooms defined: one for popping and one for pushing. It can only do a pop if it is in the pop room and similarly for the push room. As only one process is allowed in a room at a time the issue of concurrent pushes and pops is removed. Interestingly there is no mention of concurrent

[10] As the collector is replicating the same object will exist in from and to space at the same time.

pushes and pops being shared so that if a push and a pop happen at the same time the stack itself never needs to be queried.

The scheduling for each collector thread is work based. This again is susceptible to the "bursty" nature of memory allocation. However this is helped somewhat by the parallelism and the two-phase scheme.

2.4.2 Sapphire

Sapphire [34] is a concurrent replicating collector written for Java. Its main aim is to minimise the mutator pause times. It avoids pausing all threads during the *flip* phase of the replicating algorithm and it is implemented with a write barrier instead of a read barrier.

Sapphire defines several distinct memory regions for the heap:

- **U** region called "uncollected" memory, which will not be collected in this collection cycle.
- **C**, a region of memory which is called "collected", the memory currently being collected in the current GC cycle. C is further split into regions:
  - **O** (old space) essentially the same as from-space.
  - **N** (new space) essentially the same asto-space.
- **S** region for each threads stack.

The algorithm begins with a mark-and-copy phase. A tracer first traverses the memory graph to discover live objects in O. All live objects are copied into N. Note that this is all done concurrently with the mutator, different from Baker's algorithm. It differs from Cheng's collector in that instead of "double allocation" the to and from space objects are *loosely* synchronised. This means that any changes made to the copy in O space between two synchronisation points will be passed to the N copy before passing the second synchronisation point. This is done by utilising a write barrier and exploits the JVM memory synchronisation rules. After this, the flip phase is started.

The flip removes any pointers to O objects that threads may see. It is split into several smaller phases: Pre-Flip, Heap-Flip, Thread-Flip and Post-Flip. The first phase starts a write barrier to check any objects in U or N do not point to O in order to keep up an invariant that no object in U or N point to objects in O; this is then done in the Heap-Flip phase. The next phase (Thread-Flip) changes all the O pointers in the thread's stack to point to N objects; this is relatively easy to do as the write barrier ensures that the invariant is upheld. Finally the Post-Flip phase does clean up, by freeing of O space and reverting the write barrier back to the original.

This collector employs two separate write barriers. First during normal execution it uses an eager write barrier, which updates N region objects with their O objects updates as soon as they happen. Second it employs a different barrier in the flip phases to ensure no object in N or U space point to something in O space.

The Sapphire collector has an excellent way of incrementally doing the flip phase of a collection, overcoming the main issue in the Nettles et al collector. It also has an eager write barrier making changes to copied objects straight away rather than logging them and doing it later. It also does not use the to-space invariant which other replicating and copy collector algorithms have employed. There is an issue with root-set calculation phase which

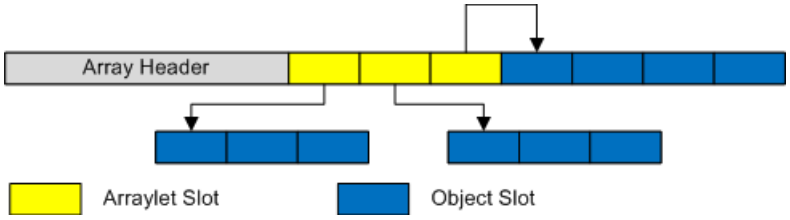


Figure 2.7: Picture demonstrating arraylets

is done in one go impacting on its real-time bounds. As well as this it has several features that require it to be implemented for Java or a language with features similar to Java i.e. with memory synchronisation. Whilst this is a nice idea it restricts the languages for which this collector can be implemented.

2.4.3 Metronome

The Metronome [10] created by Bacon, Cheng and Rajan differs from other collectors in that it was aimed at Java embedded real-time systems. It also differs from other collectors in that it is only one of two that have a true claim to being "real-time", although only soft "real-time". This is due to it requiring the user to specify the maximum live memory usage and the average allocation rate over a collection interval.

The collector has a two plans in one approach. It normally acts as a mark-sweep collector, switching to a copy collector when memory becomes fragmented. The collector as a whole is incremental so it uses an incremental mark-sweep similar to Yusa's snapshot-at-the-beginning algorithm [51]. It utilises the Yusa write barrier to maintain the weak tri-colour invariant. The copying portion of the collector is a Brooks-style collector and as such also uses a Brooks-style read barrier to maintain the to-space invariant.

The scheduling is either time or work based. A comparison of the two show that work based scheduling can tend to cause poor mutator utilisation. Interestingly their results for time based scheduling indicate that the problem with space explosion is somewhat alleviated due to the way the collector utilises parameters to control the collector to keep it within space bounds.

In order to maintain its real time claim, it uses the parameters listed at the beginning of this section. The user specifies these and as such the collector will not perform correctly if these are badly specified. In order to get these parameters correct the user may have to do some experimentation. They also utilise arraylets which are a way to break arrays into small fixed sized sections demonstrated in Figure 2.7. However they seem not to implement stacklets, opting for a snapshot-at-the-beginning approach. Thus, when dealing with program stacks which incur a large number of pushes and pops the collector may break its real-time bounds during root set evacuation. Hence it is only soft "real-time" and not hard "real-time". The reason for the lack of stacklets is because they do not scale to large number of threads well. This is counteracted with a weakened snapshot-at-the-beginning property. This does not allow any reference from the stack to "escape" to the heap, by it being logged. This is a combined Yuasa and Dijkstra-style write barrier.

2.4.4 Stopless, Chicken and Clover

Stopless [43] is a concurrent collector which is lock-free and parallel. It uses field level locking in order to do this. Pizlo et al. then went on to write Chicken and Clover [44] which are extensions of Stopless. Chicken uses a chicken write barrier and clover uses a statistical method to locking. Chicken and Clover are less complex and as such have a better lower and average case performance. This was traded for the lock-free abilities of Stopless and having a greater worse case performance. All three are written for the Bartok compiler[11], in its language of C#. They employ a concurrent mark-sweep algorithm with a copying compaction thread if necessary. The architecture of the collectors as a whole work so that a collector thread and compaction thread must run on separate CPUs to that of mutator threads. This enables collection, compaction and mutation to all occur at the same time. The collectors only differ in the compaction thread implementation.

The mark-sweep collector used is based on the Doligez-Leroy-Gonthier(DLG) collector [27, 26, 25, 24]. It uses a lock-free virtual mark-stack, a lock-free work stealing mechanism and a non-blocking mechanism to determine the termination condition. The first problem it overcomes is the issues with the mark-stack overflowing. It does this by putting an extra header word into each object. This word is used for the mark bit and a forwarding pointer used to create a linked list of grey objects to be forwarded. Due to the allocation of these pointers being atomic, using a CAS[12] operation to simultaneously mark an object and modify its pointer, means that these can be stolen and modified in a lock-free manner. This means it is also possible to have many collector threads, each stealing grey object from other collectors lists, when their own is exhausted.

2.4.4.1 Stopless

The collectors differ in the compactor, named CoCo. This works as a partial compactor for this collector although it is noted it could be used to be a full compaction algorithm like that used in the Compressor [37]. Objects that are to be moved must first be tagged. The collector thread does this by setting a bit in the object header and adding it to a list accessible to CoCo. CoCo then begins to copy the object to to-space. However it does this in a lock-free manner by first utilising a read barrier with a cloning mechanism so its cost is zero when the compactor is not running. It then forwards the from-space object to a "wide" to-space version of itself. This wide version is larger than the original because each field has an extra word header assigned to it called the status field. The read barrier checks this status field to ensure it is reading the most up to date copy of the field. A write barrier must also be employed so to make sure all object writes are only to the latest data and ensures writes are not lost through the use of a CAS operation. After each field is copied into the wide object a final to-space copy is made. At this point there are three versions of the object: the old one, the wide one and the new to-space one. Now the wide objects fields are copied to the to-space object and after this the forwarding pointer from the from-space object pointed towards the to-space object. Note that the barriers are still in place to ensure the most up to date version is being used.

The way barriers are controlled is interesting. It at first determines what phase the collector is currently in and dependent on the phase it either goes down the fast path (does nothing) or slow path (runs the barrier code). This is further optimised by removing the

[11] Compiler for C#
[12] Compare and swap

barrier code completely during phases it is not needed, done via cloning. They do this using the methods of [7] which enables two pieces of code to be in one place, in this case one that does nothing and one with the barrier. There is then the issue of having the code in the right place at the right phase. This is done by using GC safe points.

2.4.4.2 Chicken

The compression outlined above with the wide object is specific to Stopless. In Chicken objects are first copied in their entirety and thus the read barrier is an unconditional Brooks style read and write barrier. It works on the assumption that it is very unlikely that the mutator will change an object whilst it is being copied and as such, plays "chicken" with the mutator. However collisions can occur and in order to combat this, the write barrier is modified to create an "aborting" write barrier. In this barrier if the object is currently being copied when a write takes place the copy is aborted through a CAS operation so that a copy is not aborted twice. This means that objects who's copy is aborted will not be copied in the current compaction phase. A final step must be performed in order complete collection; a "heap fixup" phase. Here one more mark phase must be completed in order to make sure no objects in to-space point to objects in from-space. In order to do that an eager read barrier and logging write barrier similar to that of the Metronome collector must be employed.

2.4.4.3 Clover

Clover ensures that objects will always be copied. This is counteracted by the fact that the worst case scenario the collector is not lock free. The paper claims that this will occur with "negligible" probability. It does this by randomly generating a value, α, and assumes that this place in memory will never be written to. It then uses this alpha to block access to fields that are on original versions of the object. It is noted that the chance of this occurring in a 32 bit system is 1 in 2^{32} and in a 64 bit system 1 in 2^{64}. However to ensure 100% correctness the write barrier checks to see if there is a write to this place in memory and if so acts accordingly. However this will cause the mutator to block and thus lock freedom is broken.

Overall these set of collectors work very well. The worse case pause time in experimentation of Stopless, Chicken and Clover was 5ms, 3ms and 1ms respectively. The use of multiple barriers at different points of execution seems to be key in order for these all to work.

2.5 Java

Java is an object orientated programming language whose design goals where to be simple, robust, architecture neutral and portable and high performance [2]. The Java Virtual Machine (JVM) is how Java gains it's portability. It allows for bytecode created by compiling Java code on any machine to be interpreted by machine specific VMs meaning that code made on one computer will run on all computer with Java. The language also has garbage collectors which allow for automatic memory reclamation.

Java bytecode instructions are interpreted by a program thread, each thread having its own private stack. The stack consists of several stack frames. A new stack frame is pushed onto the stack whenever a method is invoked.

- **Operand Stack:** Used as an intermediate place to store results of bytecode operations. Most bytecodes interact with the operand stack in some way.
- **Local variable array:** Stores values which are method specific. The length is precalculated at compile time. This holds the parameters of the method and at index 0 the callee reference is stored (return reference). Things are stored in the local variable array from the operand stack with astore bytecode operations and things are loaded from the local variables and placed on the operand stack with an aload function.
- **Constant pool reference:** The constant pool is a per class/interface representation of the constant pool defined in the Java class file.

2.6 Jikes

The Jikes RVM[13] is an open source implementation of a JVM. It came from the Jalepeno [5] project started at IBM which they subsequently open sourced and it became Jikes. The most interesting thing about Jikes is that it is written in Java. In order for this to be possible a boot image of the VM is created and then loaded at runtime. Secondly Jikes does not have a bytecode interpreter instead it opts to compile everything into native machine code before execution. There are three versions of this compiler: Baseline, JNI and Optimising.

The Jikes project is well documented and due to the abstractions Java has it allows for easier implementation. Thus it has a high adoption rate from the research community. There have been lots of changes to Jikes over the years; most notable for this project is the introduction of the Memory Management Tool Kit (MMTk).

Jikes is a complex system and for this project we will mainly be editing MMTk code. The main part of Jikes communicates to MMTk via a set of predefined interfaces. The main ways it does this is when an object is to be allocated it calls an alloc method defined by each collector. This then uses a method which returns a boolean to see if a collection cycle is needed.

2.6.1 MMTk

MMTk [14] is a framework for building garbage collectors and allows programmers to create high performance memory managers. The main goals are to be flexible and offer comparable performance with the monolithic collectors built for Jikes. Currently there are already several collectors built with the MMTk these include the classic three collectors (mark-sweep,reference counting and copying) and generational/hybrid versions of them. It also has the IMMIX collector written by Blackburn et al [16]. There has also been several other notable collectors written using the MMTk: The Metronome [10], MC^2 [45] and the sliding views collector [8], although these have not been contributed back to the project, or have not been maintained. Therefore, they do not ship with Jikes.

The MMTk utilises several clever methods in order to attain its high level design goals. First it makes use of *Compiler Pragmas*. These are implemented in the Jikes compiler and allow programmers to control things such as inlining and interruptibility. It also identifies "hot" and "cold" paths and employs lightweight and efficient mechanisms for frequently executed paths (the "hot path"). It also makes heavy use of garbage collector design patterns; first outlined by Yeates et al. [50].

[13] Research Virtual Machine

The MMTk has what is known as a *plan* for each collector. Each plan tells the RVM how the collector should be run and is split into several classes:

- **Global class:** This class is the main class of the collector. It holds information that is global to the plan such as spaces, traces and phase definitions. It also contains methods for calculating how many pages are left.
- **Collector class:** This class defines local information for each collector thread. The MMTk purposely abstracts this into a separate class so that synchronisation is localized and explicit, thus hopefully minimized. This means that anything in this class is strictly local to each collector thread.
- **Mutator class:** This class contains all the code linked to the mutator of the collector. Here methods to allocate new objects are implemented and read and write barriers. All information in this class is local to each mutator thread.
- **Constraints class:** This class has all the constraints for the collector. It is there to communicate to the host VM/Runtime any features of the selected plan that it needs to know for instance whether it needs a write barrier or how many header words each object has.
- **TraceLocal class:** This class is responsible for implementing the core functionality computing the transitive closure of the memory graph.

An important aspect to note about the way the MMTk works is the *phase stack*. Each step of the collector has a *phase* associated with it. For instance the root set is computed in the *ROOTS* phase. Each phase is split into several *simple* phases each of which has a name. Each of the simple phases is created for a specific part of the collection: global, for the global class of the plan, mutator, for the mutator class of the plan and collector, for the collector class of the plan. A set of simple phases can be joined together to create a *complex* phase. It is also possible to create a *place holder* phase which can be replaced by a collector at runtime.

There is a common underlying feature of most of the collectors in Jikes. As the collectors are based on a hierarchical inheritance structure they all stem from one plan known as Simple. In Simple each plan has several region which it must collect. These are:

- **SmallCodeSpace and LargeCodeSpace:** This is used to store code in memory
- **Non-moving space:** This is used to store objects that will not move. These include things such as the program stack.
- **VM Space:** This space holds all VM specific object such as the boot image of the VM.
- **Immortal space:** This is a place for all immortal objects that are allocated after the VM has finished booting.
- **Large object space:** This is where all large objects are stored.
- **Meta space:** All meta data that is used by the MMTk is allocated here.
- **Sanity space:** This space is used to store the sanity checker which ensures collections are correct. This is only used if sanity checking is enabled

Algorithm 2.1 Code demonstrating some features of Magic

```
1  public void getObjectClassName(Word word,Offset offset) {
2      Address address = word.toAddress();
3      ObjectReference objectReference = address.loadObjectReference(
           offset);
4      //increments the address by the offsest
5      //then loads to object reference at that point
6      Object object = objectReference.toObject();
7      Log.writeln("The objects name is:");
8      Log.write(object.getClass().toString());
9  }
```

Several of these are subject to collection when a GC cycle is run and they utilise a simple mark-sweep collector in order to be collected. However this would cause issues with some collectors. It is possible to bypass these spaces by ensuring that no writes occur to these spaces and making sure other classes of the plan do not call the super class that does things such as initiate the spaces and schedules them to be collected. This is demonstrated well in the reference counting plan. However Immortal, VM, Sanity and Meta space must be used as normal. As they are not subject to collection this will not affect the overall collection cycle of a collector. It should also be noted that all objects that would be allocated in non-moving space must not move even in GC phases.

Debugging collectors is somewhat easier than on other platforms with the MMTk. The main way this can be done is by using the MMTk test harness outlined in section 3.1.5.2. However this is often a misrepresentation of how the collector would work in the full Jikes collector. To debug in this state is more difficult having to rely on tools such as GDB [1] and Valgrind [4].

2.6.2 Magic

Garbage collection requires a lot of low level manipulation of memory. Since Jikes and the MMTk are in Java this is done in a novel way called Magic [29]. Magic defines a set of extensions to the Java language by adding system programming abilities such as memory management to take place. In order to do this it requires support from the JIT compiler and the JVM it is running in. There are a few main types that a programmer needs to know about when dealing with magic: Word, Address, Offset, Extent and object reference.

A Word is a word sized section of memory which is architecture specific and changes dependent on the machine. The Address type holds a memory address. The Offset type is a memory offset with this it is possible to do things such and increasing an Address by a certain offset. The Extent type is used to represent a length of memory. Finally the Object reference type represents a reference pointer to an object in memory. This can be turned into the object itself. Object references can also be turned into Address values. An algorithm demonstrating some of the features of Magic can be seen in Algorithm 2.1.

Magic allows developers to utilise high level abstractions for low level languages. This is part of the reason why it is possible to debug Garbage Collectors directly in IDEs such as Eclipse.

2.6.3 The Jikes Compilers

Jikes speed comes from its ability to turn bytecode into native code and heavily optimise it. The Baseline compiler in Jikes favors correctness over speed. In fact the optimising compiler has around a 5% success rate compared to the Baseline version.

2.6.3.1 Baseline

The baseline compilers aim is to be correct. As such it is very desirable to use with test versions of the VM. It is used to dynamically load other compilers and thus is always a part of the Boot image. It is simpler than the optimising compiler as it maps bytecode directly to machine specific assembly.

2.6.3.2 Optimising

The optimising compiler gives a efficiency increase to the VM and is used in the production VM. It breaks down Bytecode into three intermediate representations (IR) before converting to assembly code. A different level of optimisation is does at each IR representation:

- High-level IR (HIR): Pre HIR any magic 2.6.2which has been called is inserted in HIR form. Loop unrolling takes place as well as the Basic Blocks of the control flow graph are analysed so hot and cold paths can be highlighted. Subsequent to this the HIR is turned into static single assignment form (SSA). SSA turns the program into a form where each variable is assigned once. The SSA code is further optimsed and then reverted back to HIR. Here again the HIR is optimised and then converted into LIR.

- Low-level IR (LIR): The first stage of this optimisation is to coalesce moves, move loop invariants out of loops and common sub-expression elimination. Common sub-expression elimination is a barrier optimisation technique utilised by Cheng's collector 2.4.1. After this peephole optimisation occurs as well as a general simplification of all code. The LIR is then moved to Machine-level IR.

- Machine-level IR (MIR): Here architecture specific optimisations are applied. Register allocation occurs before MIR is converted into native machine code.

Chapter 3

Design and Implementation

3.1 Design and Implementation

In this section we detail the design and implementation of the elements we have created and highlight common elements in order to design and build a framework for future Jikes MMTk developers to utilise.

3.1.1 Indirection pointers

An indirection pointer is required to show the mutator where the most recent version of an object is, once it has been moved. This pointer is stored in the header of the object. On creation this pointer will point to the object itself. The indirection pointers requires an extension to the current Jikes header which currently consists of three parts:

- **Main object header:** This consists of several object specific information such as: the TIB Pointer, a HashCode, a Lock state, an Array Length.
- **Garbage collection header:** All information that the memory management system requires includes things such as a mark bit.
- **Misc header:** Used for experimental configurations and is easily expanded, typically used for profiling.

As we are working with the garbage collection subsystem, we extended the garbage collection header. Currently Jikes has a class, *ForwardingWord,* which utilises an indirection pointer. This was unsuitable for our needs as it modifies the object header in a way such that it causes issues to occur when a program is running outside of GC. We could not use this as we need the indirection pointer to run with an incremental collector, which will need to be active outside of GC.

The next issue is to ensure that all objects are created with the indirection pointer (to themselves initially). This was done via a postalloc method in the mutator of the GC. This missed all objects that were allocated in the bootimage 2.6.1. This was rectified by initialising the indirection pointer during the normal Java object header initialisation. All objects must go through this method including that in the bootimage.

Following this a brooks style read barrier was created in order to test that the indirection pointers where working correctly. This consists of dereference the pointer to the object we

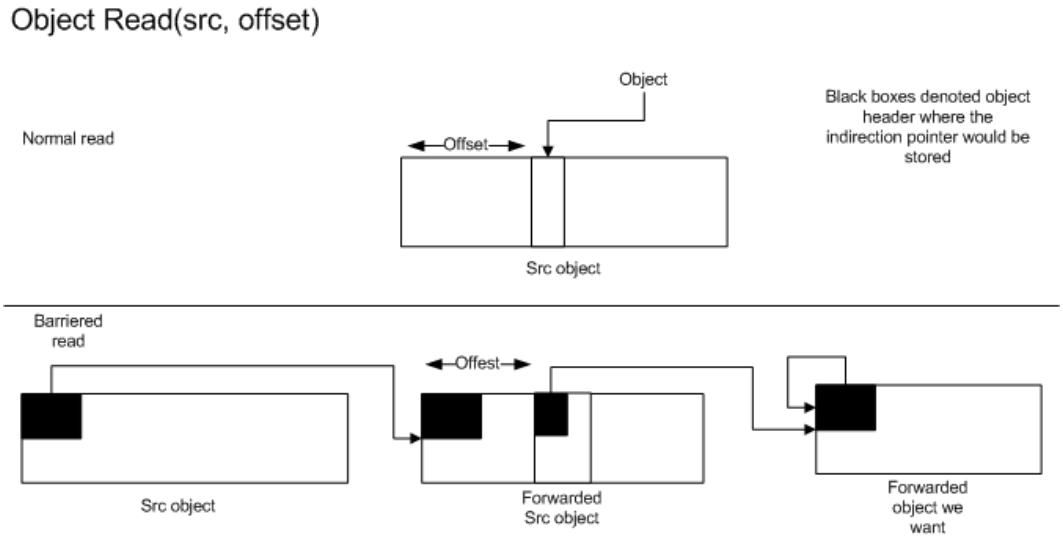


Figure 3.1: Picture demonstrating a standard read in Jikes (top) and a barriered read (bottom)

will read. Jikes object reads work by going to the object (source) where the object (target) that will be read is in. It then adds an offset to the head of source to find target and then does an *ObjectReference* load from this point. In turn our code had to first dereference the container object and then subsequently dereference the pointer to the object (see Figure 3.1).

3.1.2 Incremental Mark Sweep

Incremental mark sweep uses the original mark sweep algorithm interleaving the collector with the program. The overall execution flow of this and all the incremental collectors we made is outlined in Figure 3.2. Collection is broken down into 4 sections:

1. Collection is triggered for the first time in an incremental cycle. Housekeeping is done to prepare different spaces for collection. All roots are then marked. The collector then allows the mutator to resume.

2. After certain criteria are met, collection is re-triggered. Here the collector threads do a certain amount of scanning and ensure all dirty queues are cleared. If marking has finished then a flag *WORK_COMPLETE* is set.

3. The root set is rescanned. This is necessary as mutations to the stack and registers will not be caught by the write barrier employed. If the work is sufficiently low enough then the program marks everything needed and moves on to stage 4, otherwise it moves back to stage 2.

4. The collector sweeps all unmarked objects and does any post collection tidying; including handling soft, weak and phantom references and sending notification of collection termination to the mutators.

We will now go into some points mentioned above in further detail.

3.1.2.1 The write barrier

We used a slot based write barrier. Once a full collection cycle begins[1] this is turned on via the *collectionStarted* flag. The barrier stores any objects which are written into another objects field. There is also a second barrier on all new object allocations which are also stored. There are two mechanisms for storing the objects: with a sequential store buffer and in a linked list of objects.

The SSB is used to store object in a queue for processing later. It has two problems: 1. It can get large and use up undesirable amounts of heap space, 2. it has no way to determine which objects are already in it and ,as such, there is a risk of adding the same object twice. The second of these problems can be alleviated by the use of a mark bit to establish if an object is already in the SSB. This causes an extra bit in the header of all objects requires the barrier to do a checking step. In the implementation it was possible to avoid the overhead of the mark bit by using unused bits in the current object header.

The object linked list works by each object having an extra pointer-sized space in it's header. This is used for the linking in the link list. A pointer to the head of the list is stored. If an object is in the list then it will have its header pointer marked or be pointing to another object. This enables us to distinguish between object in the list and objects which are not. The method fixes the problems with the SSB however at the cost of the extra pointer in every object.

The object linked list requires the header word in the object to be left even if the object is no longer there. This is fine for object overwrites with a merge on the GC portion of the header. However when an object is set to null the header is destroyed. This creates a major issue with the object linked list method in that if an object is part of the linked list and it is set to null there will be a gap in the list. In order to alleviate this dummy objects are inserted on null writes. This meant that we had an extra overhead on the write barrier to catch null writes. In implementation we also had to edit the compiler to work with these dummy null objects. However this was unsuccessful as such we have moved dummy null objects to future work 4.2.3.1.

Write barriers are needed to track whether pointers have changed between incremental phases. To do this we log any modified objects and later recheck their pointers. The issue with this is that we create "floating garbage". Floating garbage are objects which are dead but were missed during collection. Figure 3.3 demonstrates the need for the write barrier and floating garbage.

3.1.2.2 Work calculations

How much work a collector should do in a work-based scheme is often difficult to decide. If too much is done then the pause time will be too long. If too little is done then the collector will not be able to keep up with the allocation rate of the program. We have adopted two different methods for comparison. The first is similar to the classical Baker work-based approach the second is a new time and work-based variation named work-time.

The number of objects which must be traced in a Baker collector, k, is:

$$k > \frac{L}{H - L}$$

[1] That is when the first cycle in a set of incremental collection cycles

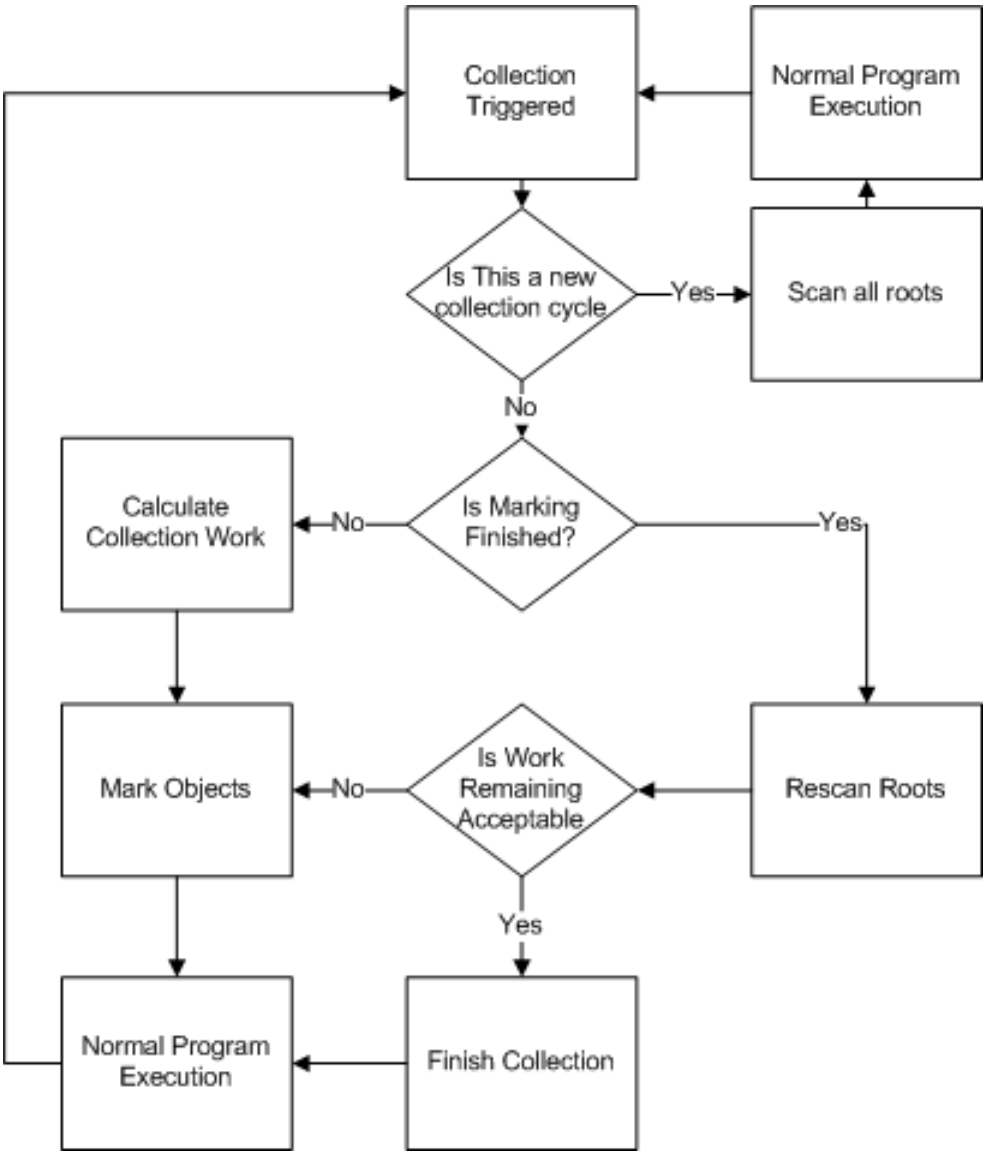


Figure 3.2: Image demonstrating garbage collection cycle of the Incremental Mark Sweep collector

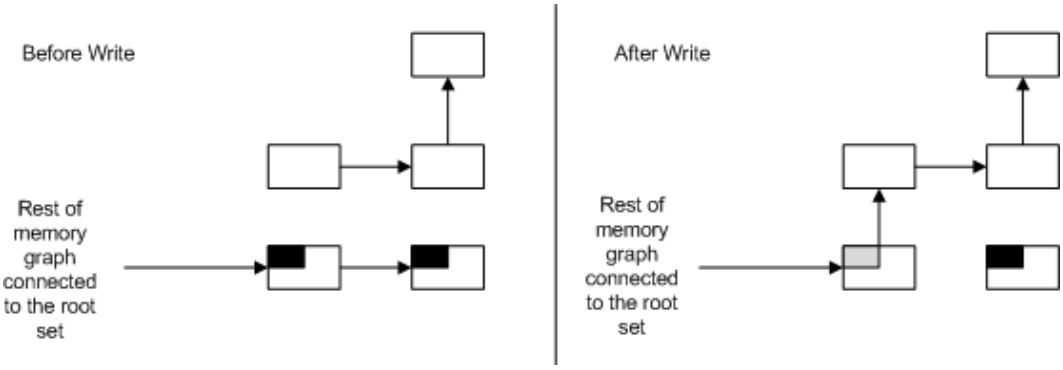


Figure 3.3: Image demonstrating the need for a write barrier. The squares represent objects the arrows represent links the objects have to other object and the colour in the top left of the squares demonstrates their current colour according to the tri-colour invariant are.

However, in the original Baker collector there were very different barrier techniques involved. Baker utilised a conditional read barrier which under certain circumstances would cause the object to be scavenged by the barrier. In the incremental mark-sweep collector we have a barrier that increases the workload on the collector threads. As such we must take this into account when scheduling. We also have an extra work cost in that all newly allocated object will also need to be scavenged. In the work-based approach we allow for k new allocations at each incremental step. The extra work incurred by the write barrier is calculated using a count, *W,* in the write barrier which is incremented each time the barrier is hit. In order to create smoother collection cycles an average of W is taken by dividing W by the number of incremental cycles so far, *N*. So the amount of work we must do at each incremental step is:

$$2k + \frac{W}{N}$$

The problem is that, in practice k is often very small giving the mutator very little time to do any work between incremental cycles. As such we have introduce an incremental trace rate multiplier, *M,* which allows the user to amplify k meaning fewer incremental collection cycles and better MMU. However it should be noted that the larger the larger the incremental grain size and thus fewer incremental cycles will occur but we will spend longer in garbage collection.

The second method requires more analysis but tries to maximise the MMU. In this scheme the user specifies the minimum time between each incremental cycle, *T*. It should be noted that this time is not strictly adhered to. Instead the collector will trigger at the first allocation after T time. As many allocations can occur between two time points the tracing rate must be modified by this amount. The number of allocations, A, is monitored. Again we have to factor in write barrier hits, *W*. The issue is that these values may be very large for one incremental cycle and very small for the next creating very "lumpy" pause times. Therefore we take average rates giving:

$$k + \frac{A}{N} + \frac{W}{N}$$

Obviously, substantial issue with this is we are now using estimates and, as such, the formula can become inaccurate. To alleviate this we pre-compute an estimate of the number of incremental cycles we believe we will need in order to complete collection, R. If this number of incremental rounds occurs we begin increase the tracing rate by 5% for each round above R. We found that a 5% increase was generally enough to stop a forced collection in practice. If we are about to completely run out of heap space we will force a full collection. R is computed as follows:

$$R = \frac{L}{T}$$

where L = estimated total number of live objects and
T = Current estimated tracing rate

3.1.2.3 Triggering mechanisms

The triggering mechanism for the collector is in two phases: *pre collection* and *during collection*. Pre collection as the name suggests occurs before a full collection cycle has begun and during collection is the triggering mechanism between incremental cycles. In pre collection, stage 1 of the collection cycle is triggered when the heap occupancy goes over a certain limit. By default this is at 78%, matching with figures established by Yusa [51]. Note that these experiments where not undertaken inn the context of a Java VM. In practice however they gave reasonable results. Due to this the user is able to tweak this value if they wish.

In *during collection* there are two different sets of criteria depending on which scheduling mechanism is used. The standard one is the work based version. In this collection is triggered after k allocations have occurred. The second triggering mechanism utilises a time specified by the user. This time is added to the time when the collector finished its last incremental step and the collector is not allowed to retrigger until this time limit has been reached. The intention of the time based approach is to keep MMU high but at the same time ensure robustness.

3.1.2.4 The final phase

Before the final part of collection (Phase 4) of the collection begins the collector must determine whether there is sufficient work such that it will not break real-time bounds. As re-scanning the roots for new roots is expensive we want this to occur as few times as possible but at the same time keep the pause times low. We have taken the original formula for calculating the amount of work to do. However due to the costly nature of rescanning the root set we multiply the amount of work to do by a factor, F, which is incremented each time an rescan attempt is made.

3.1.2.5 Incremental Parallel Mark Sweep

When making a parallel collector we must consider how to allow multiple threads to do the tracing work at the same time. In order to do this we require the tracing algorithm to be multi-thread safe and also ensure that each thread does a similar amount of work. The mark-sweep algorithm is naturally multi-thread safe as if multiple threads try to mark the same object at the same time nothing will happen, the object will just become marked. The work balancing is an interesting factor in a parallel collector.

Jikes implements the work sharing with an unsynchronised central work stack. In our future work section we propose a new method of work sharing with each collector thread having its own personal trace deque 4.2.3.1.

We also have to consider the work each processor will do. We have two options. We can either lower the pause times for each incremental cycle by dividing the work calculation by the number of collector threads, P. Alternatively we can reduce the number of incremental sections we need to do by allowing each processor to do the same given amount of work. We have opted for the second approach as we have found that the longest pause times are caused by step 3/43.1.2. This means we need to change our estimate of the number of rounds of work we will need to do to:

$$R = \frac{L}{T \times P}$$

3.1.3 Brooks Style Incremental Copy Collector

The copying collector employs the same execution flow as the mark-sweep collector. It has all the features of a classical semi-space collector so we used the standard semi-space collector as a base for development. There were several components which needed to be tackled in order to get this into a Brooks style collector. These concern the tracing algorithm, barrier code and the integration of all the incremental features from the incremental mark-sweep collector.

3.1.3.1 Tracing algorithm

The tracing algorithm had to be modified to work with the new indirection pointers we had created 3.1.1. The base algorithm works with the forwarding pointer class. This had a method that was able to distinguish whether an object was forwarded or not. This was done with a bit mask on the forwarding address. Instead of doing this we opted for the single threaded version to have a mark-bit in the header. This was marked by the collector thread when the object had been forwarded successfully. This did not add extra overhead to the object header as there were several bits that where unused which we were able to engineer to use for our purpose. The performance implications of this would be that this is a much simpler system that of the bitmask system and thus should be slightly faster. However it would not be suitable for parallel collectors as there was no direct synchronisation over this mark bit. The new tracing algorithm is shown in algorithm 3.2.

3.1.3.2 Barriers

Similar to the Brooks collector we use both a write and a read barrier to ensure the weak tri colour invariant.

Write Barrier The write barrier is different to the original barrier employed by Brooks. Brook's original scheme coloured object hit by the write barrier black. This meant that if the object was currently in from-space (white) it would have to be moved into to-space and then all its children traced. We felt that this was an unnecessary cost for the write barrier to incur and so used a similar write barrier to our incremental mark sweep collector. This colours the objects grey, ready to be moved. Their children are scanned during a collection cycle and not at while the mutator is running (like the Brooks write barrier). The

Algorithm 3.1 Algorithm demonstrating the parallel incremental copying tracing algorithm

```
1    public ObjectReference traceObject(TransitiveClosure trace,
         ObjectReference object, int allocator) {
2      /* If the object in question is already in to-space, then do
           nothing */
3      if (!fromSpace) return object;
4      if (isToggleBit(object)) {
5        /* Now extract the object reference from the forwarding word
             and return it */
6        return getForwardingPointer(object);
7      } else {
8        ObjectReference newObject = VM.objectModel.copy(object,
             allocator);
9        /* Set the forwarding pointer and the toggle bit of the
             object */
10       setForwardingPointer(object, newObject);
11       setToggleBit(object, (byte) 1);
12       trace.processNode(newObject); // Scan it later
13       return newObject;
14     }
15   }
```

consequence of this is that the collector threads will have to do more work than the collector threads of the Brooks collector. We feel that due to the time saved by not offloading this to the mutator this is justified.

There is a second reason for this write barrier. Due to the multiple space paradigm adopted by Jikes that we could only modify so much. We would have only been able to remove: large object space and large code space. The other spaces use a mark sweep style algorithm for collection. As this is the same write barrier we made for the incremental mark-sweep collector (3.1.2.1) we were able to collect the Jikes spaces incrementally as well. This is desirable as it is not possible to just collect one space at one time as pointers between spaces can point into into other spaces.

Read Barrier The read barrier is, in theory, the same as specified in the original Brooks collector. It works by unconditionally dereferencing the indirection pointer which is stored in each object. In practice, however, it was not possible to implement it this way. The first problem is that null objects do not have a header and so have no indirection pointer. This meant that if we tried to dereference a null object this would fail. Thus we had to put a null check in the barrier. This simply checked if the object the program wanted to read was null and if so did not attempt to do a dereference. The second issue was discussed before with the Jikes implementation of read barriers which requires two pointer dereferences; one to find the object containing the actual object we wish to read and then a second to actually read the object itself.

The read barrier for Jikes did not provide 100% coverage of all objects. It did cover the program heap correctly. However objects could escape from the heap onto the stack. This would mean we had a pointer on the stack which could potentially point to an old object.

There were several solutions to this outlined in Section 3.1.3.3.

3.1.3.3 Fixing the read barrier

There are three alternative solutions to the read barrier issue, with objects escaping to the stack. First was to make a way to incrementally scan the stack. This would mean we would have to monitor the stack for mutations and then rescan and redirect these as they happened. Alternatively we could try to catch objects before they got to the stack and eagerly redirect them. The third way would be to put a partial read barrier over the stack. The first thing we did before attempting to is ensure that the problem lay with the stack. This was done by creating a *complex incremental phase* (see 2.6.1). This is similar to the incremental phase except that now multiple elements could be added to it. When the elements of the complex incremental phase where finished a check would be done on a boolean to see if the incremental phase could be discarded. The reason for the complex incremental phase was to allow us to rescan all the roots after each incremental scan step. This indeed showed that the issue lay in the stack.

We opted to try to catch objects before they entered the stack and then eagerly forward them. The main reason for this is we felt that it would be able to be completed during the remaining time and in theory should be faster than putting a read barrier over the stack. However on attempting to do this we had issues capturing objects escaping to the stack through Magic 2.6.2 calls. In fact on injecting code into Magic to eagerly forward object we created more errors than we solved.

We therefore elected for approach 3; put a partial barrier over the stack. This was created only for the optimising compiler. This was done by inserting HIR (see Section 2.6.3.2) during the transition from bytecode to HIR. The reason for this is that it is the most natural place to hijack specific bytecode operations and we would gain all the optimisations offered by the Jikes optimising compiler. The opcodes which were chosen to have barrier code inserted into them were: *checkcast*, *invokeinterface*, *invokespecial*, *invokevirtual*, *if_acmpeq* and *if_acmpneq*. Code was inserted at the point where the object references where popped off the stack. See algorithm 3.2 for an example on the *if_acmp* bytecode.

The IR code we inserted was a direct call to the barrier method we had made through the MMTK. It would of been nicer to inline the code directly however we felt that as we inserted this at the top of the optimising compiler that it would be optimised enough. Had we put in the code ourselves we could of done things such as null check folding. This would have allowed our barrier to execute the null check implicitly whilst also running the barrier code itself.

3.1.3.4 Parallel Brooks Style Incremental Copy Collector

The parallelisation of the copying collector was not as trivial as the mark-sweep collector. The main issue lay in the setting of the indirection pointer during scavenging. In the single-threaded version we did not have to worry about multiple collector threads scavenging the same or interconnected objects simultaneously. This posed a major problem with indirection pointers which now had to be set atomically when they were changed during scavenging. Also, we had to ensure that no two collector threads tried to copy the same object at the same time.

We did this in a lock-free manner due to the overheads that would be incurred by locks. The first stage is at the beginning of the *traceObject* method. Here objects raced to gain the right to scavenge an object. This process worked by utilising the inbuilt CaS operations

Algorithm 3.2 Algorithm demonstrating the bytecode

```
1         /* Pop objects off the stack as this is an ifcmp we pop 2
              value */
2         Operand op1res = popRef();
3         Operand op0res = popRef();
4
5         /* Create temporary registers to hold results from barrier
              code */
6         Operand op1 = gc.temps.makeTemp(op1res.getType());
7         Operand op0 = gc.temps.makeTemp(op0res.getType());
8
9         /* Get the method name and the class name the bytecode
              belongs to */
10        String bcodeClass = gc.method.getDeclaringClass().toString()
              ;
11        String bcodesMethod = gc.method.getName().toString();
12
13        /* Check that bytecode is not a memeber of the following
              classes or methods */
14        /* This is to stop barrier code being inserted into barrier
              code creating an infinte loop */
15        if(!(bcodeClass.startsWith("org.mmtk.policy.Inc") ||
              bcodesMethod.equals("resolveInternal")
16              || bcodesMethod.equals("<init>") || bcodesMethod.equals
                    ("<clinit>")))
17        {
18            /* Create IR code to call barrier code */
19            appendInstruction(Call.create1(CALL, (RegisterOperand)
                  op1, IRTools.AC(Entrypoints.
                  getForwardingPointerNullCheckIfCmp.getOffset()),
20                        MethodOperand.STATIC(Entrypoints.
                              getForwardingPointerNullCheckIfCmp),
                              op1res));
21            appendInstruction(Call.create1(CALL,(RegisterOperand)
                  op0, IRTools.AC(Entrypoints.
                  getForwardingPointerNullCheckIfCmp.getOffset()),
22                        MethodOperand.STATIC(Entrypoints.
                              getForwardingPointerNullCheckIfCmp),
                              op0res));
23        }
24        else
25        {
26            op1 = op1res;
27            op0 = op0res;
28        }
```

that Jikes provides. This works on a pair-wise prepare and attempt method. The prepare gains the value of the address as signals to the VM that a critical section has been entered. The attempt method attempts to write a value to that address if the value given is the same as the original. This set is atomic. There were no address-level versions of these calls so these had to be implemented. Once this had occurred the collector would either return a zero address or the correct address. If the zero address was returned then the collector thread had lost the race and would have poll, waiting for a non-zero address to be written. The winner would then go on to forward the object itself. At which point both would return the location of the new object. Algorithm 3.3 demonstrates the parallel tracing algorithm.

3.1.4 Framework for incremental collectors

We now had two working collectors and were able to take common elements from both creating subclasses and a sub-plan. This would allow future garbage collection developers using Jikes and the MMTk to create their own incremental collectors with relative ease. A UML diagram of the plans are show in 3.4. We will now delve into each of the classes in more detail:

- **Incremental class:** Here all the core global elements for collectors are stored. This includes all the global variables for work calculation, the work calculation function, the scheduling code and the phase stack.
- **IncrementalCollector class:** Contains the *collect* method which is responsible for scheduling the phase stack and some accounting code. The incremental collection phases are also handled in this class.
- **IncrementalConstraints class:** All the constraints for an incremental collector.
- **IncrementalMutator class:** Contains the alloc and posalloc methods for incremental collectors which have the accounting code for allocations. This also ensures all new objects are marked grey. We have also put the common write barrier code in here. This is overridible if others do not wish to use it or use their own write barriers.
- **IncrementalTraceLocal:** This has the force retrace methods used when an object needs to be retraced when it is marked from black to grey.

These classes along with the incremental phases and incremental spaces we have created will enable user of the MMTk to develop their own incremental collector variants.

3.1.5 Testing

To ensure our collectors work they were thoroughly tested. We also ensured that the regression tests of Jikes work so that we have not interfered with Jikes itself. It should be noted that for the changes made for the read barrier extensions for the copying collector the regression tests were compromised. Due to a check on generated HIR code from specific bytecodes.

3.1.5.1 Jikes

Jikes has a great test framework to enable developers to easily test and benchmark the changes they have made. It also has several different build levels which have different levels of features for testing outlined below:

Algorithm 3.3 Algorithm demonstrating the parallel incremental copying tracing algorithm

```
1    public ObjectReference traceObject(TransitiveClosure trace,
         ObjectReference object, int allocator) {
2      /* If the object in question is already in to-space, then do
           nothing */
3      if (!fromSpace) return object;
4
5      /* Try to forward the object */
6     ObjectReference obj = getForwardingPointerTryingForward(object)
         ;
7
8      if (obj.toAddress().isZero() || object.toAddress().NE(obj.
           toAddress())) {
9        /* Somebody else got to it first. */
10
11       /* We must wait (spin) if the object is not yet fully
             forwarded */
12
13       while (obj.toAddress().isZero())
14       {
15         obj = getForwardingPointerTryingForward(object);
16       }
17
18       /* Now extract the object reference from the forwarding word
            and return it */
19       return obj;
20     } else {
21       /* We are the designated copier, so forward it and enqueue
             it */
22       ObjectReference newObject = VM.objectModel.copy(object,
             allocator);
23       setForwardingPointer(object, newObject);
24       trace.processNode(newObject); // Scan it later
25       return newObject;
26     }
27   }
```

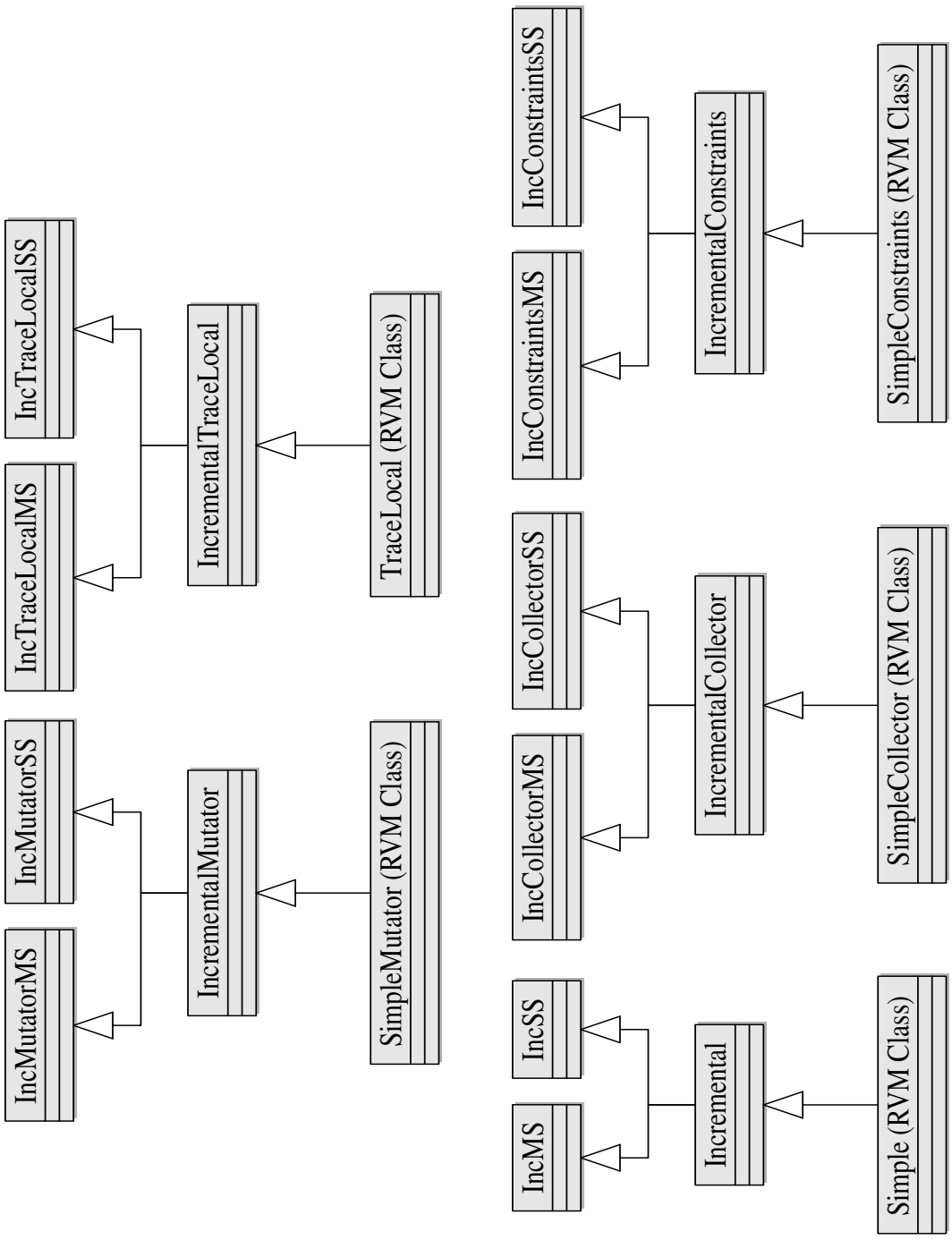


Figure 3.4: UML diagram of the new incremental garbage collectors

- **BaseBaseCollectorPlan**: A *.properties* file must be created in order to create new Base versions. To do this look at current examples. This build target is aimed at the testing of a garbage collector. It takes the shortest amount of time to build of the following plans, but in turn creates the slowest virtual machine and only has the garbage collector named. So for instance if we wanted to build the Base version of a *SemiSpace* collector our target would be *BaseBaseSemiSpace*. This is only for testing quick changes to the codebase and should not be relied upon to behave exactly the same as the other builds.

- **Prototype:** Similar to the BaseBaseCollector. However this contains all the collectors in this build. This will determine whether any other collectors have been broken from the changes made. This build should be used sparingly to test other collectors it is still not a good representation of the final RVM.

- **Prototype-Opt:** Again, like the previous two builds prototype-opt is designed to be a quick build. It is however still slower that other more advanced builds but contains more feature of Jikes than the Prototype build. The optimising compiler and the adaptive system are both here. This means the user will get a much better picture of if their code has broken either of these which makes probably the most usable for full system tests of the collector.

- **Development:** Here we get a much better performing RVM. However, the build time is much longer. It has all the Jikes features enabled and also assertions and debugging enabled. This means debugging and any errors will be found with this build and it should be used before every commit to the Jikes repository. This should be used at the end of every milestone.

- **Production:** This is the fastest build of the RVM. It is the same as the development build apart from with assertions and debugging turned off. This also takes quite a while to build but will be faster than the other builds. This should only be used for benchmarking and not for testing.

Jikes also provides a series of tests for users to run. These tests go through performance as well as correctness:

- **Core:** The core set of tests are functionality tests that ensure the correctness of Jikes. It runs its tests on the four main build targets (Prototype,Protoype-Opt,Development,Production)

- **Performance:** This runs a set of performance tests to benchmark Jikes. It runs SPECjvm98, SPECjbb2005 and the Dacapo benchmarks.

- **MMTk-unit-tests:** This runs a suite of unit tests to check it is still working correctly. This will enable us to get a quick overview of any issues we have caused with our changes we make to the MMTk.

- **Pre-commit:** This is meant to be run and passed before committing any code to Jikes repository. This test is quite extensive it runs against the prototype and the development build targets. It first runs a series of functionality tests, then it is also benchmarked with the Dacapo benchmark, finally it does a code style check. This test ensures that any code committed to the Jikes repository is up to standard.

3.1.5.2 MMTk test harness

The MMTk also includes a test harness. With this we can debug our collector code from within an IDE. We are also able to write specialised tests so that we can test for strange edge cases and such. It should be noted however that this is a controlled environment setup to just test the collector and the collector ran very differently in a real build.

Chapter 4

Evaluation

In this section we will evaluate the collectors we have made. First we will test the impact of the indirection pointers. We then benchmark the incremental mark-sweep collector in both parallel and non parallel modes. We also will assess the impact of our new parallel work load balancing scheme and our replacements for SSBs. Finally we will appraise the incremental copying collector with its read barrier extensions, again in parallel and non parallel mode.

4.1 Benchmarking

Benchmarking was performed on a machine with 2 Intel Xeon E5345 2.33 GHz Quad core processors with 8MB L2 cache and 8GB RAM with a 12GB swap file. The operating system is Ubuntu 9.04 server edition x64. The computer was not running X and had a network connection with all regular Linux programs installed. All benchmarks were run twice in a row and the second time taken. This is to give the VM time to "warm up" compiling and optimising all code. This was done as we want to test the GC, not the rest of the Jikes VM.

In order to reduce variation in runtime the pseudo adaptive driver for the Jikes RVM compiler was used. This applies compiler optimisations according to advice computed ahead of time. This avoids variations in methods that are regarded as "hot". Despite this there will always be a slight variation in run times. Thus, all experiments were conducted 10 times and averages taken.

We used the following benchmarks from the Dacapo suite [13]. Due to time constraints and failures in some of the benchmarks in Jikes (eclipse and chart) we opted to use the following benchmarks:

- antlr - parses one or more grammar files and generates a parser and lexical analyser for each
- bloat - performs a number of optimizations and analysis on Java bytecode files
- fop - takes an XSL-FO file, parses it and formats it, generating a PDF file
- jython - interprets a the pybench Python benchmark
- luindex - Uses lucene to indexes a set of documents; the works of Shakespeare and the King James Bible

- lusearch - Uses lucene to do a text search of keywords over a corpus of data comprising the works of Shakespeare and the King James Bible (multi threaded)
- pmd - analyses a set of Java classes for a range of source code problems
- xalan - transforms XML documents into HTML (multi-threaded)

The graphs shown are interesting samples showing trends and anomalies over all the benchmarks. Appendix B.1 shows all relevant graphs generated. There is also tabulated numerical data shown in Appendix A. We also should note that we have called the tracing rate multiplier k in the evaluation.

4.1.1 Indirection pointers

	antlr	bloat	fop	jython	luindex	lusearch	pmd	xalan
Non Read Barrier	5410	8121.6	1532.2	7453.2	9447.2	3074.4	5810.2	3530.8
Std Read Barrier	5465.6	10689	1660.8	9855.2	11237	28769	7922.2	42058.8
Stack Read Barrier	5989.4	17160.6	1930.6	13384	15210	32614	9966.6	60523
Std Read Barrier Ov.	1%	32%	8%	32%	19%	836%	36%	1091%
Stack Read Barrier Ov.	11%	111%	26%	80%	61%	961%	72%	1609%

Table 4.1: Table demonstrating end to end times for benchmarks and the overhead caused by the read and the stack read barrier

The results show a very interesting pattern occurring. The read barrier is very costly, in fact, the standard read barrier has a higher cost than indicated in the Blackburn et al. paper [15]. This is probably due to our unconditional read barrier not being fully unconditional due to need of a null check 3.1.1. The differences between the single threaded benchmarks can be explained due to a more reads in the bloat, jython and pmd benchmarks. This will result in more read barrier hits and thus higher overheads.

The overheads for the standard read barrier increase by a factor of 8-10 in the multi-threaded benchmarks (xalan, lusearch). This is somewhat strange and probably indicates a synchronisation issue with the barrier. However the code in the barrier is quite simple so there may be an underlying synchronisation issue with read barriers in Jikes. Unfortunately we have not had time to fully debug this issue and have not come to a full conclusion.

The Stack based read barrier is, as expected, worse than the standard read barrier. Now as the costs are imposed on object compares, casting and other operations there will be a lot more chances to cause overhead. On average there is 37.2% additional overhead over the single threaded benchmarks and a 118.4% increase over all benchmarks. The benchmarks increase with a range of 16 - 48% which follows from the benchmarks complexities.

4.1.2 Incremental Mark Sweep

We will now evaluate our incremental mark-sweep collector. We will be doing the following analysis outlined in 4.2.

Analysis	Desired outcome
1. Varying k analysing overhead, maximum pause time and average pause time	Investigating the effects of varying k on these three criteria and also comparisons between stop the world collector
2. Doing the same on the IncMS with a mark-bit	Investigating the effects on these criteria based on using a mark-bit
3. Analysing overhead, maximum pause time and average pause time for time-work based approach varying k for different times	Establishing the costs of the time-work based approach.
4. Analysing the pause time distribution and average mutator utilisation of the incremental mark-sweep both work and time-work based for "best" values of k and time.	Evaluate the potential for the approaches.
Analysing MMU for our work-time based approach.	Evaluating the effectiveness of time-work based approach on MMU.

Table 4.2: Table analysing the experiments for incremental mark-sweep

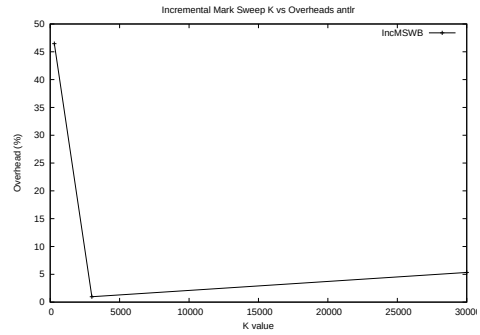
4.1.2.1 Experiment 1

The overhead figures (Figure 4.1) tend to follow a trend of decreasing as k is increased as seen in 4.1d. This is to be expected as when the incremental grain size is increased there will be less GC cycles and so the costs associated with switching between GC and the program are reduced. There is a large drop when increasing the grain size from 300 to 3000 this shows that a small increase in the grain size can cause the overhead to shift quickly. The question may be asked why do we not set k to infinity as the trend shows lower overheads for higher values of k. If we did this it would increase the grain size so much that our collector would work like a STW collector with a rescanning roots phase which would put overheads higher than that of a STW collector. This is demonstrated in the graphs for antlr 4.1a seeing an upward trend for k=30000.

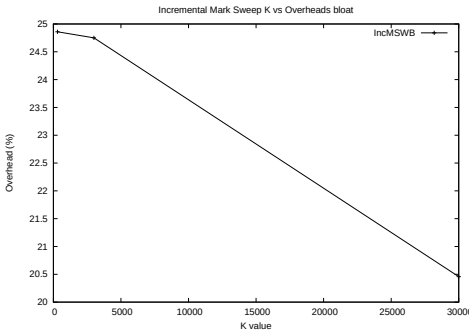
The jython figures 4.1c seem the most out of place of all the results with a large spike at k=3000. On investigation jython succumbs to a large issue with the logging write barrier, in that if a large number of writes occur between incremental cycles a large amount of extra scanning must be done.

Maximum pause times (Figure 4.2) are quite application dependent. These maximums are centered around the rescanning roots portion of collection. This is because currently there is no incremental scavenging of soft, weak and phantom references. This portion of collection will therefore take longer than the other portions. However our maximum pause time is significantly lower than that of the STW collector by an average of 52% for k = 3000. This is a good result as this is one of the main points of an incremental collector.

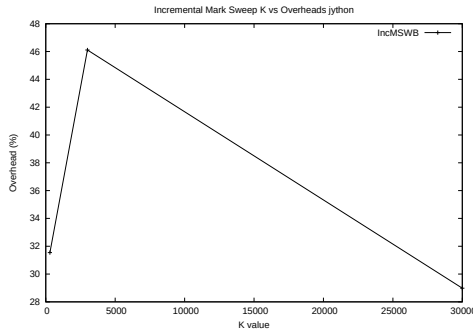
Figure 4.2a shows the general trend of the maximum pause time going down k is increased. This is to be expected as at greater ks the collector will do more work at each incremental step. This means there will be fewer chances for the write barrier being hit and allocations to occur. Therefore there will be a lower chance of a spiked pause time from a spike of writes or allocations. However 2 benchmarks follow the pattern in 4.21b. On closer inspection we see that at this k the write barrier is hit a lot between 2 incremental cycles



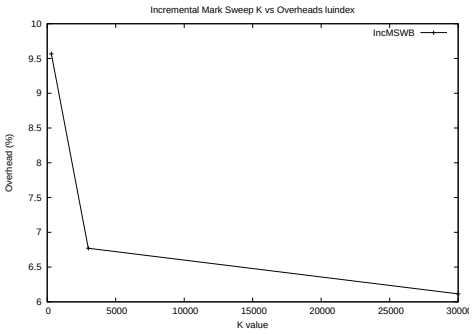
(a) antlr overheads for different k's



(b) bloat overheads for different k's

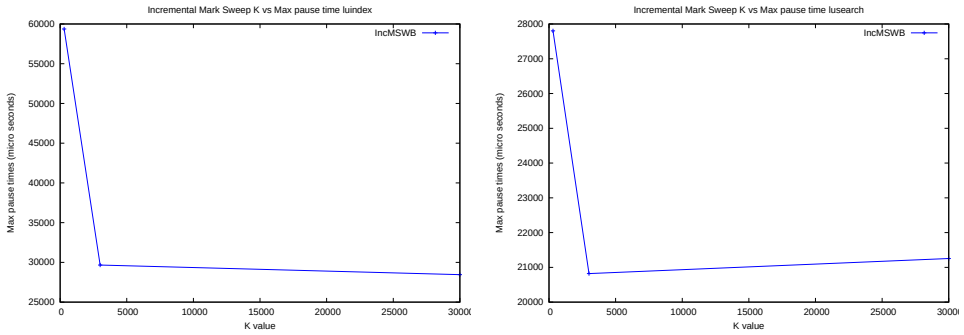


(c) jython overhead for different k's



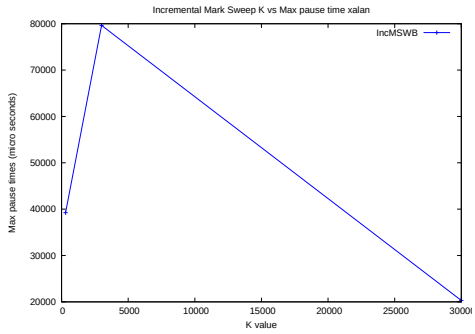
(d) luindex overhead for different k's

Figure 4.1: Results for overheads, Incremental Mark-Sweep on single core using the work-based approach



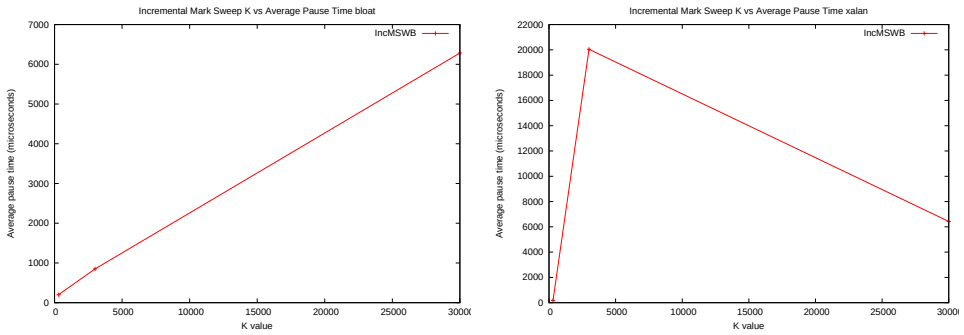
(a) luindex maximum pause times

(b) lusearch maximum pause times



(c) xalan maximum pause times

Figure 4.2: Results for maximum pause times, Incremental Mark-Sweep on single core using the work-based approach



(a) bloat average pause times

(b) xalan average pause times

Figure 4.3: Results for average pause times, Incremental Mark-Sweep on single core using the work-based approach

causing this peak. It should be noted however that the maximum pause time is still lower than that of the baseline garbage collector.

The average pause times (Figure 4.3) follow a tread of a higher average as k is increased. This follows from the fact that as k is increased the coarser incremental grain size becomes. This in turn will cause less incremental collections to occur and thus the average collection time to go up. These averages are significantly lower than the STW versions at an average of 2.8% of the original.

The outlier to this trend is xalan 4.3b. This is in line with the maximum pause time being very high for k = 3000. Again this is due to a large number of writes between incremental phases causing the amount of work needed to be done to spike. The use of a Mark-Bit should stop this.

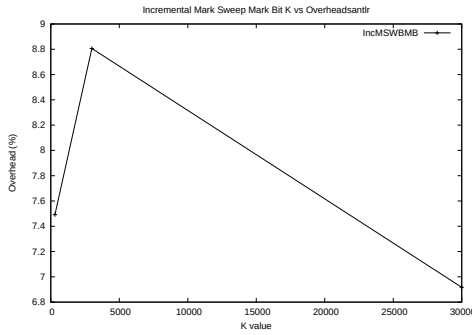
Conclusion The mark-sweep work-based approach enables us to shorten the average and maximum pause time significantly. However it does increase overhead compared with the baseline version but this is to be expected with "real-time" collectors. The best trade-off between overhead and pause times at k = 3000. An issue with this approach is it is not resiliently to sudden bursts of writes. This should be amended with the addition of a mark-bit.

4.1.2.2 Experiment 2

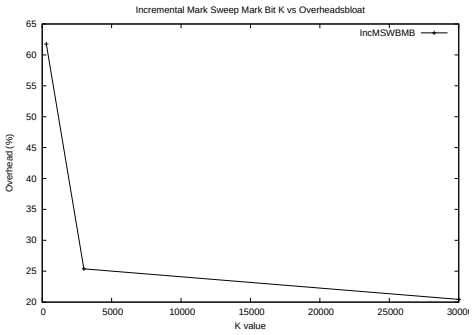
The total overheads (Figure 4.4) of including the mark-bit on the whole cause an increase in overheads. This is to be expected as due to the increased cost of the write barrier now having a check on the mark-bit.

The maximum pause times (Figure 4.5) from the mark bit versions are lower than the mark bit. This is to be expected as with the mark-bit there is a lower chance of a sudden burst of writes causing a pause time spike. An excellent example of this is shown for the xalan benchmark at k = 3000 (Figure 4.5d). For the original example (Figure 4.2c) there is a large spike a k = 3000. In the mark-bit version (Figure 4.5d) this spike has been removed.

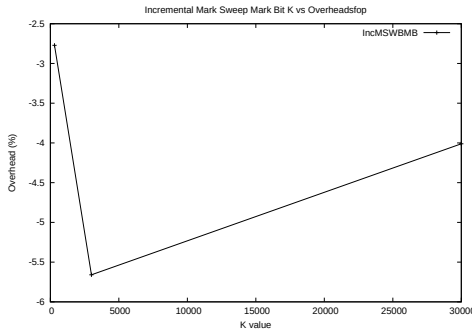
The average pause time is essentially the same with the exception of xalan (Figure 4.5d). Xalan we see a big drop at k = 3000 which is excellent as it shows that the mark-bit is doing



(a) antlr overheads for different k's

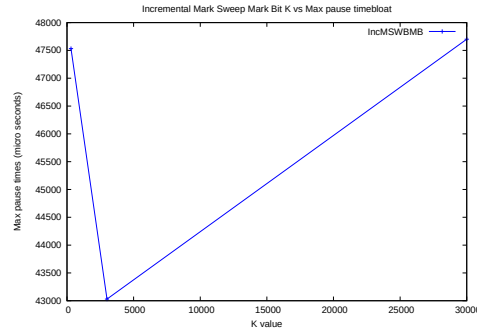


(b) bloat overheads for different k's

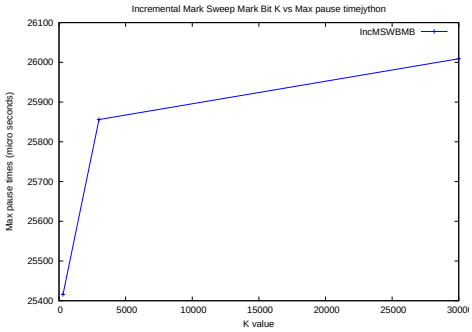


(c) fop overheads for different k's

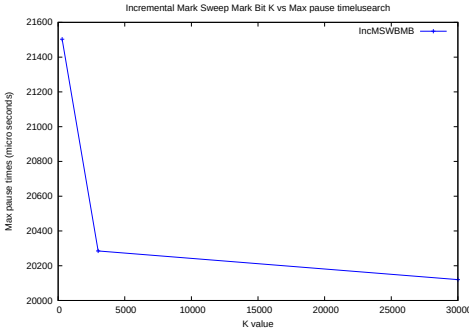
Figure 4.4: Results for overheads, Incremental Mark-Sweep with mark-bit on single core using the work-based approach



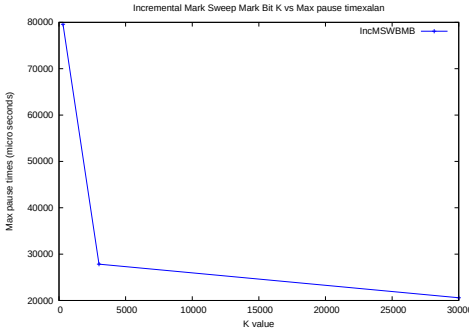
(a) bloat maximum pause times



(b) jython maximum pause times

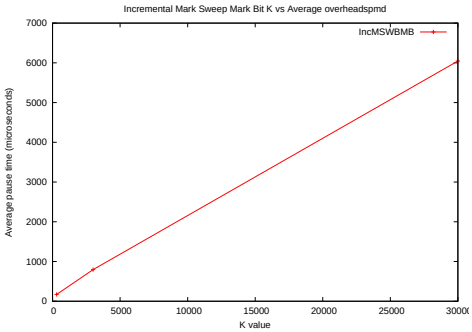


(c) lusearch maximum pause times

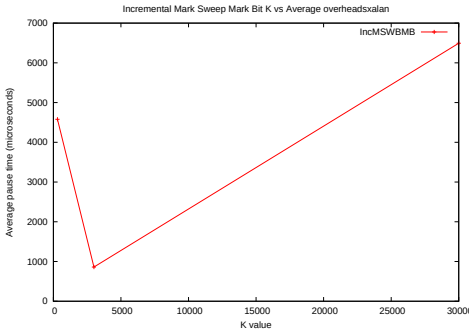


(d) xalan maximum pause times

Figure 4.5: Results for maximum pause times, Incremental Mark-Sweep with mark-bit on single core using the work-based approach

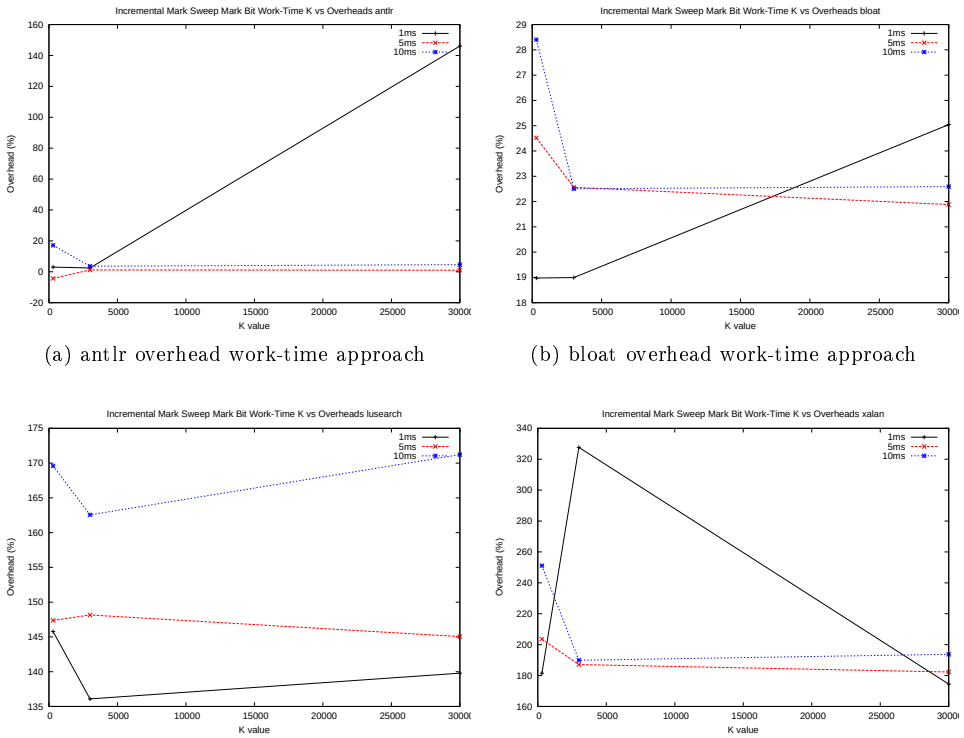


(a) pmd average pause times



(b) xalan average pause times

Figure 4.6: Results for average pause times, Incremental Mark-Sweep with mark-bit on single core using the work-based approach



(a) antlr overhead work-time approach

(b) bloat overhead work-time approach

(c) lusearch overhead work-time approach

(d) xalan overhead work-time approach

Figure 4.7: Results for overheads, Incremental Mark-Sweep on single core using the work-time approach

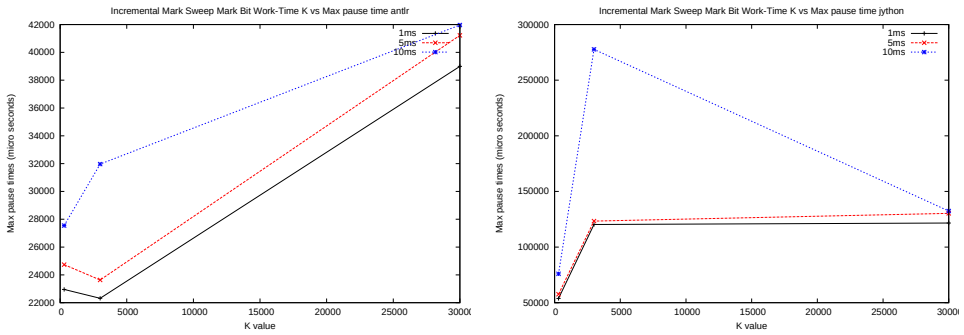
its job reducing the work incurred by the write barrier.

Conclusion These results show that using a mark-bit tends to lend itself to better results with respect to pause times. From now on the collectors will utilise the write barrier with a mark-bit unless otherwise stated.

4.1.2.3 Experiment 3

The overheads for work-time 4.7 tend to be quite random. This is to be expected as the work-time approach is very susceptible to bursty allocations and bursty writes. Changing k can have a better or a worse affect dependent on benchmark. This is due to the k value being more effective due to a more aggressive tracing rate algorithm. As such the effect of having a k that is too large much sooner than for the work-based approach.

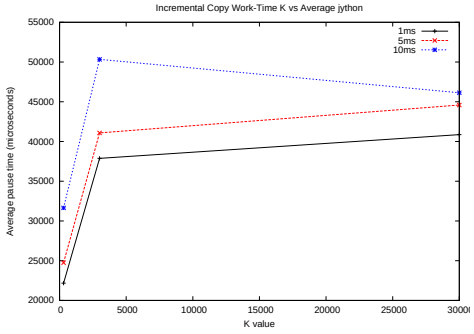
An average decrease in overheads is observed when compared to the standard work based approach. This is good as it shows that and increase in mutator utilisation shows a decrease in overheads. The only one where this does not follow is antlr. This is due to this being a very short running benchmark and an increase in mutator utilisation would not cause have much of an impact on pause times.



(a) antlr maximum pause times

(b) jython maximum pause times

Figure 4.8: Results for maximum pause times, Incremental Mark-Sweep with mark-bit on single core using the work-time approach



(a) jython maximum pause times

Figure 4.9: Results for average pause times, Incremental Mark-Sweep with mark-bit on single core using the work-time approach

The general case for maximum pause time shown in 4.8a shows an increase in k causes an increase in pause times. These pause times come from the stack rescanning phase which has little to do with k. We were unable to profile this issue and are unsure why it has occurred.

Generally the time periods stack, with the higher time at the top and the smaller at the bottom. This is due to a larger window occurring before the rescanning phase. As the maximum pause times lie here there generally is more work at this point due to this larger window. However in some situations 4.8b this does not happen. In this situation on further investigation we found that at k = 3000 for the 10ms time that a forced collection was incurred causing the max pause time to spike. This demonstrates that there is no set k and t configuration that will work for all applications and they need to be tuned appropriately.

The average pause time results (Figure 4.9) shows the pattern you would expect with average pause time increasing as k increases and as the time is increased. However the jump from k = 300 to k = 3000 is very large. This shows again that the tracing rate is more aggressive than the work-based approach.

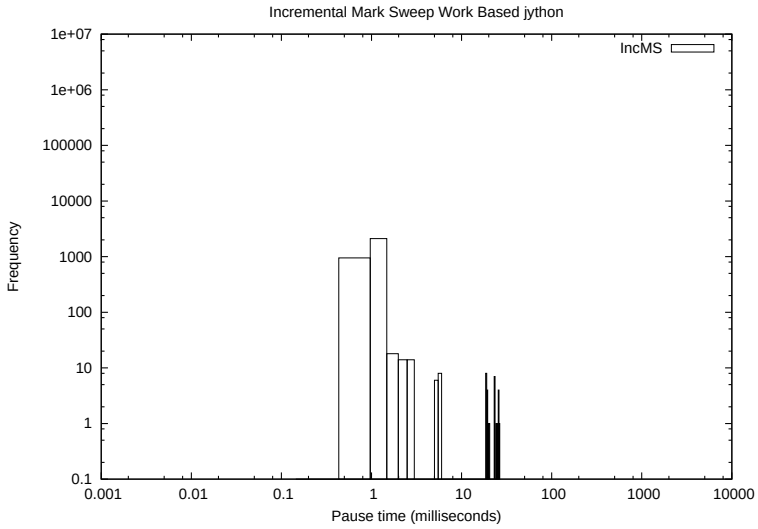


Figure 4.10: Pause time distribution for Incremental Mark-Sweep work-based k = 3000 (non omission)

Conclusion Overall that work-time increases the mutator utilisation and thus decreases overheads. However it is more susceptible to poorly configured k value causing poor performance. This approach is effected by "bursty" write and allocations more so than the work-based approach due to an increase in the possible window for this to occur.

4.1.2.4 Experiment 4

In this experiment we are trying to show the potential for the different approaches. We speculate on certain measures by modifying the data sets. To begin analysis of the pause time distribution of the incremental work based collector is undertaken.

In Figure 4.10 two groups of pause times have formed, one between 0.5 - 6 ms and one between 10 - 15 ms. The ones at the lower end will be the pause times caused by the incremental trace work and the ones at the top end will be the root set evacuation and the rescanning phase. Figure 4.11 shows the pause time distribution minus the root rescanning and the initial root evacuation phase. This will somewhat emulate the pause times which could be achieved if incremental stack scanning were implemented 4.2.3.1.

The omitted version shows that the pause times for this collector are significantly lower than that of standard STW Mark-Sweep. The graph also demonstrates that if we were to implement future work outlined in 4.2.3.1 it could be possible to have sub 10ms pause times. This is because with incremental stack scanning the cost of the initial scan would no longer be incurred.

Figure 4.12 shows an example of the average mutator utilisation. This is calculated by taking set window sizes and seeing how long the mutator was running for during that time span. This graph shows that the work based approach allows the mutator to run for short

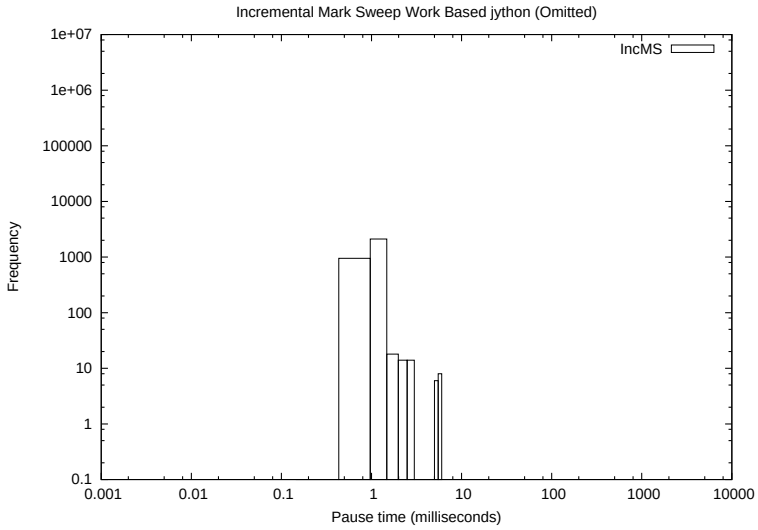


Figure 4.11: Pause time distribution for Incremental Mark-Sweep work-based k = 3000 (omission)

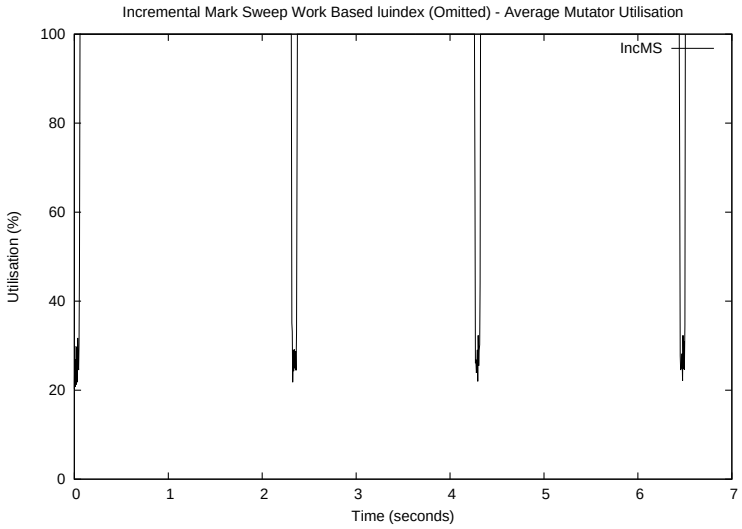


Figure 4.12: Average Mutator utilisation for Incremental Mark-Sweep work-based k = 3000 (omission) 5ms window

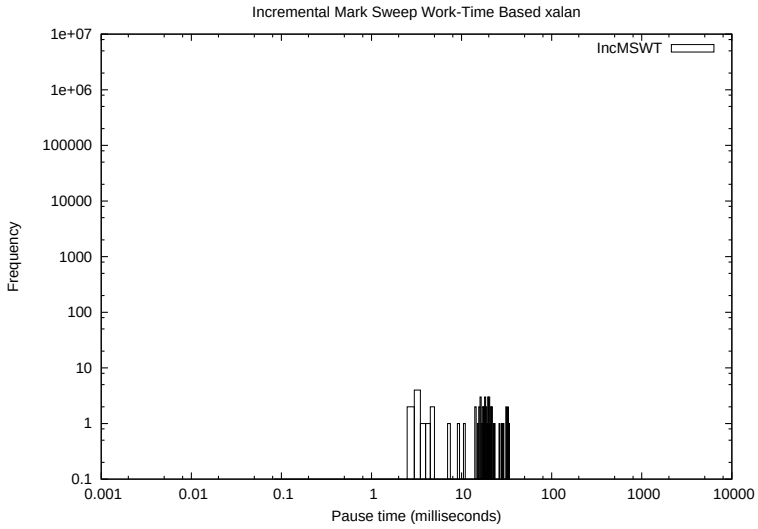


Figure 4.13: Pause time distribution for Incremental Mark-Sweep time-work based approach k = 300 t = 5ms (omission)

amount of times between incremental cycles demonstrating the incremental nature of this approach. This also shows when collections occur. Comparing this to the standard MS the incremental approach has better mutator utilisation.

Figure 4.13 shows that the work-time based version of the pause time distribution graph. Notice that this style of collector has higher pause times than the work based approach. This is due to the mutator having a set amount of time to do work between incremental cycles. There are quite a few times in excess of 10ms and these tend to be due to the work calculation not keeping up with the allocation rate. In Figure 4.14 there is an example of a benchmark having a forced collection causing a large pause time. This affect only occurs at the beginning of collection due to poor estimates of the amount of scavenging needed to be done. A possible solution to this for the future would be to make the first few phases more aggressive in collection and then scale this back over time.

Figure 4.15 shows a picture that differs to the work based approach. The work based approach never dropped below 20% mutator utilisation for this benchmark. This has not been then same for the work-based approach dropping to 0% at some points. However this is somewhat mitigated by the mutator jumping much higher in between incremental steps. The full garbage collection cycles take slightly longer than the work-based approach. This will be due to the mutator window causing more work for the collector. It should be noted that these figures are not better than the work-based but we have not had time to tune the work time parameters.

Conclusion The incremental mark-sweep approach has a great deal of promise. It has good "real-time" characteristics and consistently outperforms the standard mark-sweep col-

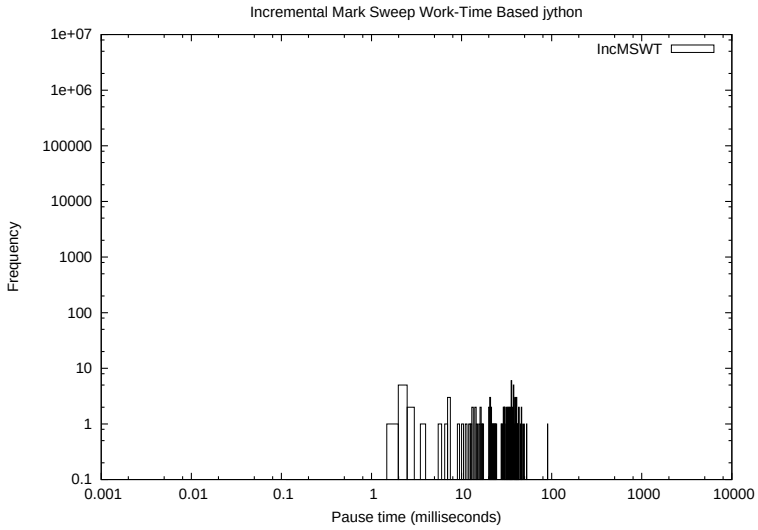


Figure 4.14: Graph demonstrating pause time distribution for IncMS time-work based approach k = 300 t = 5ms (omission)

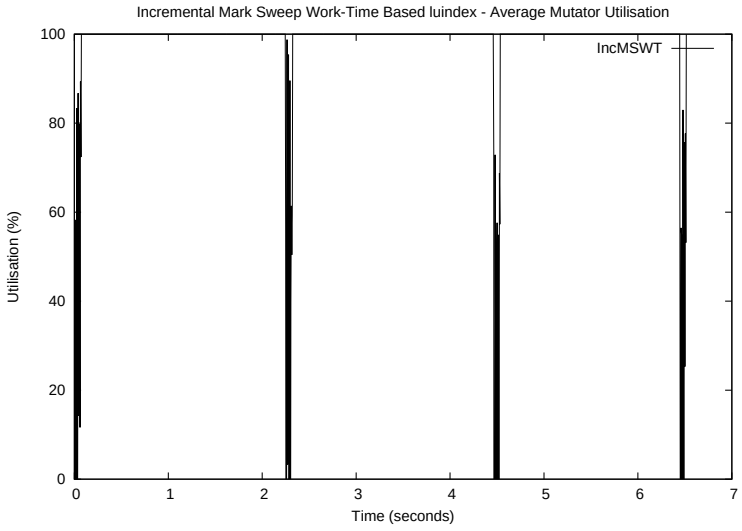


Figure 4.15: Average mutator utilisation for for Incremental Mark-Sweep time-work based approach k = 300 t = 5ms (omission)

lector.

4.1.2.5 Experiment 5

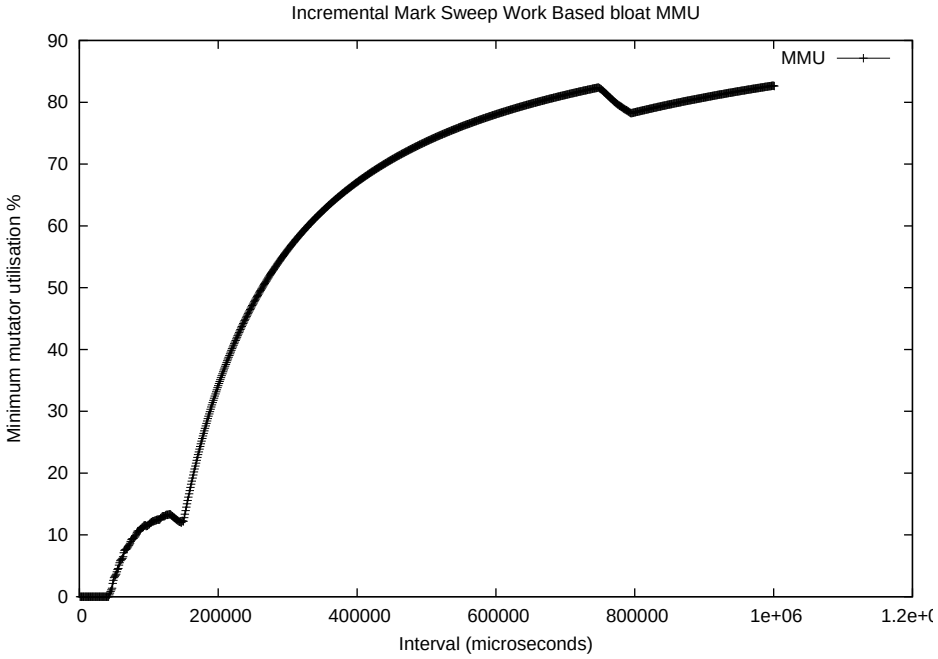


Figure 4.16: MMU for Incremental Mark-Sweep work based k = 3000

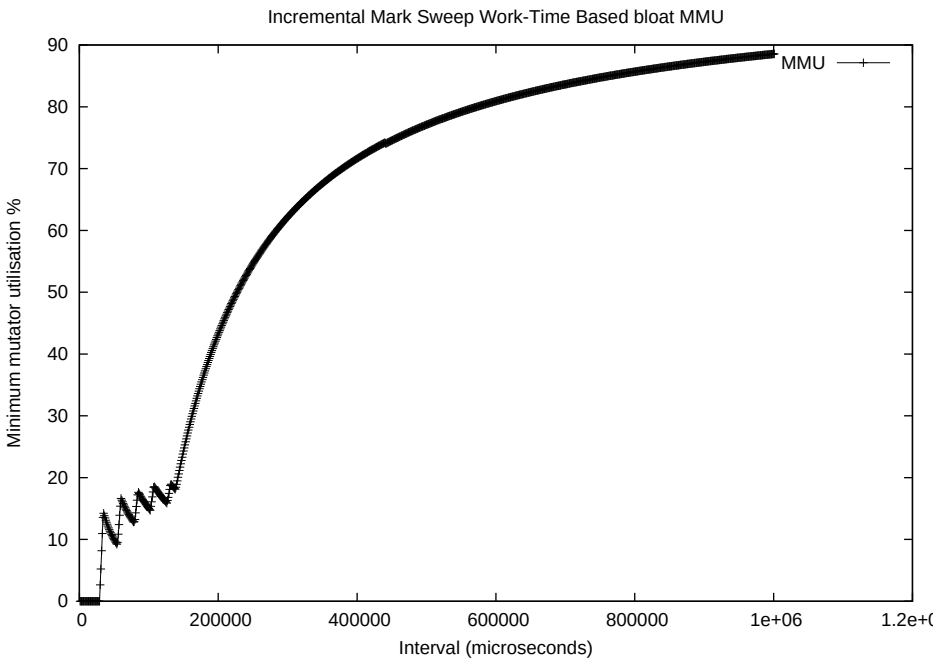


Figure 4.17: MMU for Incremental Mark-Sweep time-work k = 300 t = 5ms

MMU is calculated similar to the average mutator utilisation. A window is chosen and the lowest mutator utilisation is recorded. This window is then increased and the figure is taken again. This demonstrates what percentage the MMU is for a certain interval. The figures show that the MMU of the work-time based approach is better than that of the work-based approach. This is show by the MMU graph of the work-time approach increasing from 0% before the work-based. Observe that generally the percentage is higher at all intervals.

Conclusion This shows that our work-time approach is better than that of the traditional work-based approach. If the minimum pause time was decreased this could be further improved on the MMU.

4.1.3 Incremental Parallel Mark-Sweep

We know evaluate the parallel incremental mark-sweep collector. Figure 4.18 shows the experiments we will conduct for incremental parallel mark-sweep.

Analysis	Desired outcome
1. Varying number of cores for k=3000 analysing overhead, maximum pause time and average pause time	To establish the effect on vary the number of processors on these statistics comparing them to using a single processor
2. Varying the number of cores and times for k=300 analysing overhead, maximum pause time and average pause time	To establish the effect on these statistics when more cores are added in the work-time approach
3. Analyse pause time distribution and average mutator utilisation for best configuration of cores	To discuss the potential of this approach
4 Compare the MMU for the best core configuration with that of a single core.	To discover if adding more cores has an effect on MMU

Figure 4.18: Table showing experiments for Incremental Parallel Mark-Sweep

4.1.3.1 Experiment 1

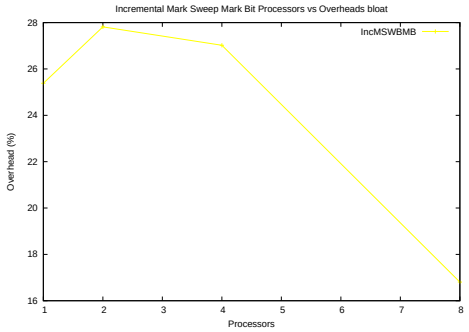
The overheads (Figure 4.19) in the general case (Figure 4.19b) show a large spike in overheads for 2 cores. Intuitively adding more cores would yield better results however this is not the case. The reason for this is that there is a set amount of extra overhead needed in a parallel collector to co-ordinate many threads. In the case of Jikes this can be seen in the Phase stack handling as after each phase all threads must synchronise. This in combination with the large numbers of GC cycles, caused by incremental collection, pushes the overheads of parallel garbage collection beyond the increase in tracing rate.

However there is a drop below the original overhead when the collector uses 4 cores. This shows that 4 cores does enough work to offset the overheads of parallel collection. A decrease tends to occur at 8 cores however this time it is much less than 4 to 8 cores. Sometimes an increase occurs at 8 cores (Figure 4.19c). This will be due to the overheads needed to co-ordinate the collector threads is greater than the increase in tracing.

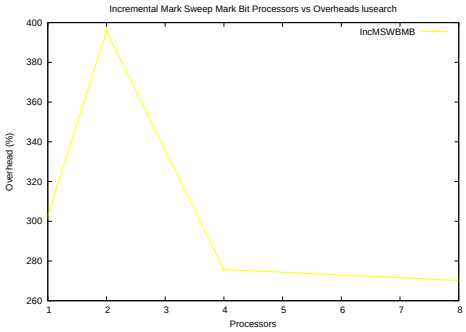
Figure 4.20b shows the general pattern for the maximum pause times with a increase from 1 - 2 cores and then a decrease using 4 and 8 cores. This is due to the same reason as for the overheads however it is magnified somewhat as these maximum pause times occur at the root evacuation and the root rescanning phases which have larger phase stacks and thus more synchronisation points.

Note that the decrease slows a lot between 4 and 8 cores. This is due to the amount of extra work which can be done with 8 cores, not offsetting the inherent costs of synchronisation between collector threads. However if a larger heap size was used there would be a greater decrease as there would be more work to do and having more cores would be beneficial.

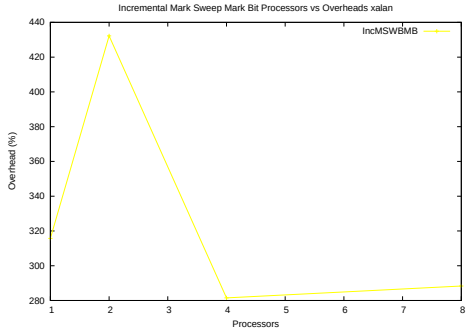
The averages 4.21 for parallel incremental mark-sweep come in two different variants. First the average increases linearly as the cores are increased 4.21a. This makes sense as when the number of cores are increased, the number of pauses will decrease and so the average will increase. The second pattern 4.21b shows again a spike at 2 cores similar to max pause and overhead general pattern. This will be due to the expense incurred using 2 cores vs. 1 core.



(a) bloat overheads for different p's

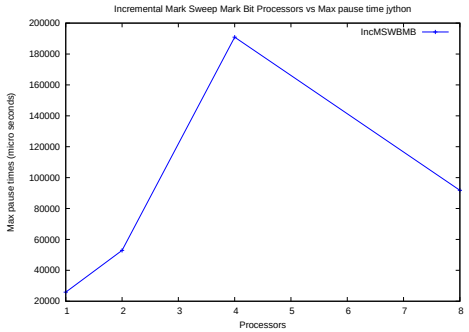


(b) lusearch overheads for different p's

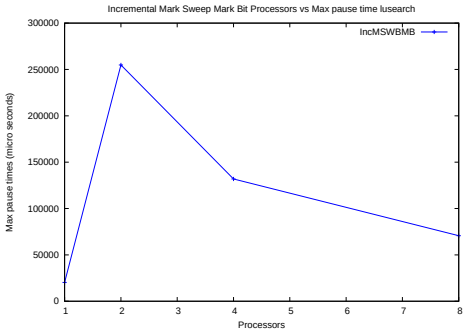


(c) xalan overheads for different p's

Figure 4.19: Results for overheads, Parallel Incremental Mark-Sweep using work-based k = 3000 varying cores

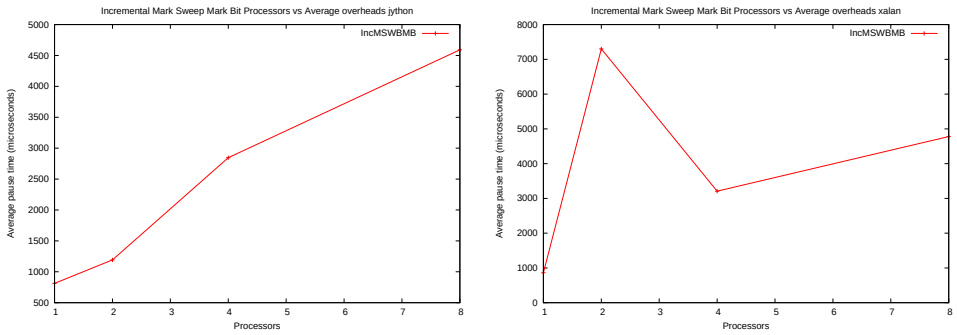


(a) jython max pause times for different p's



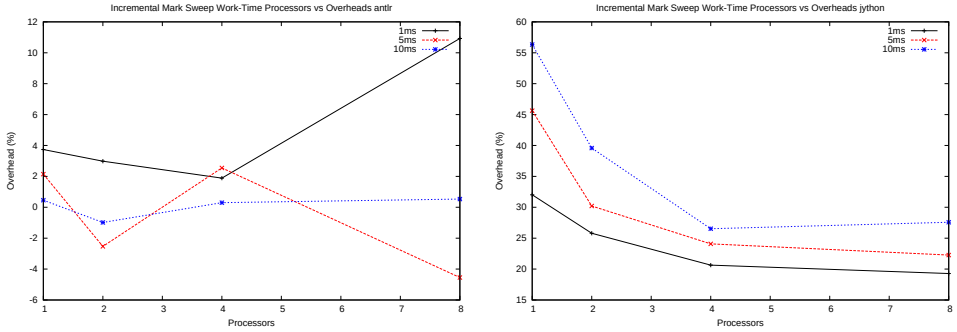
(b) lusearch max pause times for different p's

Figure 4.20: Results for maximum pause times, Parallel Incremental Mark-Sweep using work-based k = 3000 varying cores



(a) jython average pause times for different p's (b) xalan average pause times for different p's

Figure 4.21: Results for average pause times, Parallel Incremental Mark-Sweep using work-based k = 3000 varying cores



(a) antlr overheads for different p's (b) jython overheads for different p's

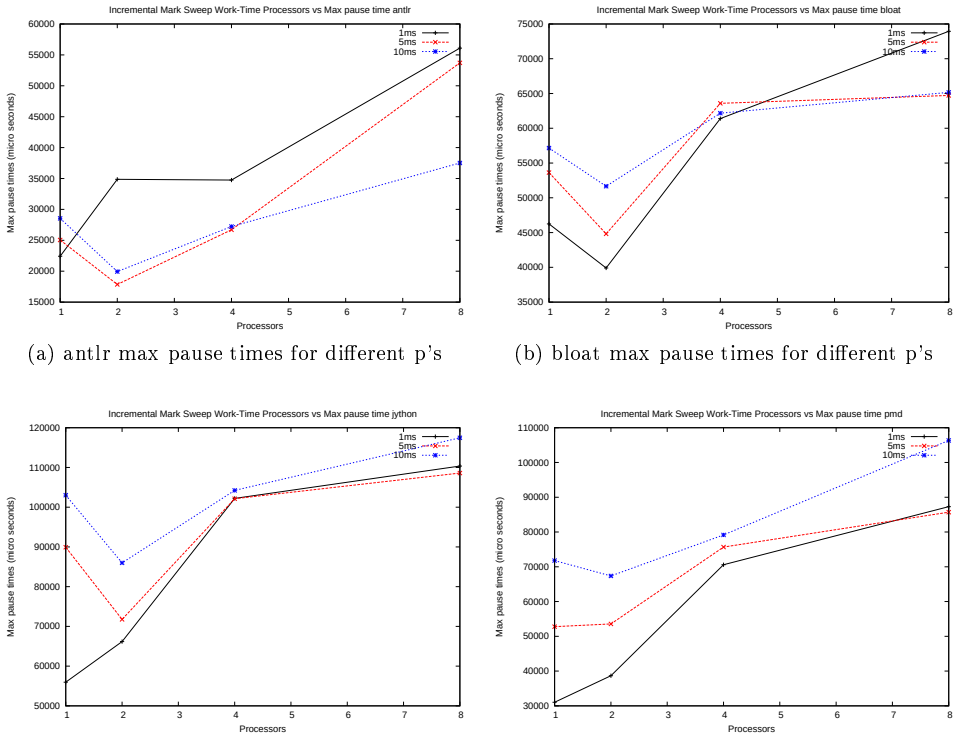
Figure 4.22: Results for overheads, Incremental Mark-Sweep using work-time k = 3000 t = 5ms varying cores

Conclusion Overall putting the collector in parallel more cores are needed to offset the extra overheads caused by parallelisation. However generally the statistics are better standard mark-sweep version despite these problems.

4.1.3.2 Experiment 2

The overheads 4.22 for parallel mark-sweep work-time based show a variety of results. Graph 4.22b is expected to happen, with overheads falling as more cores are added. However some graphs 4.22a do not show this at all and actually appear quite random. We have not had time to explain this and it requires further investigation however it does seem to demonstrate that the core configuration could be application specific.

The graphs for max pause times 4.23 are somewhat erratic like the overheads. However there tends to be a decrease in pause times for 2 cores but then an increase for 4 and 8. Again we are unsure of why this has occurred. Looking at the raw pause times these pauses are on the root rescan phase. Normally this is expected to go down with more cores. It



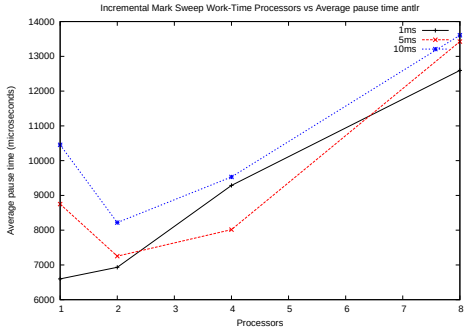
(a) antlr max pause times for different p's

(b) bloat max pause times for different p's

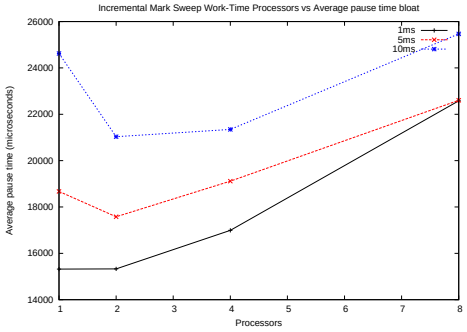
(c) jython max pause times for different p's

(d) pmd max pause times for different p's

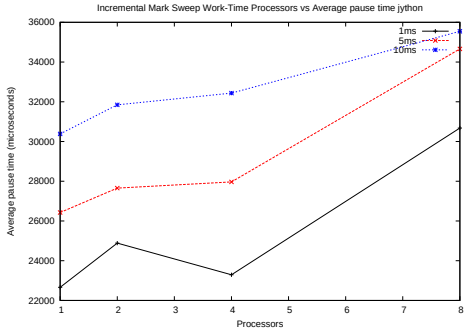
Figure 4.23: Results for maximum pause times, Parallel Incremental Mark-Sweep using work-time k = 3000 t = 5ms varying cores



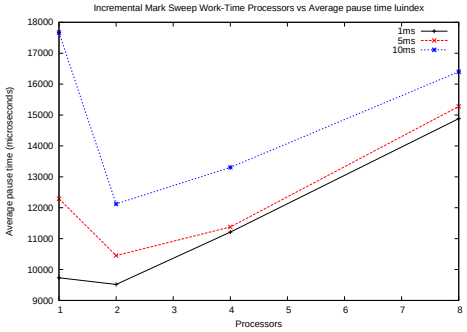
(a) antlr average pause times for different p's



(b) bloat average pause times for different p's



(c) jython average pause times for different p's



(d) luindex average pause times for different p's

Figure 4.24: Results for average pause times, Incremental Mark-Sweep using work-time k = 3000 t = 5ms varying cores

could be due to thread synchronisation overhead caused by the syncing in the transition between phases, but this is just speculation.

Average pause times (Figure 4.24) tend to decrease with 2 cores. This makes sense as on average the time needed to do the two most expensive phases of collection (root evacuation and root rescanning) will be cut. We then see the average move back up with more cores which also makes sense as when we increase the number of cores we will have less GC cycles and as such a higher average.

4.1.3.3 Experiment 3

Figure 4.27 shows, as for the single core version, 2 distinct groups of pause times. As for the single core version this will be one set (the higher one) for the root scan and root rescan phases; the other for the incremental phases. Similar to the single core version with omission (Figure 4.26) of the start and end phases of collection there are much lower pause times than before. The distributions of the pause times are now lower than in the single core version due to more work is done in each incremental phase. This will cause there to be less incremental cycles which means there is less chance for the write barrier to be triggered. This means that less work will need to be done and thus the lower pause times occur.

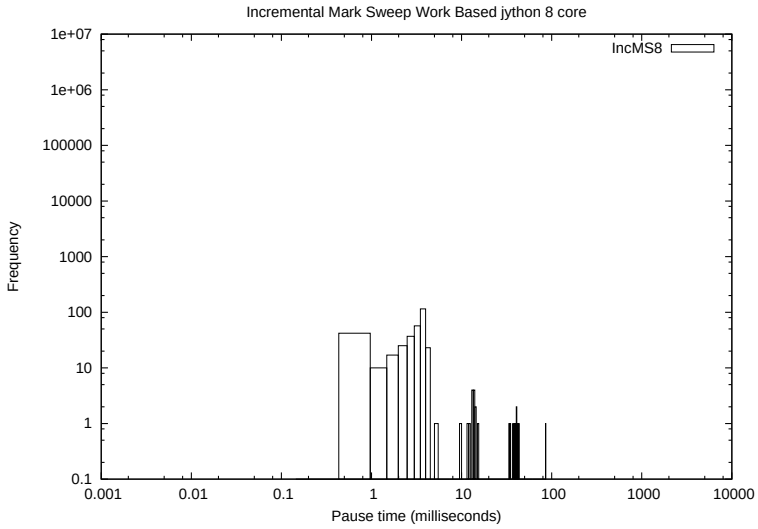


Figure 4.25: Pause time distribution for Parallel Incremental Mark-Sweep work-based k = 3000 8 Cores (non omission)

Figure 4.27 shows the pause time distribution for the work-time approach. When comparing it with the work-based we see that the work-time approach has lower pause times. This is different to the single core results where work-time overheads increase. We believe this to be because the parallel approach causes significantly fewer GC cycles. This means that the pauses of the work-time caused by the write barrier or allocations are less likely and as such there are lower pause times.

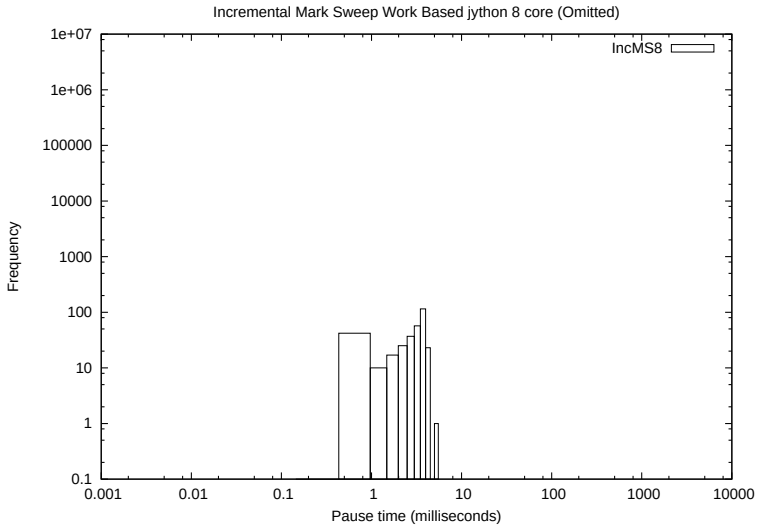


Figure 4.26: Pause time distribution for Parallel Incremental Mark-Sweep work based k = 3000 8 cores

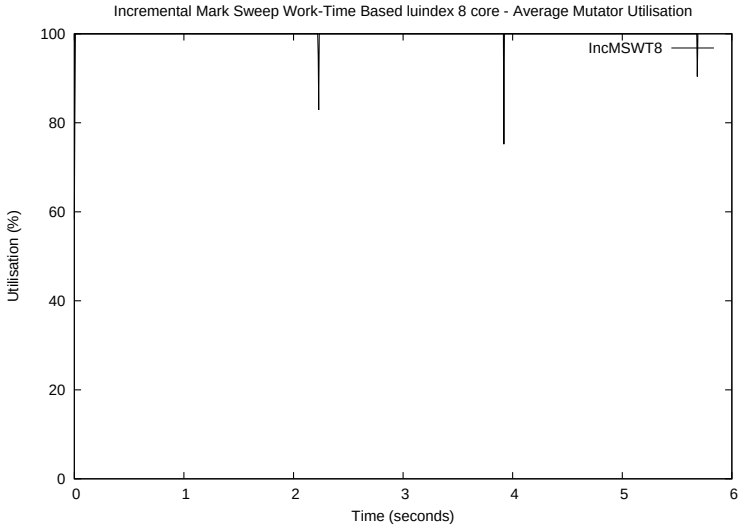


Figure 4.28: Average mutator utilisation for Parallel Incremental Mark-Sweep work-based k = 3000 t = 5ms

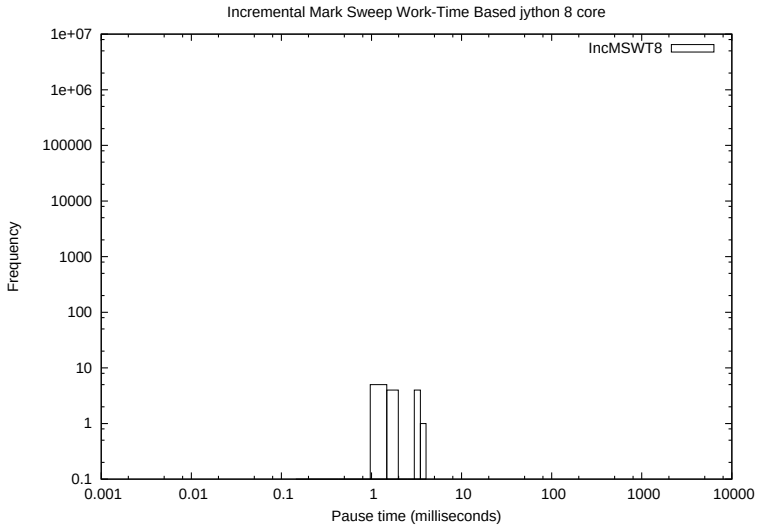


Figure 4.27: Pause time distribution for Parallel Incremental Mark-Sweep work-time based k = 3000 t = 5ms 8 cores

The average mutator utilisation (Figure 4.28) demonstrates where the potential lies in the multi core approach. Compared to the single core approach drops are shorter than in the single core version. This shows that the multi core approach gives better average mutator utilisation than that of a single core which is an important real-time characteristic. An increase in window size would have given more interesting results but we wanted a comparison with the single core version to keep things consistent.

Conclusion The work-time approach is better overall than the work-based approach when run in parallel. This is promising and we hope to see the work-time preform well in Experiment 4 4.1.3.4 like in the single core tests. This will verify that our work-time approach is better than work-based collection.

4.1.3.4 Experiment 4

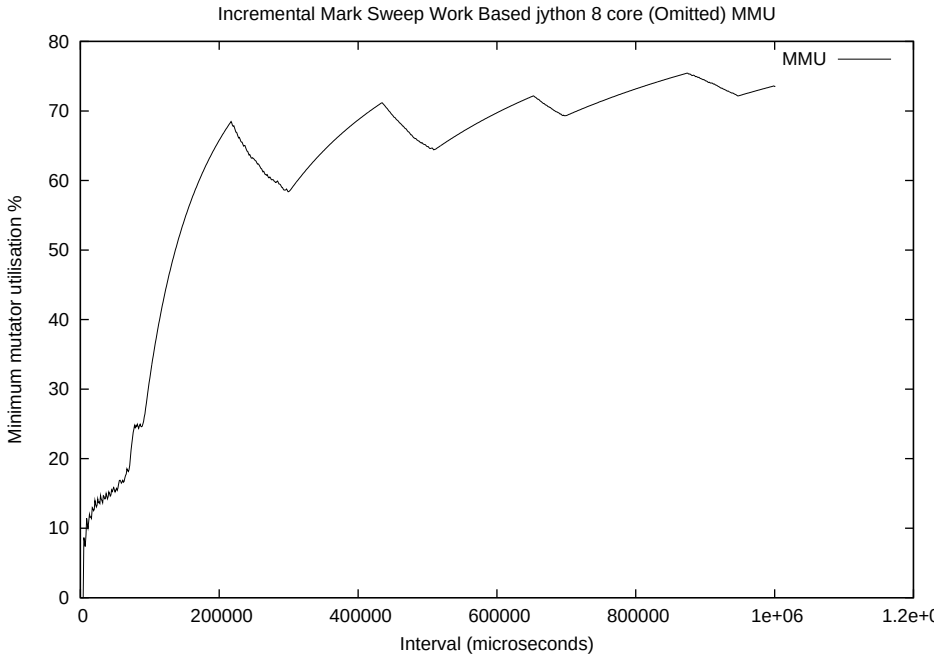


Figure 4.29: MMU for Parallel Incremental Mark-Sweep work based k = 3000 8 cores

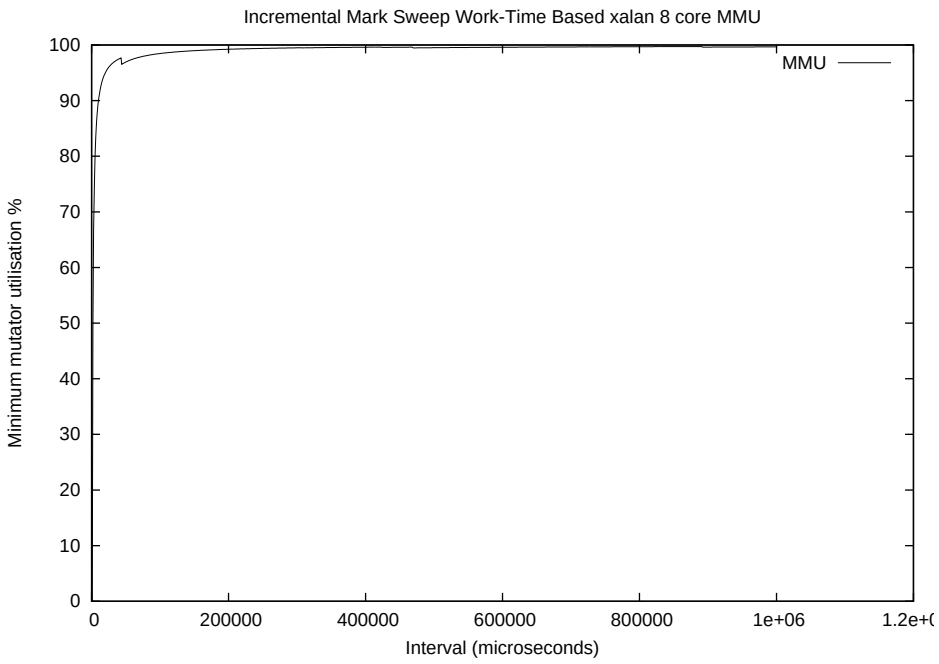


Figure 4.30: MMU for Parallel Incremental Mark-Sweep work-time based k = 3000 t = 5ms 8 cores

In Figures 4.29 and 4.30 we see the MMU for the work-based and work-time based approaches. First both are better than there single core counterpart which demonstrates that increasing core improves MMU. When we come to analyse them individually we see that the work-time approach is significantly better than the work-based counterpart. This will be because the extra cores doing work combined with the work-time approach give the mutator more space than work and thus increasing MMU. This demonstrates that the parallel incremental mark-sweep collector exhibits excellent real-time characteristics.

4.1.4 Incremental Copy Collector

We will now evaluate our incremental copying collector. We have carried out the experiments outlined in 4.3:

Analysis	Desired outcome
1. Varying k analysing overhead, maximum pause time and average pause time	Investigating the effects of varying k on these three criteria and also compare with stop the world collector. This will give us our "best" k value.
2. Analysing overhead, maximum pause time and average pause time for time-work based approach varying k for different times	Establishing the costs of the time-work based approach when applied to an incremental copying collector and establish our "best" k and time value.
3. Analysing the pause time distribution and mutator utilisation of the incremental copying both work and time-work based for "best" values of k and time.	Evaluate the potential for the approaches.
4. Analysing MMU for our work-time based approach.	Evaluating the effectiveness of time-work based approach on MMU.

Table 4.3: Table analysing experiments for the Incremental Copying Collector

4.1.4.1 Experiment 1

The overhead figures 4.31 are particularly high for incremental copying collection. This is somewhat expected as the cost of the stack based read barrier is very high. If we subtract this cost we get an average overhead of 40%. This is more in line with the overhead figures of incremental mark-sweep.

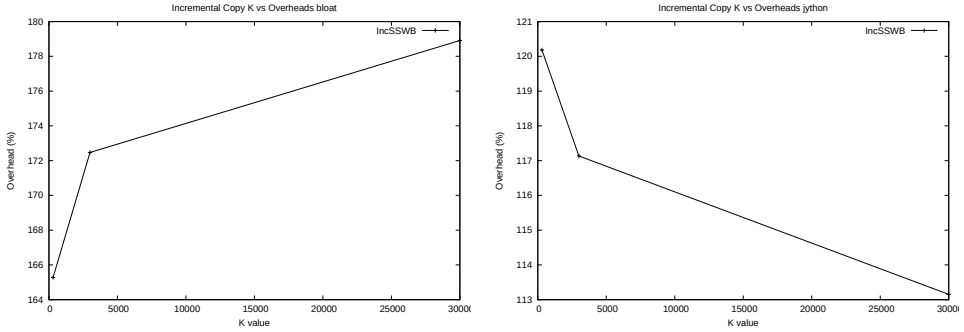
The maximum pause times (Figure 4.32) are also a lot higher than that of the incremental mark-sweep algorithm. This is expected as copying an object to scavenge it is more expensive than setting a mark bit. However on average there is a lower maximum pause time than the base semi-space collector.

The average pause times generally follow the pattern of 4.33a increasing with k. Due to the same reason as the incremental mark-sweep collector in that with a coarser grain size fewer collection cycles will be done which will be larger and thus a higher average. The exception to this rule is in the jython benchmark 4.33b. The reason for this is similar to the large pause time in that when k = 3000 there is an influx of writes during one incremental cycle which causes collection to be forced. Note that this did not happen on every run. This issue is magnified in a copying collector due to the heap being halved.

4.1.4.2 Experiment 2

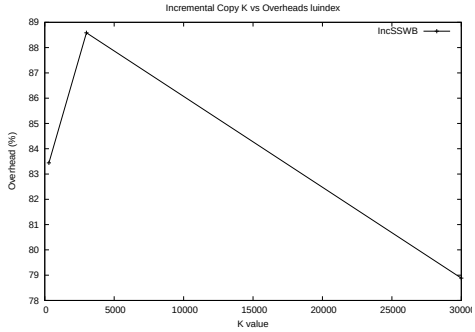
The overheads for the work-time based 4.34 approach are quite erratic. This is most likely due to the associated randomness caused by the time-work based approach in that in any time quanta it is unknown if a burst in allocations or writes will occur. However we have not had time to confirm this with more in depth analysis.

The overhead is slightly higher than the work based method. This is because more activity is allowed between collection cycles and thus there will be higher overhead in the root rescanning phase. Confirmation of this was given after examining the raw results in more detail.



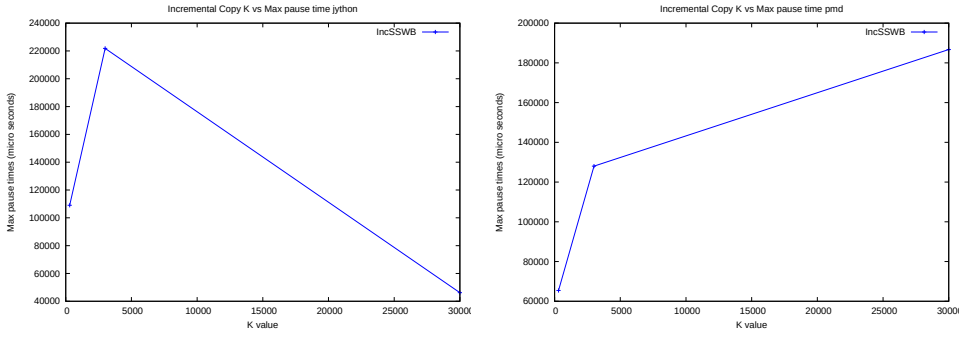
(a) bloat overheads for different p's

(b) jython overheads for different p's



(c) luindex overheads for different p's

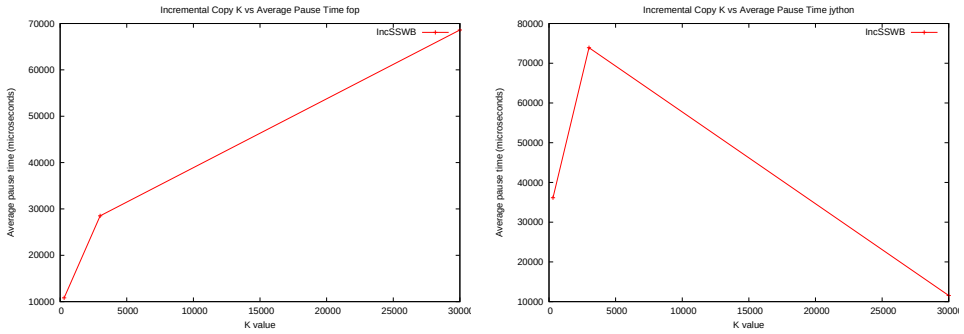
Figure 4.31: Results for overheads, Incremental Copy using work-based varying k



(a) jython max pause times for different p's

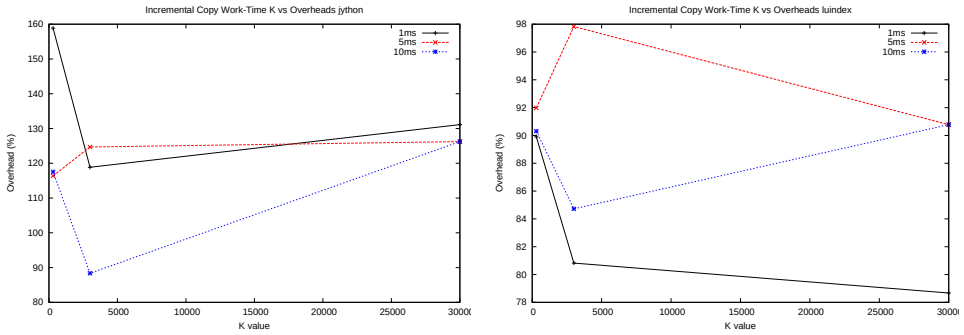
(b) pmd max pause times for different p's

Figure 4.32: Results for maximum pause times, Incremental Copy using work-based varying k



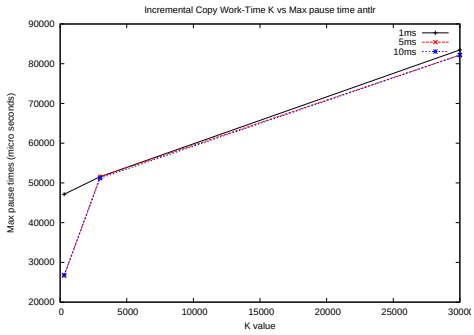
(a) fop average pause times for different p's (b) jython average pause times for different p's

Figure 4.33: Results for average pause times, Incremental Copy using work-based varying k

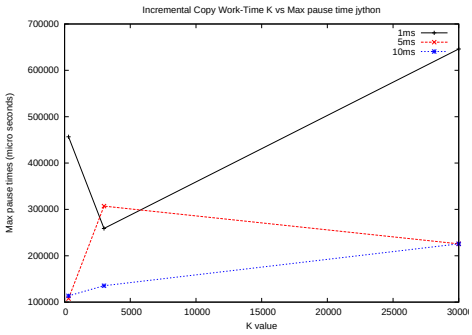


(a) jython overheads for different p's (b) luindex overheads for different p's

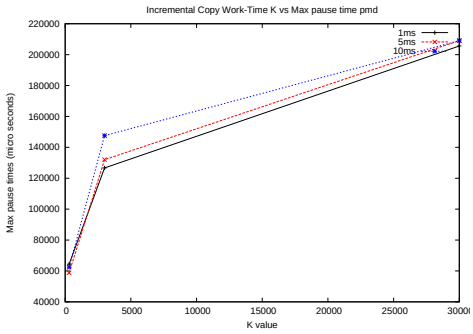
Figure 4.34: Results for overheads, Incremental Copy on single core using the work-time approach



(a) antlr max pause times for different p's

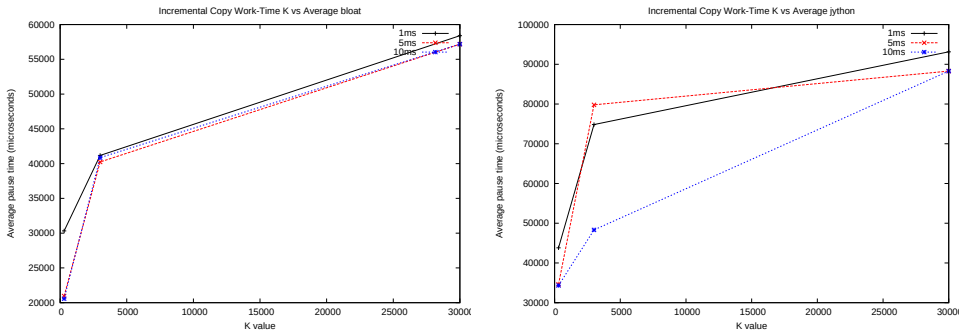


(b) jython max pause times for different p's



(c) pmd max pause times for different p's

Figure 4.35: Results for maximum pause times, Incremental Copy on single core using the work-time approach



(a) bloat average pause times for different p's (b) jython average pause times for different p's

Figure 4.36: Results for average pause times, Incremental Copy on single core using the work-time approach

The maximum pause time (Figure 4.35) has increased over the work-based approach. This is to be expected as the work-time approach gives more opportunity for the mutator to run.

In 4.35c as the time frame is increased there is a higher maximum pause times. This follows from that as we increase t the collector will have more work to do and thus the max pause time will increase. However overall this pattern is not followed by all the benchmarks 4.35b, 4.35a. Once again we have been unable to analyse this fully in order to say why this is but we believe that the read barrier is slowing the collector down at such a rate that a lot of results are unexpected.

The average pause times generally have a similar pattern 4.36a to what is expected, with the average pause time increasing as the grain size is increased, k. What is strange is that at the different time quanta there is the same average time. As the read barrier overhead is causing the mutators to become slower it is reasonable to assume that the number of write barrier hits and allocation rate are less than normal. This will mean that the main reason for differences between time frames will be smaller and as such the variance between the times normally seen is not here.

The jython benchmark is different to the trend. This seems to be a common theme for jython in this set of benchmarks and is probably due to a specific characteristic of jython. However we have not had time to investigate this so can only speculate.

Conclusion The benefits gained from the work-time approach that were demonstrated in the mark-sweep collector are not present here. This is due to the large overhead incurred by the stack based read barrier meaning the mutator cannot make much progress during these time periods.

4.1.4.3 Experiment 3

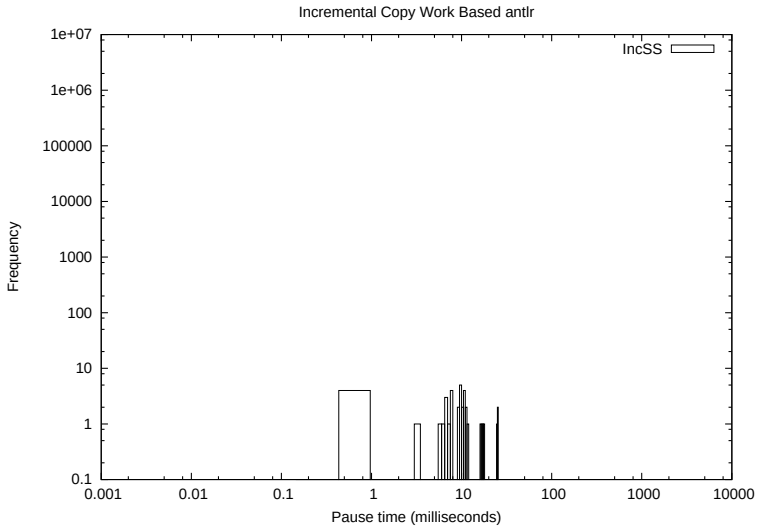


Figure 4.37: Pause time distribution for Incremental Copy work-based k = 300 (non omission)

There are 3 separate groups in the pause time distribution graph (Figure 4.37). However there is a group clustered from 15-5 ms. This group occurs at later portions of the program after collection statistics have stabilised. The 1-0.5 ms is seen in the beginning of the program. This is probably due to poor estimation of the work calculation. The other group at 15ms+ are the primary root evacuation phase and the root rescan phase. It is interesting to note that generally there is an increase in this pause time from the mark-sweep algorithm. This is to be expected as scavenging in the copy collector is more expensive than in the mark-sweep collector.

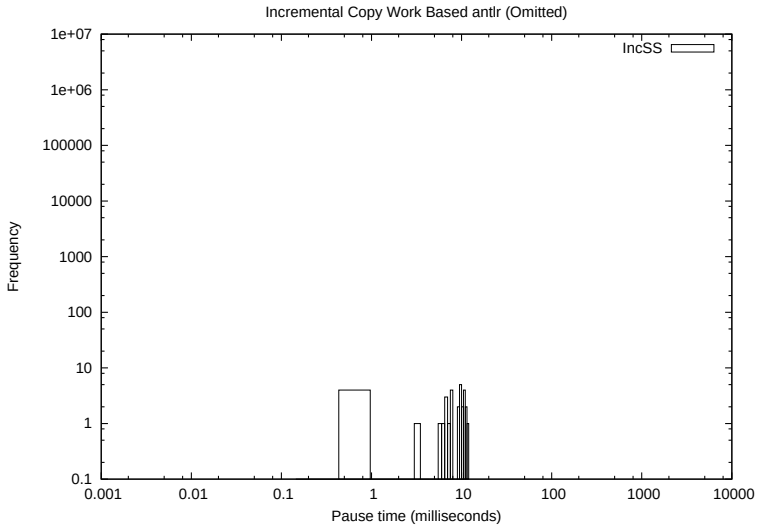


Figure 4.38: Pause time distribution for Incremental Copy work-based k = 300

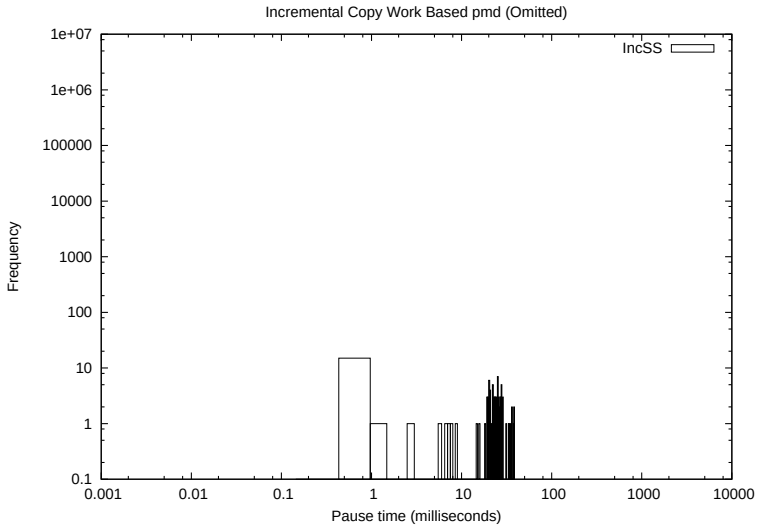


Figure 4.39: Bad pause time distribution for Incremental Copy work-based k = 300

When the initial root scan and the root rescan phase are removed (Figure 4.38) the

pause times are all lower than 15ms. However this is not always the case as seen in 4.39. The reason for this is that the increased overheads of copying objects are more expensive than originally thought. Note that our pause time distributions are still significantly better than the baseline version of the GC.

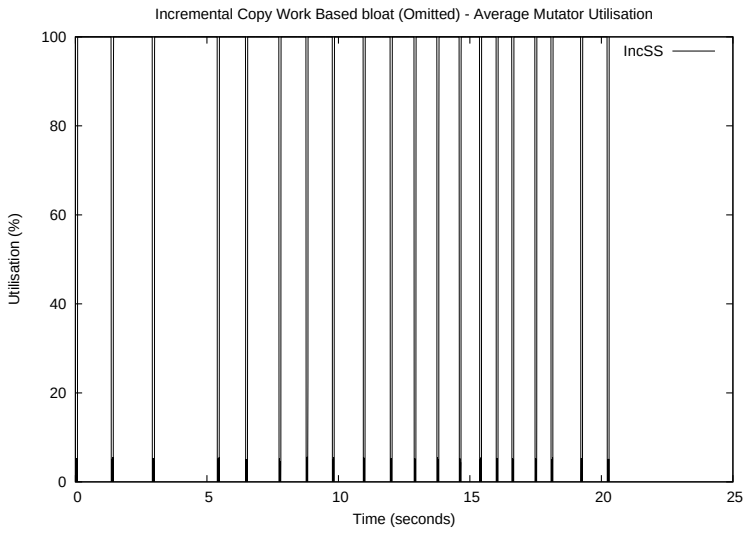


Figure 4.40: Average Mutator utilisation for Incremental Copy work-based k = 300

The average mutator utilisation (Figure 4.40) shows that the real-time characteristics of copy collector are not as good as the mark-sweep. This is to be expected due to the extra overheads of copying. What is good is there is still shorter periods of drops to 0% especially compared with the baseline version of the collector.

Conclusion Overall I think that whilst the copying collector is good it has a issues the same core issues as the incremental mark-sweep but compounded. This is because the tracing is now more expensive.

4.1.4.4 Experiment 4

For the MMU (Figure 4.41, 4.42) there is an interesting set of results showing increase at around the 80ms mark. What is interesting is that this occurs on both approaches. This is somewhat difficult to explain and due to a lack of time we have been unable to explore this. We believe it is due to the increased overheads of the copying collector coupled with the extra work incurred by the work-time approach causing work-time to be no better than work-based. However we must again stress that the work-time collector is much more susceptible to poor choices of k and as such performance is probably being hindered by this.

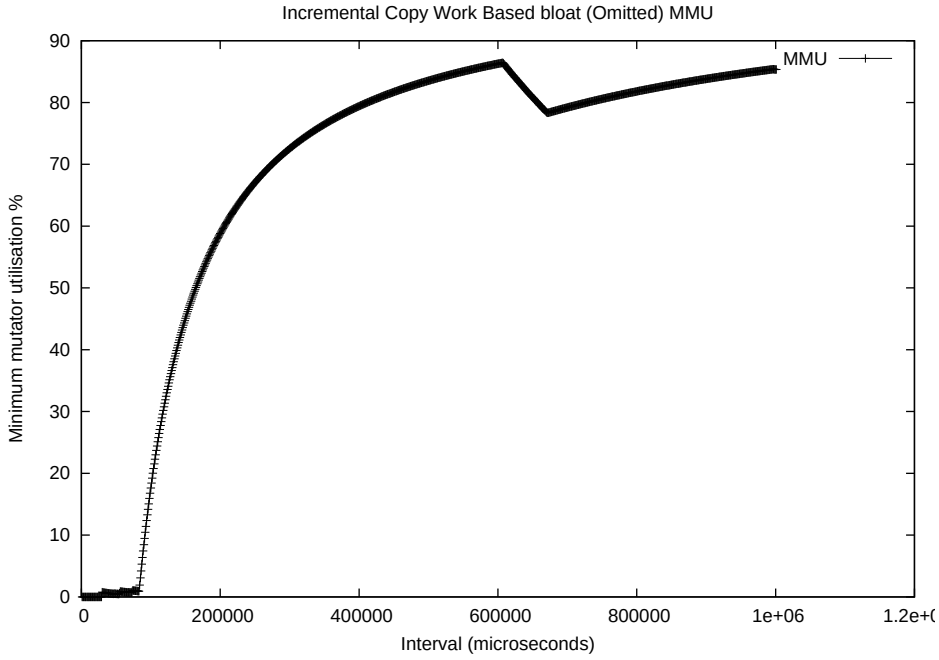


Figure 4.41: MMU graph for Incremental Copy work-based k = 300

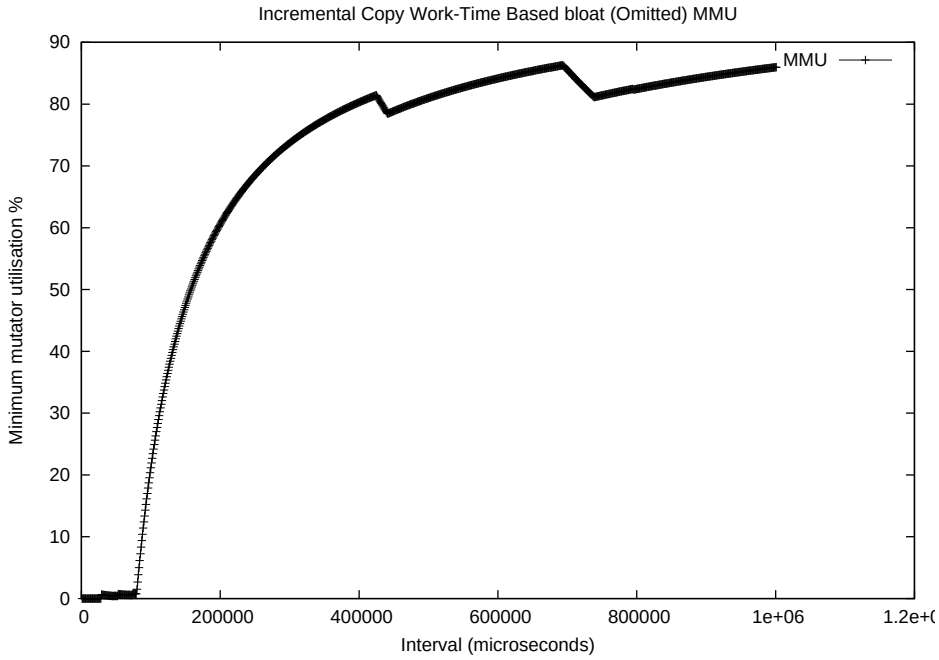


Figure 4.42: MMU graph for Incremental Copy work-time based k=300 t = 10ms

4.1.5 Parallel Brooks Style Incremental Copy Collector

Unfortunately due to an implementation issue the parallel copying collector was unable to finish several benchmarks on the machine specified. We could of analysed the partial results but felt they would be uninteresting as comparisons would be very speculative. We instead show results gathered from the machine the collector was developed on. This machine were: An Intel(R) Core(TM)2 Duo CPU T9400 @ 2.53GHz, with 8 GB of ram and no swap space. The OS was Ubuntu Jaunty 9.04 ia32. The OS was running X with standard networking connections enabled. We also ran the baseline version of the collector to do relevant comparisons between our collector and the baseline version.

4.1.5.1 Development machine results

For the development machine we decided to concentrate on testing the real-time characteristics of the collector. We conducted the experiments outlined in 4.4.

Analysis	Desired outcome
1. Analyse the pause time distributions and mutator utilisation for the multiple core approach comparing to the single core	Establish if using more cores gives a benefit to these characteristics
2. Analyse the MMU of the multi core approach vs the single core	Establish the effect on MMU of adding multiple cores to the approach

Table 4.4: Experiments for Parallel Incremental Copying on development machine

4.1.5.2 Experiment 1

Figure 4.43 shows the pause time distribution for luindex running on 2 cores. Note the large outlier which has occurred at around 5000 ms. On investigation this was a rescan phase which took a particular long time. This coupled with the overhead which using multiple cores causes a very large pause time.

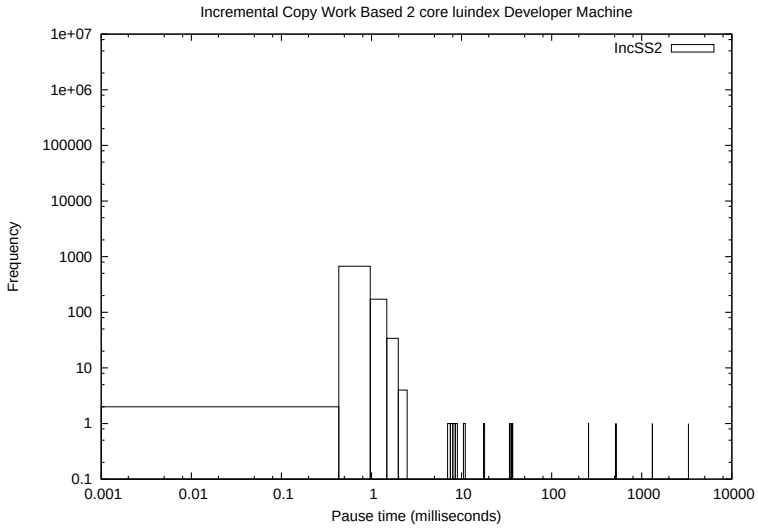


Figure 4.43: Pause time distribution for Parallel Incremental Copy work-based k = 300 p = 2 (non omission)

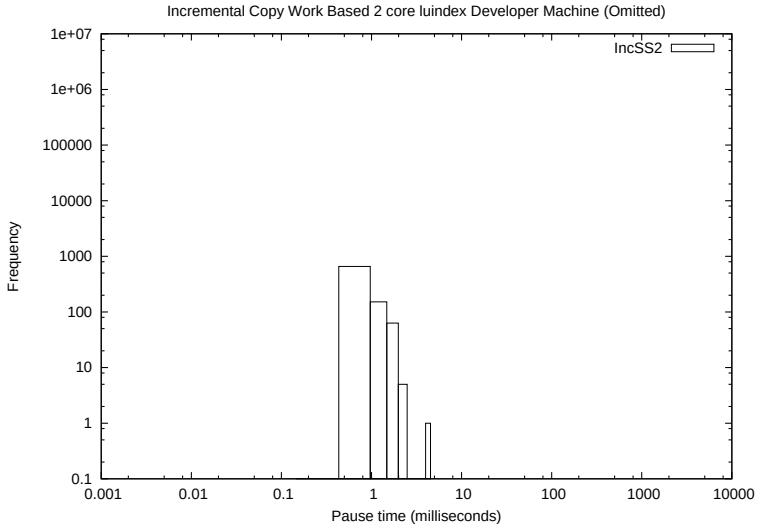


Figure 4.44: Pause time distribution for Parallel Incremental Copy work-based k = 300 p = 2

The omitted version (Figure 4.44) of the pause time distributions shows great potential in the multi core approach. The pause times are all sub 5 ms. However this is not the case for all benchmarks as can be seen in Figure 4.45. The reason for this is that xalan is a much more demanding application when it comes to memory management. This means that more collection work has to be done and as such there is a higher chance of larger pause times. What could be done in the future is to estimate this pause time pre-collection and then either opt to have each core do less work. This would mean that there would be more GC cycles but shorter pause.

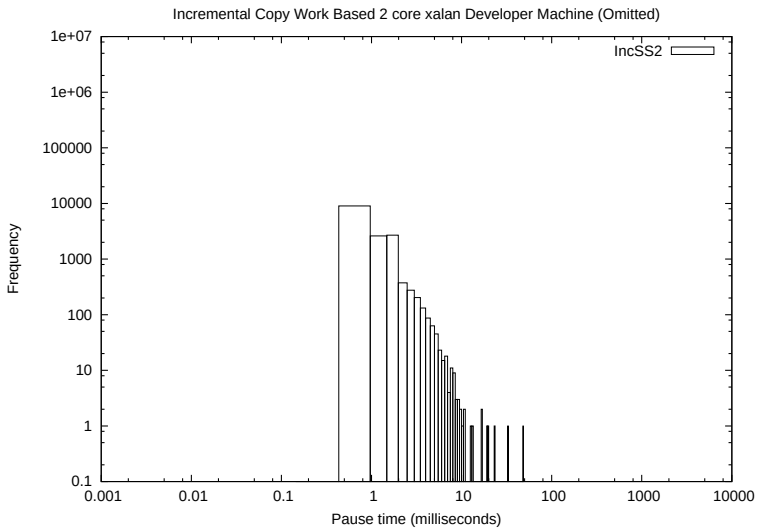


Figure 4.45: Bad pause time distribution for Incremental Copying work-based k = 300 p = 2

Figure 4.45 shows a worse pause time distribution than what occurred in Figure 4.44. In general there is a sub 15 ms pause time. This is good however the outliers, which lie between 20-80 ms, cause the overall maximum to be greater. The reason for these higher values is due to the tracing rate not keeping up with allocation and write barrier hits cause a spike in the amount of tracing rate to do. It is very hard to mitigate this. With the multi-core approach we could make collectors do work which scales with spikes to smooth them.

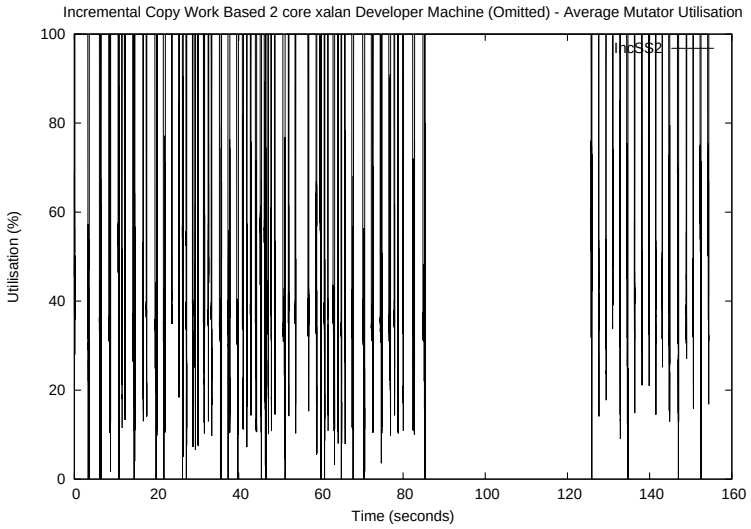


Figure 4.46: Average Mutator utilisation for Incremental Copy work-based 2 cores

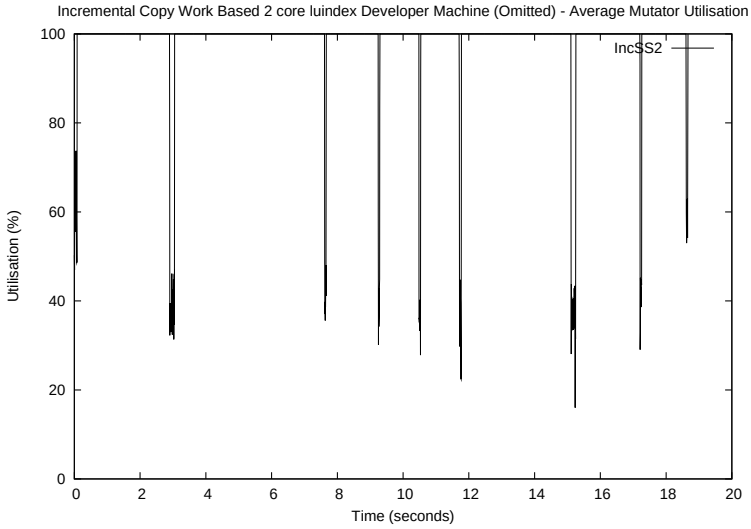


Figure 4.47: Average Mutator utilisation for Incremental Copy work-based 2 cores

Figures 4.46 and 4.47 show the average mutator utilisation for xalan and luindex. Notice

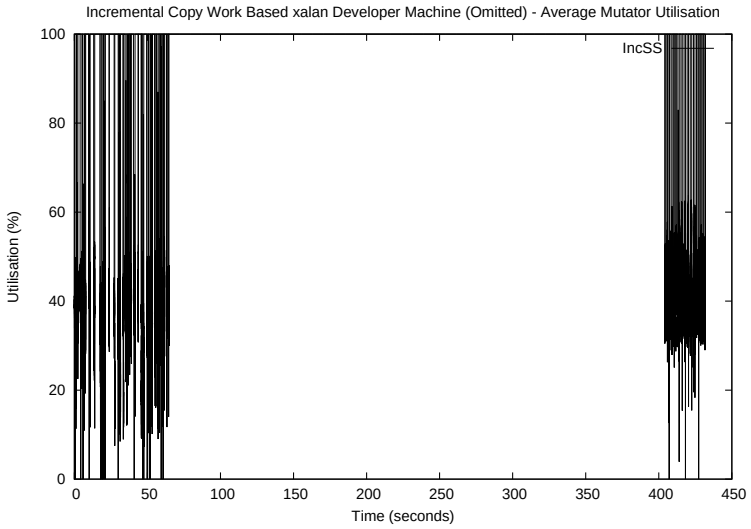


Figure 4.48: Average Mutator utilisation for Incremental Copy work-based single core

the large difference between the two graphs and this is largely due to the differences between the two benchmarks (xalan is multi threaded and long running while luindex is shorter running and not multi threaded).

Figures 4.48 and 4.49 are the same benchmarks for a single core. Notice that the total time for each benchmark to run decreases in the 2 core version by 64% for xalan and 92% for luindex. This indicates that the multi-core approach will greatly decrease overheads for the Incremental Copy collector.

Generally the mutator utilisation is higher in the multi-core graphs. This is especially apparent when comparing luindex. This shows that the real time characteristics of the multi-core are better than that of the single core.

The number of spikes in the single core approach is much greater than the multi-core. This will be because of one of the multi-cores aim to reduce the number of GC cycles and thus reduce overheads.

Conclusion Over these results the multi-core has much better real-time characteristics than the single core. This effect appears much greater than in the mark-sweep collector. This will be because the cost of scavenging is greater in the copy collector and thus adding extra core would be more useful than in mark-sweep.

4.1.5.3 Experiment 2

Figures 4.50 and 4.51 show the MMU for the luindex benchmark. Here the MMU for the 2 core outstrips that of the single core. Demonstrating the "real-time" characteristics of the multi-core approach are better than the single core. This is the same as for the parallel

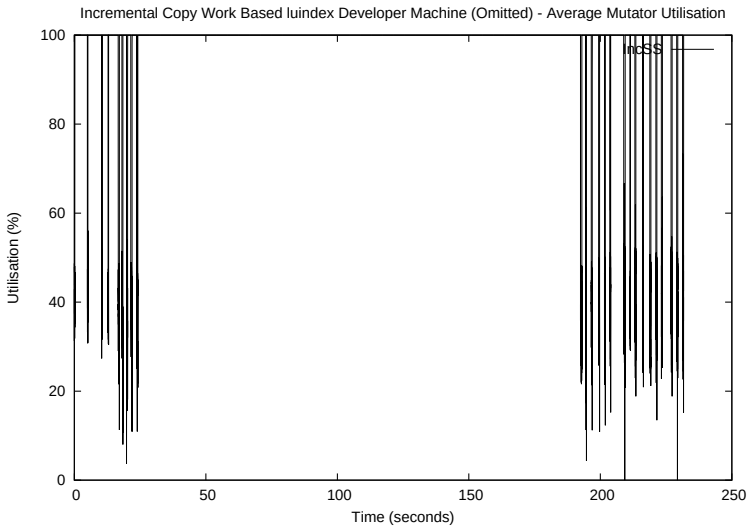


Figure 4.49: Average Mutator utilisation for Incremental Copy work-based single core

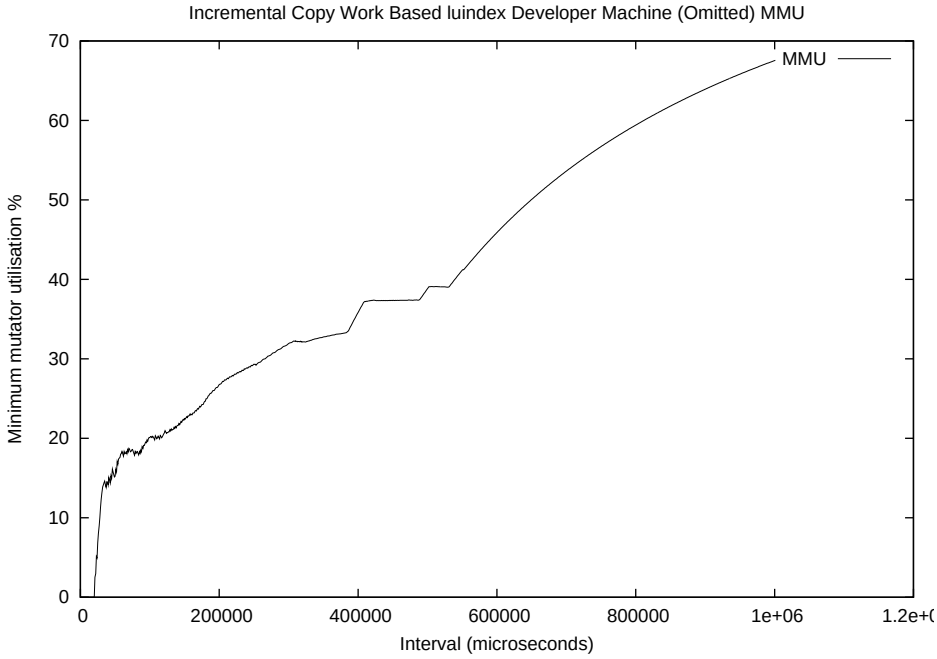


Figure 4.50: MMU for Incremental Copy work-based single core

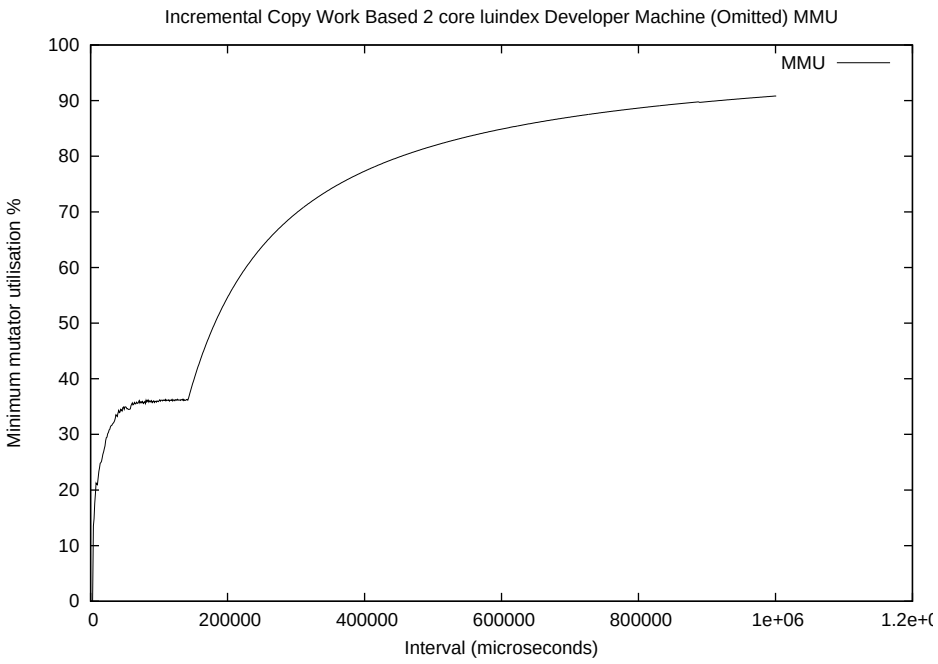


Figure 4.51: MMU for Incremental Copy work-based 2 cores

incremental mark-sweep. We hypothesise that this trend would continue up to 8 cores.

4.2 Conclusions

4.2.1 The Collectors

We have given back two incremental collectors to the Jikes comunity. Our collectors outperform the baseline version over all aspects of analysis. Whilst there was an increase in overhead due to our approach this is to be expected from a "real-time" garbage collector. We have also shown the impact of adding more cores to garbage collection does not always give preferable results largely due to implicit overhead caused by parallel garbage collection. What we do tend to find is that for a certain core configuration dependent on heap size we get a performance increase from the multi-core approach.

We have shown that our approaches have potential to achieve constantly shorter pause times (sub 10ms for the platforms described) for both style collectors. Also we have shown that "real-time" characteristics for collectors become significantly better with multiple cores than on a single core despite an increase in overheads.

We also proved that our new work-time based metrics for the incremental tracing rate shows better "real-time" characteristics than the classic work-based approach. This is promising however what would be good to do is now evaluate the work-time approach against pure time based incremental tracing.

4.2.2 The Framework

The main contributions of this project is a new framework for building incremental collectors. This has been given back to the Jikes community in the form of a set of incremental subplans. This framework will allow future developers to create their own incremental garbage collector. This will grow Jikes as a platform for developing "real-time" garbage collectors. We have demonstrated the frameworks extensibility through the use of it in construction our two garbage collectors.

4.2.3 Future work

4.2.3.1 Extensions to the current collectors

The current collectors have several features that could be extended. The first is our new parallel work balancing mechanism. In this we would have each collector thread have their own scavenge deque. The collector threads would use these until they were exhausted after which they would "steal" work from other collectors. The stealing mechanism would take from the tail of the deque so that it did not interfere with the collector. This would need some locking to ensure that no two collector threads get the same piece of work. The processes from which to steal would be determined from a centralised counter which would be incremented each time a collector thread needed to steal.

As the evaluation showed, the collectors main issues where with the pause times in the beginning and end where we had to scavenge and then rescavenge the root set. In order to alleviate this we could incrementally scavenge the stack. This would enable us to spread the cost of the associated pauses.

The read barrier overhead was increased due to a need to do a null check on objects. This could be fixed with a dummy null object which always had an indirection pointer. With this we could also use our linked list object storage method for write barriers.

4.2.3.2 Incremental Immix

We have now created a framework for developing incremental collectors. However, currently the collectors we have made are well known and widely used. To demonstrate the usefulness of this collector further we could apply our incremental framework to a more novel collector. An excellent candidate for this would be the Immix collector[16]. This is already made in the VM.

The main plans for future work can be outlined as follows:

- **Make Immix extend the incremental plan:** This will be similar to how it was done for the mark-sweep plan.
- **Create a logging write barrier:** This barrier will be active during collection cycles so that if any writes occur that object can be rescanned.
- **Make the defragmentation step incremental:** Immix works by occasionally defragmenting memory in an efficient manner this defragmenting could be made to be incremental by splitting it into smaller chunks similar to splitting the closure stage of the other collectors.

4.2.3.3 A Collector Similar To The Metronome

Currently we have created two incremental collectors. One that is fragmenting but fast and one that is defragmenting but slower. As such we have the building blocks to create a more advanced collector combining these. This would be a collector similar to the metronome collector[10].

Below is an outline of the main features of the Metronome and implementation specifics.

- **Combining the Incremental Semi-space and Mark-Sweep together:** We will first have to determine when a certain threshold of fragmentation has been reached. This could be done with a time difference between separate collection cycles or how long it took the mutator to allocate a new object on the heap last allocation. Then a method to utilise two collectors in one space. This should not be too difficult as Jikes allows users to rebind the collector to a different space on-the-fly. The problem will be that semi-space collectors use bump pointer allocation and mark-sweep uses free lists. We could use both or create a way to merge the two dependent on cycle. Finally we need a way to switch the barriers dependent on what space is currently live. It will be simpler to use a boolean in order to change what code is utilised however it could be possible to implement cloning like in STOPLESS which is the more desirable method.
- **Creating Stacklets/Arraylets:** By doing this it allows the collector to incrementally scan the stack and arrays. Arraylets should be made first first as it has less impact on how the collector as a whole works. The implementation of an array can be seen in: *RVMArray*. There needs to be a way of being able to split the array into smaller chunks and scan the array chunk at a time. Stacklets will be a lot harder to create in Jikes due to it being implemented at a very low level. The current implementation of the stack is split into a few sections: *StackBrowser* used to explore the stack, *StackManager* used to manage the compiler-specific portion of the stackframe, *StackTrace* describes the state of the call stack at a particular instance, *StackframeLayoutConstants* constants used to represent the stack note this is architecture specific. These will need to be reimplemented so that we can scan the stack section at a time

it should be noted that linking the stack together when it is partially between to and from space during a copy cycle will be particularly hard and may need extra pointers in the header of the stacklets.

- **Time based scheduling:** This is the least desirable of these three points however also probably easiest to implement. A new scheduling algorithm will be created that is time-based. To do this you will need to utilise *Timer* objects. This algorithm would allow us to evaluate the time-based approach against our work-time approach.

Appendix A

Evaluation Results

Note some tables have been omitted for brevity

A.1 Incremental Mark Sweep Collector

	antlr	bloat	fop	jython	luindex	lusearch	pmd	xalan
Baseline Mark-Sweep	5462	7654.8	1565.6	6491.8	8778.6	2644.2	4873.6	1741.8
IncMSWBNMB k = 3000	6124.4	9992.6	1492.4	8299.8	9831.2	12155.2	7319.8	10143.33
IncMSWBNMB k = 30000	5906.4	9132.8	1495	8052.2	9952.2	9059.8	5926.6	6413
IncMSWBNMB k = 300	7773.8	12307	2310.6	13493.4	11325	20401	12998.8	8933.33
IncMSWBNMB k = 3000	5961	9784	1703.6	8311	9865.8	13106.8	7432.8	9270.33
IncMSWBNMB k = 30000	5754.8	8833.2	1587.2	8068.2	9013.8	9605.4	6016.2	6262.4
IncMSTWBNMB k = 300 time = 1ms	5654.8	8790	1449.2	8098.2	8894	6160.4	5958.6	5250
IncMSTWBNMB k = 3000 time = 1ms	5445	9484.8	1545.8	8333.6	8937.6	6285.2	6014.6	4990.5
IncMSTWBNMB k = 30000 time = 1ms	5632.4	8616.6	1494	8270	9058	6033.4	6185	5315
IncMSTWBNMB k = 300 time = 5ms	5644.6	9310.6	1587.8	8404.2	9474.6	6330	6159.8	5186.5
IncMSTWBNMB k = 3000 time = 5ms	5612	8747.4	1522.4	8403.8	9686.6	6414.8	6239.6	5561.67
IncMSTWBNMB k = 30000 time = 5ms	5750.4	9276.4	1493.8	8388.2	9115.8	6111.4	6052.6	5432.5
IncMSTWBNMB k = 300 time = 10ms	7130	9538.2	1510.6	8988	9062.8	6479	6575.6	7342.25
IncMSTWBNMB k = 3000 time = 10ms	5524.8	8650.6	1574	8680.6	9449.8	6761.8	6188	5187
IncMSTWBNMB k = 30000 time = 10ms	5660.4	9080.8	1457.8	8907.4	9595.2	6581.2	6099.6	6139
IncMSTWBMB k = 300 time = 1ms	5624.8	9248.8	1586.2	8166.8	9143	7037.2	6122.6	4927.33
IncMSTWBMB k = 3000 time = 1ms	5539.6	9196.2	1602	8449.2	9403.2	6195.2	6074.8	5087.5
IncMSTWBMB k = 30000 time = 1ms	5594.6	9047.4	1612	8538.8	9394.4	6417.4	6129.8	7241.5
IncMSTWBMB k = 300 time = 5ms	5323.8	9577	1504.6	8453	8989.2	6491.4	6293	5378
IncMSTWBMB k = 3000 time = 5ms	5492	8879.6	1572.8	8586	9707.4	6212.2	6169.6	4990
IncMSTWBMB k = 30000 time = 5ms	5638.8	8863.4	1527.8	8519	9045.6	6588.2	6103.4	5127
IncMSTWBMB k = 300 time = 10ms	5408.4	9592.8	1479.8	9061.4	9405.8	6700	6768.2	7043.67
IncMSTWBMB k = 3000 time = 10ms	5642.8	9507.6	1509.6	8871.2	9442.8	6567.4	6362	5582
IncMSTWBMB k = 30000 time = 10ms	5332.4	9219.4	1472.4	8763.4	9122.4	6601	6280.4	5442.5

Table A.1: Running time 1 core

	antlr	bloat	fop	jython	luindex	lusearch	pmd	xalan
MSWB3001 MSWB30001	893.0	851.0	0.0	818.2	996.4	848.8	802.4	20043.6
MSWB300001	6027.0	6283.2	0.0	6242.6	6391.8	5810.0	5958.2	6426.0
MSWBMB3001	186.0	174.4	0.0	176.0	196.4	170.2	170.6	4578.5
MSWBMB30001	895.2	867.8	0.0	812.4	1007.4	845.8	795.8	860.5
MSWBMB300001	6028.6	6258.8	0.0	6198.4	6344.0	5828.6	6045.8	6494.0
MS300TB1mil1	6807.4	15086.2	6459.4	22174.0	10097.6	12808.6	15886.4	16520.5
MS3000TB1mil1	13778.6	25620.2	14385.4	37879.8	16157.6	17398.6	30089.4	26209.666666666668
MS30000TB1mil1	18534.6	27570.4	0.0	40863.6	19065.0	17339.4	36853.6	29933.8
MS300TB5mil1	9560.4	21312.0	8350.0	24751.4	14744.0	16582.4	23789.4	20855.666666666668
MS3000TB5mil1	14047.4	31558.0	0.0	41082.4	18810.6	19514.0	34501.0	29978.666666666668
MS30000TB5mil1	19176.6	31069.4	0.0	44592.4	20588.6	19529.8	39585.6	32957.25
MS300TB10mil1	11533.8	28687.2	4937.0	31640.8	18406.6	19413.8	32424.0	27304.5
MS3000TB10mil1	15502.6	36623.8	0.0	50333.2	21039.2	22477.8	42765.0	37289.333333333336
MS30000TB10mil1	19701.0	37222.2	0.0	46133.8	22009.4	22326.2	42444.2	36577.666666666664
MS300TB1milMB1	6599.0	15319.8	7132.8	22662.0	9734.0	13097.6	15273.2	15645.666666666666
MS3000TB1milMB1	13747.4	26609.8	15172.6	38312.0	16559.2	17737.4	27492.8	26212.666666666668
MS30000TB1milMB1	19078.0	27267.0	26441.0	40967.4	19527.8	17894.8	33795.4	29904.333333333332
MS300TB5milMB1	8750.0	18671.8	8878.4	26427.8	12289.8	16328.0	22979.8	18332.0
MS3000TB5milMB1	14306.6	28453.2	18106.4	45543.6	18857.8	19669.2	32425.0	29663.333333333332
MS30000TB5milMB1	19387.6	29328.6	26492.8	43155.0	21017.0	19668.4	38650.4	32445.666666666668
MS300TB10milMB1	10453.8	24622.0	9811.6	30381.9	17670.5	18473.625	29934.8	25381.0
MS3000TB10milMB1	14746.8	32318.8	18588.6	47778.4	21388.6	22182.0	40936.8	33134.0
MS30000TB10milMB1	19503.0	31945.0	0.0	51546.4	22116.4	22085.6	46993.8	32914.333333333336

Table A.2: 1 core avg pause

	antlr	bloat	fop	jython	luindex	lusearch	pmd	xalan
MSWB3001 MSWB30001	21840.0	45737.0	-1.0	26874.0	29664.0	20821.0	21712.0	79645.0
MSWB300001	21643.0	45405.0	-1.0	26668.0	28453.0	21255.0	21303.0	20308.0
MSWBMB3001	22401.0	47533.0	-1.0	25416.0	31346.0	21503.0	21273.0	79588.0
MSWBMB30001	21751.0	43030.0	-1.0	25856.0	29793.0	20285.0	20571.0	27832.0
MSWBMB300001	21764.0	47701.0	-1.0	26009.0	28674.0	20120.0	20523.0	20592.0
MS300TB1mil1	22952.0	53727.0	26469.0	53858.0	29725.0	22417.0	56569.0	33078.0
MS3000TB1mil1	22315.0	61799.0	25610.0	120266.0	35698.0	37538.0	77490.0	79488.0
MS30000TB1mil1	38993.0	67805.0	-1.0	121601.0	37056.0	37210.0	117989.0	134103.0
MS300TB5mil1	24739.0	58406.0	28481.0	57618.0	35469.0	36062.0	52545.0	45323.0
MS3000TB5mil1	23634.0	67920.0	-1.0	123353.0	36214.0	38166.0	91746.0	83477.0
MS30000TB5mil1	41222.0	69999.0	-1.0	130316.0	38253.0	38297.0	120647.0	82883.0
MS300TB10mil1	27547.0	74850.0	16214.0	75960.0	41922.0	34630.0	81716.0	95069.0
MS3000TB10mil1	31968.0	78677.0	-1.0	277690.0	37894.0	40576.0	103776.0	83220.0
MS30000TB10mil1	41968.0	79991.0	-1.0	132367.0	37693.0	40549.0	101798.0	83838.0
MS300TB1milMB1	22418.0	46245.0	17994.0	55956.0	29202.0	23091.0	31038.0	32100.0
MS3000TB1milMB1	22267.0	63548.0	25989.0	119272.0	34756.0	39201.0	77348.0	84316.0
MS30000TB1milMB1	40260.0	64966.0	71314.0	119898.0	36409.0	74493.0	105508.0	82599.0
MS300TB5milMB1	25056.0	53631.0	28471.0	89914.0	33076.0	26808.0	52779.0	33360.0
MS3000TB5milMB1	25217.0	68712.0	29777.0	137870.0	37150.0	38367.0	88815.0	84264.0
MS30000TB5milMB1	40102.0	66919.0	71469.0	123564.0	43579.0	38866.0	115874.0	91145.0
MS300TB10milMB1	28543.0	57155.0	32012.0	103019.0	39670.0	45708.0	71775.0	42689.0
MS3000TB10milMB1	27850.0	73743.0	21552.0	136548.0	37948.0	40847.0	100050.0	83229.0
MS30000TB10milMB1	40506.0	77425.0	-1.0	239501.0	38667.0	54063.0	124589.0	79359.0

Table A.3: 1 core max pause

A.2 Parallel Incremental Mark Sweep Collector

	antlr	bloat	fop	jython	luindex	lusearch	pmd	xalan
MSWB3002	9609.2	11575.2	1485.2	14125.0	11073.4	26849.4	12985.4	9927.333333333334
MSWB30002	6124.4	9992.6	1492.4	8299.8	9831.2	12155.2	7319.8	10143.333333333334
MSWB300002	5906.4	9132.8	1495.0	8052.2	9952.2	9059.8	5926.6	6413.0
MSWBMB3002	7773.8	12307.0	2310.6	13493.4	11325.0	20401.0	12998.8	8933.333333333334
MSWBMB30002	6237.8	9784.0	1703.6	8311.0	9865.8	13106.8	7432.8	9270.333333333334
MSWBMB300002	5754.8	8833.2	1587.2	8068.2	9013.8	9605.4	6016.2	6262.4
MS300TB1mil2	5654.8	8790.0	1449.2	8098.2	8894.0	6160.4	5958.6	5250.0
MS30000TB1mil2	5445.0	9484.8	1545.8	8333.6	8937.6	6285.2	6014.6	4990.5
MS300000TB1mil2	5632.4	8616.6	1493.4	8270.0	9058.0	6033.4	6185.0	5315.0
MS300TB5mil2	5644.6	9310.6	1587.8	8404.2	9474.6	6330.0	6159.8	5186.5
MS30000TB5mil2	5612.0	8747.4	1522.4	8403.8	9686.6	6414.8	6239.6	5561.666666666667
MS300000TB5mil2	5750.4	9276.4	1493.8	8388.2	9115.8	6111.4	6052.6	5432.5
MS300TB10mil2	7130.0	9538.2	1510.6	8988.0	9062.8	6479.0	6575.6	7342.25
MS30000TB10mil2	5524.8	8650.6	1574.0	8680.6	9449.8	6761.8	6188.0	5187.0
MS300000TB10mil2	5660.4	9080.8	1457.8	8907.4	9595.2	6581.2	6099.6	6139.0
MS300TB1milMB2	5624.8	9248.8	1586.2	8166.8	9143.0	7037.2	6122.6	4927.333333333333
MS30000TB1milMB2	5539.6	9196.2	1602.0	8449.2	9403.2	6195.2	6074.8	5087.5
MS300000TB1milMB2	5594.6	9047.4	1612.0	8538.8	9394.4	6417.4	6129.8	7241.5
MS300TB5milMB2	5323.8	9577.0	1504.6	8453.0	8989.2	6491.4	6293.0	5378.0
MS30000TB5milMB2	5492.0	8879.6	1572.8	8586.0	9707.4	6212.2	6169.6	4990.0
MS300000TB5milMB2	5638.8	8863.4	1527.8	8519.0	9045.6	6588.2	6103.4	5119.75
MS300TB10milMB2	5408.4	9592.8	1479.8	9061.4	9405.8	6700.0	6768.2	7043.666666666667
MS30000TB10milMB2	5642.8	9507.6	1509.6	8871.2	9442.8	6567.4	6362.0	5582.0
MS300000TB10milMB2	5332.4	9219.4	1472.4	8763.4	9122.4	6601.0	6280.4	5442.5

Table A.4: 2 core run time

	antlr	bloat	fop	jython	luindex	lusearch	pmd	xalan
MSWB3002	1694.8	435.2	0.0	301.6	3489.6	3622.4	1604.2	418.3333333333333
MSWB30002	9488.6	2691.6	0.0	1135.8	10018.8	7887.2	4254.4	7500.0
MSWB300002	9619.2	8952.6	0.0	8830.8	6903.0	7891.6	7471.6	6866.5
MSWBMB3002	3916.8	599.4	875.6	254.2	2666.0	2112.4	1568.6	370.3333333333333
MSWBMB30002	9864.6	2764.6	2442.6	1192.4	9404.0	8286.2	4620.2	7300.333333333333
MSWBMB300002	8228.8	9094.0	8888.8	8999.0	7141.4	6729.6	8003.2	7053.0
MS300TB1mil2	7343.4	15957.2	0.0	23110.6	9320.4	10630.2	14849.6	10750.0
MS3000TB1mil2	9837.4	24868.6	21789.4	30715.8	18539.2	13050.4	25661.8	22426.5
MS30000TB1mil2	16187.4	23653.0	0.0	30710.4	16779.8	15243.0	29320.4	24003.333333333332
MS300TB5mil2	6728.0	19022.8	11625.4	27169.4	10598.2	10579.4	20779.2	16305.75
MS3000TB5mil2	10709.4	29780.8	0.0	36216.8	15044.8	14818.2	26628.0	19441.333333333332
MS30000TB5mil2	19190.2	25036.6	0.0	37134.6	19466.2	14434.6	32186.2	23915.5
MS300TB10mil2	7269.4	19825.2	0.0	29728.6	14565.6	12999.2	25016.4	16657.0
MS3000TB10mil2	12487.0	31217.0	19424.6	42211.8	17314.0	14350.4	34910.6	26858.0
MS30000TB10mil2	17377.0	31270.2	0.0	41275.4	20247.8	16714.4	36162.0	27202.0
MS300TB1milMB2	6932.0	15328.6	9085.2	24885.0	9517.8	10571.2	15193.2	12942.0
MS3000TB1milMB2	13463.8	23293.4	19301.2	33548.6	18534.8	13176.6	23017.0	21004.5
MS30000TB1milMB2	16395.2	22165.0	2321.8	31599.4	12932.8	14676.8	26962.2	20619.25
MS300TB5milMB2	7252.4	17579.2	0.0	27653.6	10452.6	12064.2	20927.4	16600.0
MS3000TB5milMB2	10464.4	23869.8	18293.0	34572.0	20667.6	14803.4	25578.2	19409.333333333332
MS30000TB5milMB2	18106.2	22461.0	0.0	35719.2	17278.8	15634.8	30178.2	21000.25
MS300TB10milMB2	8218.0	21032.8	0.0	31845.2	12124.6	12420.8	27041.0	14818.666666666666
MS3000TB10milMB2	13720.6	26258.2	0.0	40422.4	19859.8	15562.2	35111.0	23058.6
MS30000TB10milMB2	14750.8	27079.8	0.0	40534.0	19733.4	15078.4	33880.0	21498.0

Table A.5: 2 core avg time

	antlr	bloat	fop	jython	luindex	lusearch	pmd	xalan
MSWB3002	1124871.0	662331.0	-1.0	366921.0	1617680.0	3927682.0	1800676.0	452855.0
MSWB30002	237907.0	167351.0	-1.0	83861.0	253435.0	250222.0	292125.0	361366.0
MSWB300002	44382.0	58001.0	-1.0	47134.0	49747.0	41535.0	42446.0	36235.0
MSWBMB3002	1301974.0	1514314.0	988079.0	328991.0	1656543.0	4247603.0	1392350.0	1021215.0
MSWBMB30002	246112.0	217795.0	168765.0	52919.0	257800.0	254819.0	290387.0	357071.0
MSWBMB300002	43403.0	57487.0	23644.0	45451.0	48435.0	42040.0	46926.0	59735.0
MS300TB1mil2	44623.0	44000.0	-1.0	66530.0	23753.0	41330.0	40423.0	48321.0
MS3000TB1mil2	36626.0	75134.0	33291.0	132375.0	44737.0	50560.0	104447.0	96979.0
MS30000TB1mil2	48528.0	77612.0	-1.0	133001.0	44653.0	51256.0	124192.0	95909.0
MS300TB5mil2	21711.0	50433.0	23367.0	70673.0	28414.0	40500.0	51912.0	53580.0
MS3000TB5mil2	35947.0	75753.0	-1.0	133233.0	44323.0	50736.0	112288.0	94615.0
MS30000TB5mil2	53625.0	72842.0	-1.0	133316.0	46486.0	50311.0	129320.0	96220.0
MS300TB10mil2	22410.0	62488.0	-1.0	77599.0	38125.0	46084.0	57445.0	51351.0
MS3000TB10mil2	38271.0	73441.0	42043.0	157273.0	45306.0	50398.0	104525.0	99099.0
MS30000TB10mil2	50537.0	74120.0	-1.0	157532.0	44661.0	50991.0	116700.0	96096.0
MS300TB1milMB2	34864.0	39911.0	21568.0	66175.0	27842.0	40001.0	38675.0	46397.0
MS3000TB1milMB2	37083.0	75606.0	33525.0	133340.0	49560.0	52645.0	104246.0	98763.0
MS30000TB1milMB2	49585.0	77567.0	94564.0	133904.0	45118.0	52696.0	125311.0	96145.0
MS300TB5milMB2	17855.0	44822.0	-1.0	71769.0	26487.0	45749.0	53563.0	53131.0
MS3000TB5milMB2	36805.0	74988.0	37514.0	134132.0	46956.0	53549.0	106122.0	94272.0
MS30000TB5milMB2	48380.0	78287.0	99279.0	132035.0	53005.0	53124.0	128738.0	99373.0
MS300TB10milMB2	19935.0	51666.0	-1.0	85985.0	36367.0	44813.0	67370.0	49728.0
MS3000TB10milMB2	42120.0	76589.0	-1.0	131933.0	47297.0	52244.0	118876.0	94535.0
MS30000TB10milMB2	49938.0	75611.0	-1.0	175221.0	47386.0	53180.0	114361.0	95282.0

Table A.6: 2 core max pause

	antlr	bloat	fop	jython	luindex	lusearch	pmd	xalan
MSWB3004	5580.2	11595.2	1494.0	12114.8	10225.0	12359.2	7380.8	7950.0
MSWB30004	5702.4	8905.0	1622.4	8422.0	9313.0	10268.4	5977.0	6315.333333333333
MSWB300004	5773.8	9105.0	1514.6	7827.0	9079.0	8901.2	5812.6	5957.5
MSWBMB3004	5680.2	12286.8	1793.4	13505.2	9975.0	12048.6	7516.2	8134.2
MSWBMB30004	5557.2	9723.2	1485.8	8532.0	9095.8	9935.4	6205.4	6645.0
MSWBMB300004	5548.6	9191.2	1494.2	8129.4	8898.0	9493.0	5856.4	6388.333333333333
MS300TB1mil4	5403.4	9028.2	1551.6	7990.8	9595.8	5828.4	6064.0	4555.75
MS3000TB1mil4	5511.8	8927.0	1498.8	7838.4	9156.8	6012.6	5839.6	5635.5
MS30000TB1mil4	5567.0	8887.8	1550.8	7621.6	8937.6	5869.8	5881.4	7049.0
MS300TB5mil4	5334.2	8935.8	1478.4	8004.4	9045.0	6143.6	5937.2	7177.2
MS3000TB5mil4	5230.0	8383.0	1597.4	7778.6	8877.8	6084.0	5791.6	5056.2
MS30000TB5mil4	5518.2	8715.4	1496.6	7905.2	9305.0	6113.4	5837.0	5093.0
MS300TB10mil4	5404.8	9035.0	1500.4	8158.2	9187.4	6917.6	6126.4	4996.0
MS3000TB10mil4	6391.2	8978.6	1509.6	8081.2	9115.6	6142.8	5989.2	7038.75
MS30000TB10mil4	5644.2	9057.0	1507.4	8094.4	9420.4	6488.8	5954.6	5157.666666666667
MS300TB1milMB4	5565.0	9038.0	1593.8	7831.6	9601.0	6260.8	5917.6	4782.666666666667
MS3000TB1milMB4	5402.4	8828.2	1512.0	7700.0	9032.2	5992.4	5925.4	5099.8
MS30000TB1milMB4	6544.8	8756.4	1476.4	7815.8	9008.0	6350.2	5810.2	4731.0
MS300TB5milMB4	5601.2	9110.0	1480.8	8055.4	9104.0	6407.0	6129.8	5052.5
MS3000TB5milMB4	5589.0	9190.0	1522.4	7932.6	9039.4	6265.6	5931.4	4847.75
MS30000TB5milMB4	5433.8	9092.4	1484.0	8002.4	9083.8	6341.4	5986.4	5122.5
MS300TB10milMB4	5478.4	9288.2	1494.6	8213.8	9577.6	6164.6	6275.2	4917.666666666667
MS3000TB10milMB4	5661.2	9417.0	1579.2	8269.2	9079.4	6323.6	6228.4	6829.0
MS30000TB10milMB4	5394.8	9320.0	1589.2	8228.6	9540.8	6397.6	6172.4	5074.5

Table A.7: 4 core run time

	antlr	bloat	fop	jython	luindex	lusearch	pmd	xalan
MSWB3004	198959.0	641389.0	-1.0	506095.0	197367.0	520347.0	896672.0	169130.0
MSWB30004	80915.0	109890.0	103518.0	194747.0	102312.0	110110.0	128859.0	160093.0
MSWB300004	26176.0	55865.0	-1.0	56308.0	31880.0	29202.0	77485.0	34801.0
MSWBMB3004	124843.0	737046.0	1050936.0	840179.0	253510.0	427780.0	911123.0	234612.0
MSWBMB30004	121799.0	146241.0	-1.0	190883.0	107026.0	131888.0	194678.0	151892.0
MSWBMB300004	26661.0	49554.0	-1.0	49911.0	29473.0	24244.0	26770.0	34747.0
MS300TB1mil4	32914.0	57413.0	24364.0	95211.0	31281.0	42333.0	71278.0	109308.0
MS3000TB1mil4	49313.0	59770.0	-1.0	101388.0	37995.0	42559.0	102819.0	73111.0
MS30000TB1mil4	44131.0	65383.0	75412.0	103633.0	38989.0	42965.0	96689.0	72411.0
MS300TB5mil4	33040.0	59450.0	-1.0	100923.0	35006.0	43196.0	73415.0	63349.0
MS3000TB5mil4	42487.0	60364.0	72001.0	100946.0	39829.0	43758.0	98323.0	69801.0
MS30000TB5mil4	42502.0	61653.0	-1.0	101420.0	39784.0	44678.0	98327.0	73821.0
MS300TB10mil4	27157.0	63658.0	-1.0	120874.0	34009.0	43523.0	87504.0	72680.0
MS3000TB10mil4	39064.0	61804.0	-1.0	102847.0	38681.0	42335.0	102717.0	73036.0
MS30000TB10mil4	43152.0	61317.0	-1.0	109013.0	37698.0	43707.0	97314.0	68612.0
MS300TB1milMB4	34748.0	61392.0	24269.0	102207.0	33934.0	43688.0	70597.0	68458.0
MS3000TB1milMB4	46083.0	63202.0	-1.0	103280.0	41947.0	45377.0	90078.0	70992.0
MS30000TB1milMB4	44261.0	65136.0	-1.0	102518.0	39113.0	64738.0	83392.0	71883.0
MS300TB5milMB4	38308.0	63586.0	-1.0	102139.0	33138.0	46315.0	75673.0	67738.0
MS3000TB5milMB4	43060.0	66832.0	-1.0	103810.0	41690.0	44293.0	82049.0	68819.0
MS30000TB5milMB4	43761.0	64809.0	-1.0	103094.0	41817.0	45365.0	102381.0	73086.0
MS300TB10milMB4	34228.0	62171.0	-1.0	104232.0	35363.0	45176.0	79161.0	71663.0
MS3000TB10milMB4	42510.0	63114.0	82538.0	120140.0	41260.0	45440.0	105415.0	69136.0
MS30000TB10milMB4	41244.0	62774.0	78907.0	113449.0	42088.0	45680.0	104126.0	69824.0

Table A.8: 4 core max pause

	antlr	bloat	fop	jython	luindex	lusearch	pmd	xalan
MSWB3004	731.2	1561.8	0.0	652.6	1045.0	1284.6	622.4	502.0
MSWB30004	3222.2	2924.8	3203.6	2895.8	4056.0	3732.8	2681.6	3156.3333333333335
MSWB300004	6602.8	10084.2	0.0	10075.6	7790.4	7925.0	10419.4	8825.0
MSWBMB3004	791.6	1378.4	0.0	838.2	1000.0	851.4	697.8	480.2
MSWBMB30004	4337.4	3007.4	0.0	2845.8	3646.2	3897.0	2971.0	3208.25
MSWBMB300004	7485.0	10104.8	0.0	9993.0	6716.4	7859.2	9487.6	7972.666666666667
MS300TB1mil4	8321.8	17748.8	11151.0	21871.6	10567.8	13572.4	15743.8	16725.0
MS3000TB1mil4	11563.2	22176.6	0.0	29029.4	14777.8	14133.6	24958.4	20033.5
MS30000TB1mil4	13618.0	21268.8	29853.8	26909.2	13937.4	13051.8	24270.4	19874.0
MS300TB5mil4	9143.2	19989.4	0.0	26937.0	11744.8	14677.8	19572.6	16099.0
MS3000TB5mil4	14359.6	23114.8	21641.0	30496.6	14973.0	14417.6	28568.0	20749.0
MS30000TB5mil4	13255.2	23917.0	0.0	33818.0	15880.2	15262.2	28038.2	20125.25
MS300TB10mil4	8658.2	26234.8	0.0	27535.0	11826.6	15535.6	25241.2	19571.0
MS3000TB10mil4	14105.0	24492.4	0.0	35143.0	15077.4	15368.4	30059.8	21876.25
MS30000TB10mil4	16376.4	26284.6	0.0	36436.4	15564.2	15850.0	31962.2	19260.666666666668
MS300TB1milMB4	9287.2	16992.6	12374.0	23290.4	11211.0	13301.2	17132.4	15964.666666666666
MS3000TB1milMB4	16039.2	22309.6	0.0	30804.4	13908.4	14482.0	25133.6	19897.4
MS30000TB1milMB4	14770.0	21233.8	0.0	29096.2	15336.6	14008.0	26709.4	20862.4
MS300TB5milMB4	8441.2	19112.8	0.0	27966.6	11378.2	15405.6	20868.0	14346.0
MS3000TB5milMB4	14892.6	22897.2	0.0	35600.6	15390.6	14792.4	26107.0	21336.75
MS30000TB5milMB4	14683.2	24563.6	0.0	33581.4	13736.8	15100.6	29259.8	20985.5
MS300TB10milMB4	10455.6	21345.0	0.0	32436.8	13306.0	15545.2	25141.0	17199.666666666668
MS3000TB10milMB4	15801.0	24156.8	28497.0	37417.4	16071.6	15810.4	31223.2	22619.0
MS30000TB10milMB4	13528.2	25420.6	18612.2	40450.2	17034.4	16195.0	35402.8	21540.0

Table A.9: 4 Core Avg

	antlr	bloat	fop	jython	luindex	lusearch	pmd	xalan
MSWB3008	5651.0	9658.0	1503.0	9546.6	9369.4	11075.8	6729.0	7599.8
MSWB30008	6500.8	8617.6	1499.6	8194.2	9323.4	9155.6	6066.2	6482.333333333333
MSWB300008	5829.6	8975.4	1587.4	7873.8	9102.6	9063.4	5863.8	6210.0
MSWBMB3008	5579.2	10137.8	1682.6	10266.4	9598.8	10960.0	6909.8	7531.0
MSWBMB30008	5371.6	8941.2	1481.8	7972.4	9090.6	9788.0	5999.6	6764.0
MSWBMB300008	7707.2	9064.4	1503.0	7690.2	9350.2	9514.0	5942.8	7530.0
MS300TB1mil8	5573.6	8734.4	1445.0	7706.0	9407.0	5949.6	5943.0	4614.4
MS3000TB1mil8	5214.8	8883.6	1478.8	7618.2	9038.2	6001.8	5910.4	4616.5
MS30000TB1mil8	6983.4	9003.6	1464.2	7823.8	9569.6	5930.6	5958.8	5076.666666666667
MS300TB5mil8	5532.2	8828.2	1490.0	7862.4	9065.8	6055.8	5990.0	5146.666666666667
MS3000TB5mil8	5333.6	8850.4	1506.2	8040.0	9034.4	6359.6	5939.2	4666.0
MS30000TB5mil8	5580.6	8655.6	1546.4	7813.6	9287.0	6158.2	5985.6	5110.666666666667
MS300TB10mil8	5706.0	8971.6	1562.0	7846.2	9327.0	6254.4	6174.0	4938.333333333333
MS3000TB10mil8	5510.4	8654.8	1483.8	8188.8	9460.0	6222.6	6095.6	4803.666666666667
MS30000TB10mil8	5559.4	8631.0	1471.0	8084.8	9435.6	6304.6	6101.4	4798.6
MS300TB1milMB8	6058.8	9043.8	1496.6	7743.6	9830.8	6382.8	5992.0	5629.8
MS3000TB1milMB8	5410.0	8551.8	1519.0	8093.4	9437.8	6424.0	5926.6	7140.0
MS30000TB1milMB8	5373.6	9117.8	1454.0	7883.2	9416.4	6240.0	5930.4	6764.0
MS300TB5milMB8	5213.6	8631.0	1553.0	7937.8	9137.4	7496.6	5890.4	4662.5
MS3000TB5milMB8	6449.2	8804.2	1596.0	8027.8	9166.0	6215.2	5964.6	4817.5
MS30000TB5milMB8	7106.6	9169.0	1526.8	7935.4	8939.2	6298.8	6007.8	4739.75
MS300TB10milMB8	5491.0	9222.4	1580.2	8281.8	9034.2	6565.2	6151.6	6372.0
MS3000TB10milMB8	5624.6	9279.4	1469.0	8111.0	9053.6	6449.4	6131.8	4939.5
MS30000TB10milMB8	6084.8	9137.6	1525.2	8180.6	9236.8	6395.6	6202.6	5294.0

Table A.10: 8 Core Run Time

	antlr	bloat	fop	jython	luindex	lusearch	pmd	xalan
MSWB3008	300155.0	314612.0	-1.0	127310.0	135136.0	131057.0	151970.0	296475.0
MSWB30008	36974.0	84020.0	-1.0	114324.0	71582.0	67831.0	120200.0	103972.0
MSWB300008	32117.0	46460.0	56430.0	53658.0	29804.0	36763.0	43497.0	40068.0
MSWBMB3008	76913.0	342150.0	89250.0	351423.0	130661.0	125683.0	178550.0	259355.0
MSWBMB30008	51193.0	119778.0	-1.0	91809.0	68944.0	70710.0	115173.0	98774.0
MSWBMB300008	38336.0	48027.0	-1.0	53117.0	34586.0	36497.0	42779.0	43354.0
MS300TB1mil8	65950.0	57624.0	-1.0	110931.0	39165.0	41633.0	88295.0	73440.0
MS3000TB1mil8	49669.0	83578.0	-1.0	109312.0	37882.0	41441.0	98156.0	67662.0
MS30000TB1mil8	48443.0	87481.0	-1.0	100277.0	45887.0	41728.0	85108.0	66800.0
MS300TB5mil8	36316.0	59829.0	25925.0	99390.0	38374.0	40263.0	97645.0	66647.0
MS3000TB5mil8	40133.0	58786.0	-1.0	106687.0	37795.0	41091.0	100223.0	68896.0
MS30000TB5mil8	40891.0	61779.0	79401.0	100217.0	36184.0	54532.0	109380.0	67382.0
MS300TB10mil8	39277.0	69634.0	71175.0	114531.0	38331.0	43377.0	99228.0	67404.0
MS3000TB10mil8	38147.0	60315.0	-1.0	105540.0	36587.0	42629.0	101539.0	65362.0
MS30000TB10mil8	40510.0	88059.0	-1.0	125448.0	37332.0	41830.0	101883.0	66524.0
MS300TB1milMB8	56091.0	73942.0	-1.0	110357.0	43038.0	43407.0	87304.0	69318.0
MS3000TB1milMB8	54491.0	64583.0	-1.0	101416.0	40204.0	43323.0	86801.0	65473.0
MS30000TB1milMB8	56962.0	81021.0	-1.0	140419.0	39954.0	47131.0	91834.0	64970.0
MS300TB5milMB8	53714.0	64719.0	55869.0	108589.0	47797.0	43459.0	85706.0	68349.0
MS3000TB5milMB8	44323.0	61760.0	60204.0	102215.0	40355.0	42041.0	99687.0	67544.0
MS30000TB5milMB8	64670.0	68170.0	-1.0	120004.0	38761.0	44955.0	84990.0	71382.0
MS300TB10milMB8	37516.0	65162.0	54136.0	117469.0	39797.0	45247.0	106339.0	57152.0
MS3000TB10milMB8	43961.0	63190.0	-1.0	108593.0	39119.0	44423.0	88022.0	66904.0
MS30000TB10milMB8	55583.0	74978.0	-1.0	102153.0	67705.0	63955.0	108332.0	71513.0

Table A.11: 8 Core Max Time

	antlr	bloat	fop	jython	luindex	lusearch	pmd	xalan
MSWB3008	1192.0	1576.6	0.0	686.8	1460.0	979.4	759.8	961.0
MSWB30008	5211.6	4513.6	0.0	4933.6	6632.0	4343.4	5977.4	4470.333333333333
MSWB300008	10922.6	13142.6	17288.4	14024.4	10296.8	10489.0	13296.0	10115.0
MSWBMB3008	1085.4	1309.0	983.8	772.4	1590.2	1260.4	684.2	861.2
MSWBMB30008	5147.6	4949.0	0.0	4591.4	5942.4	5558.0	4197.2	4777.75
MSWBMB300008	11769.8	12344.2	0.0	13671.4	10363.0	10630.0	11329.0	10505.0
MS300TB1mil8	15405.2	22234.0	0.0	31363.0	14120.6	13941.0	23804.6	20338.6
MS3000TB1mil8	16097.2	22480.4	0.0	29631.2	14834.4	14579.4	25586.2	19952.0
MS30000TB1mil8	15211.4	23546.2	0.0	30307.4	14380.6	13323.6	25628.8	19332.333333333332
MS300TB5mil8	11182.8	23527.2	0.0	32582.8	14756.8	15247.8	25409.8	20969.333333333332
MS3000TB5mil8	16435.2	25259.6	0.0	32310.6	13790.0	15377.0	28099.8	20878.75
MS30000TB5mil8	14802.8	23993.2	29172.6	30035.6	14810.0	15595.6	27678.2	20893.666666666668
MS300TB10mil8	13005.2	27710.0	15525.4	35790.4	14770.4	17319.2	30656.6	20108.666666666668
MS3000TB10mil8	14553.8	27436.2	0.0	35582.6	14305.0	17052.4	33550.8	22431.333333333332
MS30000TB10mil8	15323.4	27973.4	0.0	36807.0	15242.8	17076.2	30759.2	21540.6
MS300TB1milMB8	12594.6	22578.8	0.0	30676.2	14883.2	14377.0	27142.2	20825.8
MS3000TB1milMB8	17034.0	21719.2	0.0	29125.6	14956.0	14719.6	25384.6	19505.5
MS30000TB1milMB8	15224.0	24531.6	0.0	30473.6	14568.8	14851.0	27701.6	20156.5
MS300TB5milMB8	13424.6	22601.8	18584.8	34662.0	15277.2	15747.2	25832.2	20769.25
MS3000TB5milMB8	16825.0	23657.6	25111.0	31610.6	14954.8	14805.6	28410.2	22372.0
MS30000TB5milMB8	15513.8	23719.0	0.0	30756.2	15075.8	15166.6	27895.6	22252.0
MS300TB10milMB8	13607.6	25467.6	17802.2	35545.4	16399.2	17238.4	33005.0	20635.0
MS3000TB10milMB8	16292.0	24478.0	0.0	35708.2	16695.8	16006.4	30686.8	22402.5
MS30000TB10milMB8	14339.8	25145.4	0.0	36427.2	17615.8	16257.8	29966.6	22198.2

Table A.12: 8 Core Avg Time

A.3 Incremental Copying Collector

	antlr	bloat	fop	jython	luindex	lusearch	pmd	xalan
SS30000TB1mil	6239.8	22703.0	2054.0	17226.2	16879.6	NaN	13782.8	NaN
SS30000TB5mil	7570.6	24056.8	2050.0	16862.2	18023.0	NaN	15816.8	NaN
SS30000TB10mil	6172.2	22548.2	2069.0	NaN	16794.2	NaN	13788.0	NaN
SS3000TB1mil	6351.2	21067.4	2077.6	16310.6	17082.8	NaN	13026.2	NaN
SS3000TB5mil	6265.4	21679.4	2016.2	16745.4	18689.4	NaN	13385.6	NaN
SS3000TB10mil	6518.6	21719.2	1966.8	14039.0	17451.2	NaN	13537.8	NaN
SS300TB1mil	6388.4	21603.9	2001.8	19293.3	17940.5	NaN	14622.4	NaN
SS300TB5mil	6246.6	21786.0	2070.8	16126.6	18136.8	NaN	13326.4	NaN
SS300TB10mil	6107.4	21999.8	2040.4	16212.2	17978.8	NaN	13460.6	NaN
SSWB300	7215.4	21544.9	2034.6	16410.9	17330.1	NaN	13132.8	NaN
SSWB3000	6030.0	22129.2	2048.6	16183.4	17816.2	NaN	14519.4	NaN
SSWB30000	6539.8	22652.0	2010.6	NaN	16899.4	NaN	15772.8	NaN

Table A.13: Running Time 1 core

	antlr	bloat	fop	jython	luindex	lusearch	pmd	xalan
SS30000TB1mil	36666.4	58398.6	67199.0	93145.8	39209.4	NaN	69907.8	NaN
SS30000TB5mil	36469.4	57181.0	69947.6	88296.2	38242.6	NaN	65870.2	NaN
SS30000TB10mil	37718.8	58650.2	63250.4	NaN	38695.0	NaN	65934.6	NaN
SS3000TB1mil	22991.6	41175.8	31814.0	74840.6	26955.0	NaN	42668.6	NaN
SS3000TB5mil	21786.0	40238.6	29105.4	79801.0	26941.6	NaN	45911.6	NaN
SS3000TB10mil	23094.2	40859.6	29878.4	48328.5	25529.2	NaN	45583.8	NaN
SS300TB1mil	15643.2	30326.8	15912.1	43805.6	20851.6	NaN	19918.0	NaN
SS300TB5mil	9841.4	20960.6	11523.0	34612.0	14179.6	NaN	21854.2	NaN
SS300TB10mil	9463.6	20580.0	11830.8	34322.0	14175.0	NaN	21681.4	NaN
SSWB300	9815.1	20962.5	10818.0	36162.9	14036.7	NaN	22001.8	NaN
SSWB3000	21933.2	40632.4	28516.0	73907.8	26333.0	NaN	42961.4	NaN
SSWB30000	37088.2	59632.2	68604.2	NaN	38644.2	NaN	64859.0	NaN

Table A.14: Avg pause times 1 core

	antlr	bloat	fop	jython	luindex	lusearch	pmd	xalan
SS30000TB1mil	83465.0	169065.0	163029.0	646120.0	104511.0	-1.0	205569.0	-1.0
SS30000TB5mil	82173.0	151885.0	171601.0	225761.0	76085.0	-1.0	209030.0	-1.0
SS30000TB10mil	81632.0	121138.0	146868.0	-1.0	77727.0	-1.0	216293.0	-1.0
SS3000TB1mil	51557.0	115709.0	67115.0	258981.0	73621.0	-1.0	126687.0	-1.0
SS3000TB5mil	51546.0	116571.0	61484.0	307105.0	72894.0	-1.0	132039.0	-1.0
SS3000TB10mil	51199.0	116613.0	60158.0	135406.0	74682.0	-1.0	147610.0	-1.0
SS300TB1mil	47155.0	116093.0	49416.0	456634.0	90681.0	-1.0	64370.0	-1.0
SS300TB5mil	26605.0	44398.0	31183.0	107581.0	31298.0	-1.0	58761.0	-1.0
SS300TB10mil	26833.0	43978.0	31228.0	113532.0	31227.0	-1.0	62395.0	-1.0
SSWB300	27223.0	44854.0	31602.0	109011.0	31527.0	-1.0	65441.0	-1.0
SSWB3000	46942.0	118313.0	64049.0	221830.0	72653.0	-1.0	128011.0	-1.0
SSWB30000	84479.0	289000.0	168323.0	46198.0	78667.0	-1.0	186698.0	-1.0

Table A.15: Max pause times 1 core

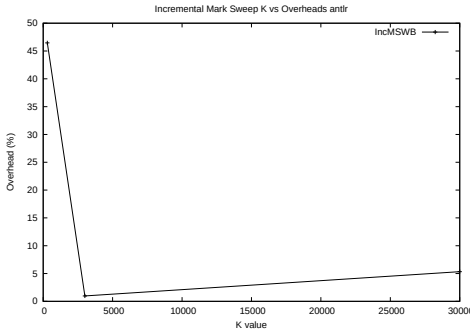
Appendix B

Evaluation Graphs

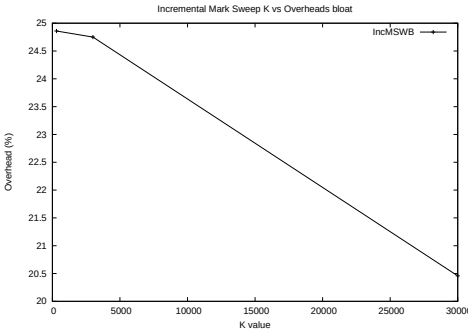
Note some graphs have been omitted for brevity

B.1 Overhead, Average and Max Pause Graphs

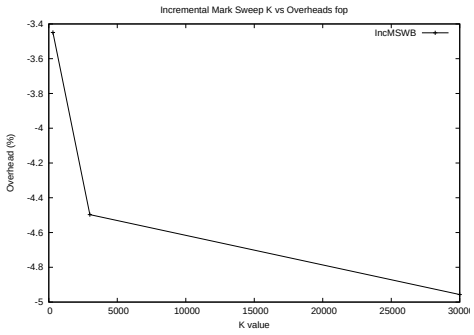
B.1.1 Incremental Mark-Sweep



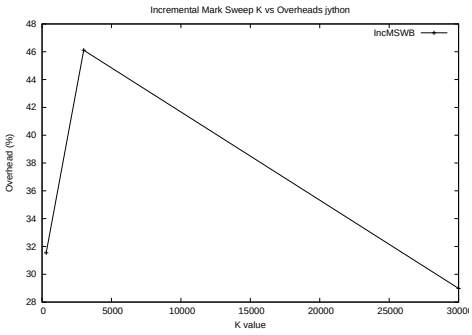
(a) antlr overheads for different k's



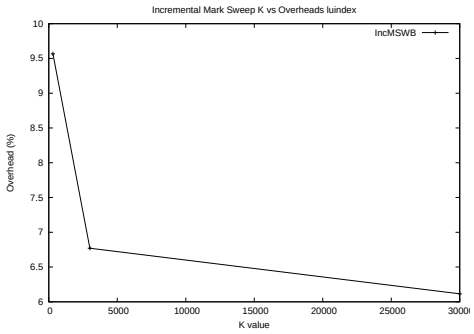
(b) bloat overheads for different k's



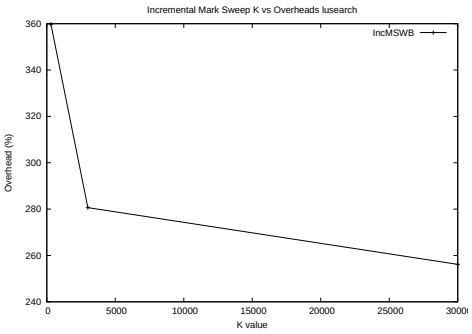
(c) fop overheads for different k's



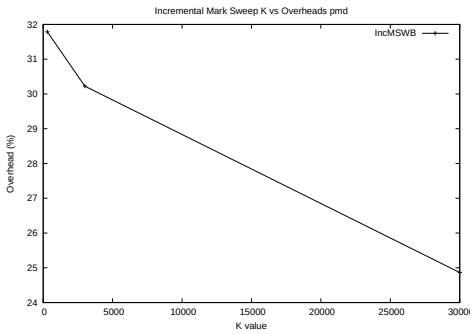
(d) jython overhead for different k's



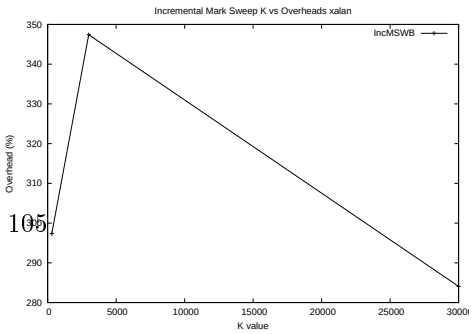
(e) luindex overhead for different k's



(f) lusearch overheads for different k's

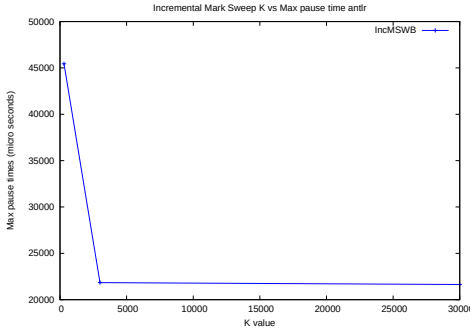


(g) pmd overhead for different k's

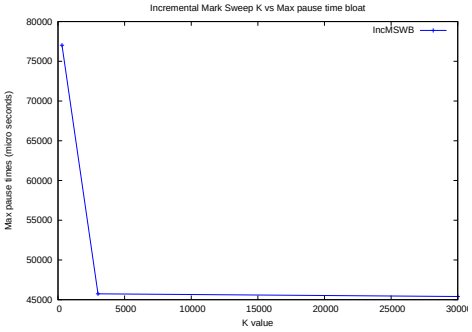


(h) xalan overhead for different k's

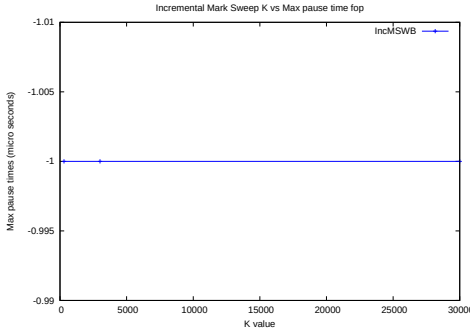
Figure B.1: Graphed results for overhead on single core



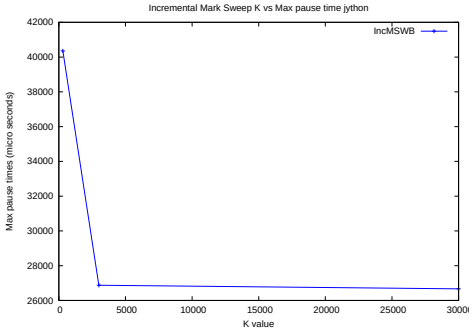
(a) antlr maximum pause time graph



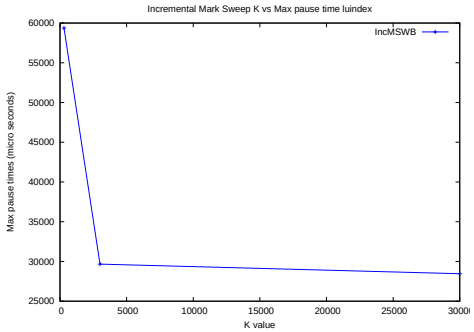
(b) bloat maximum pause times



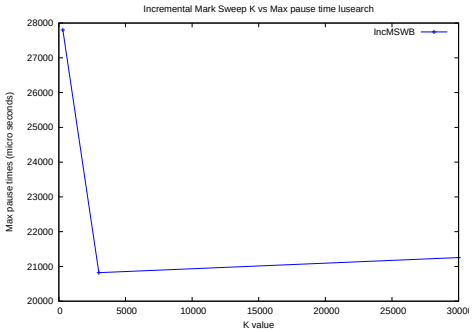
(c) fop maximum pause times



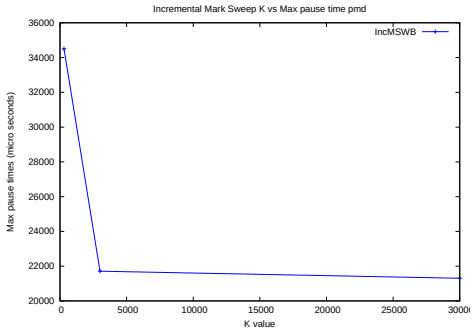
(d) jython maximum pause times



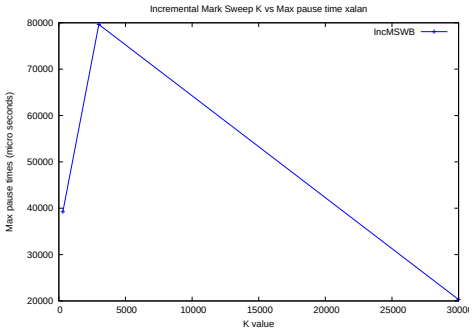
(e) luindex maximum pause times



(f) lusearch maximum pause times

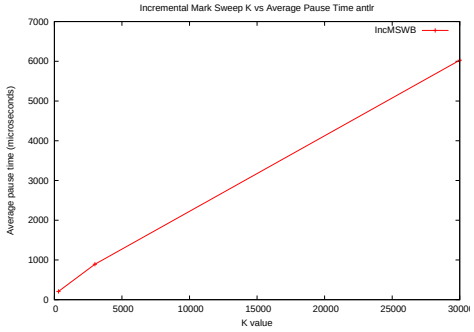


(g) pmd maximum pause times

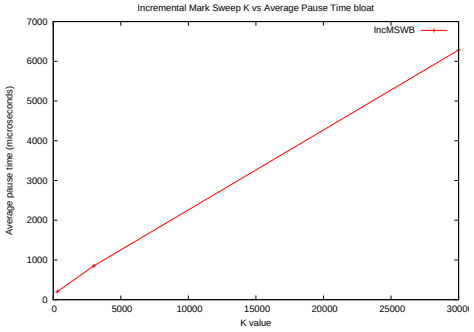


(h) xalan maximum pause times

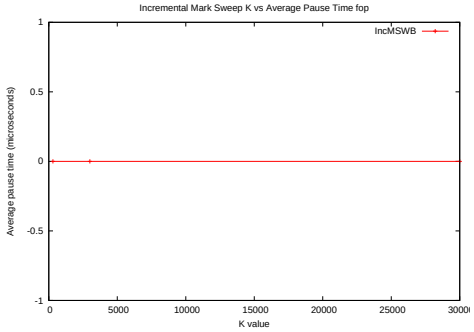
Figure B.2: Graphed results for maximum pause time on single core



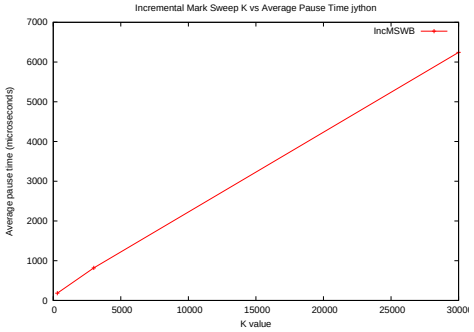
(a) antlr average pause times



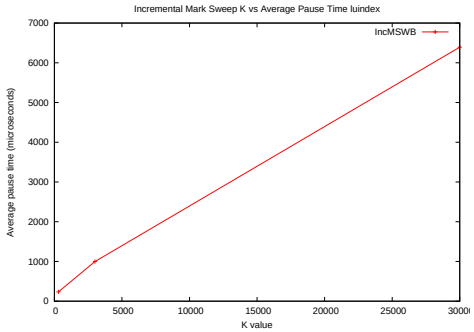
(b) bloat average pause times



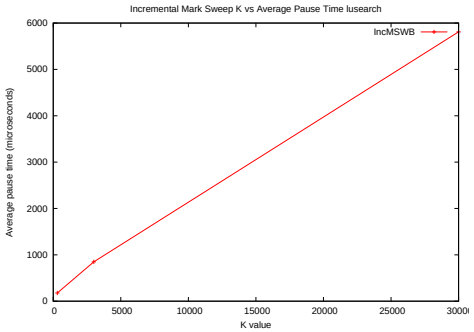
(c) fop average pause times



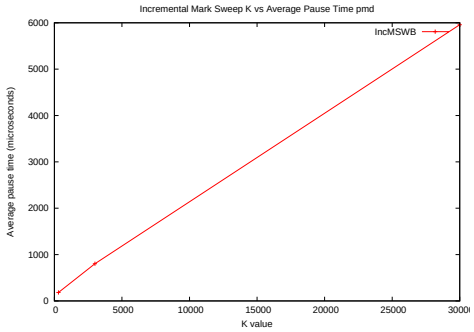
(d) jython average pause times



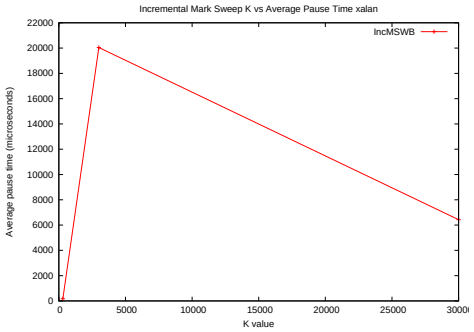
(e) luindex average pause times



(f) lusearch average pause times



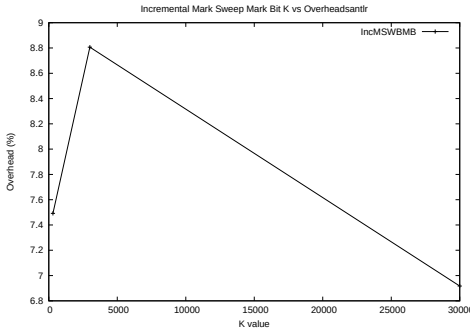
(g) pmd average pause times



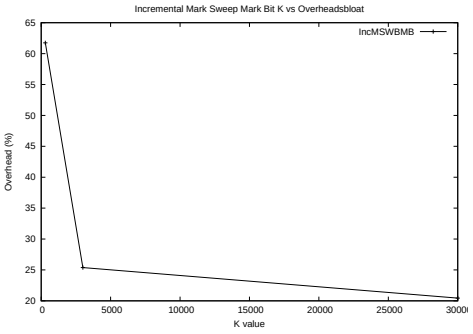
(h) xalan average pause times

Figure B.3: Graphed results for average pause times

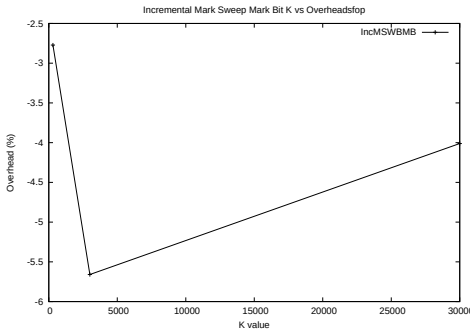
B.1.2 Incremental Mark-Sweep Mark Bit



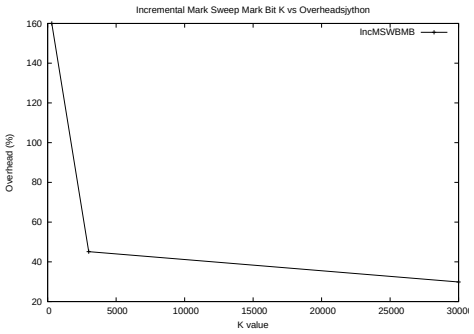
(a) antlr overheads for different k's



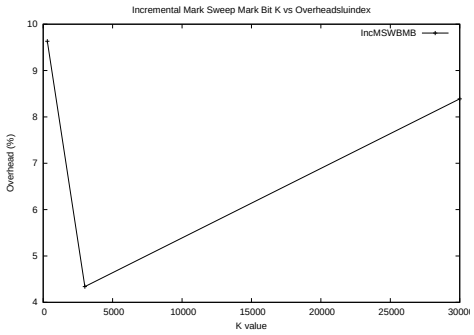
(b) bloat overheads for different k's



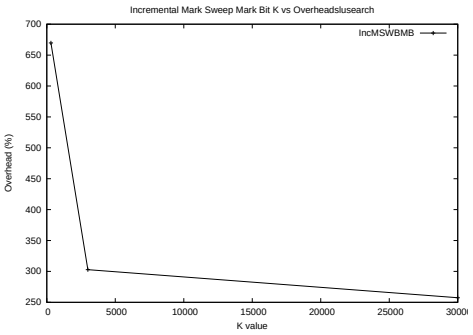
(c) fop overheads for different k's



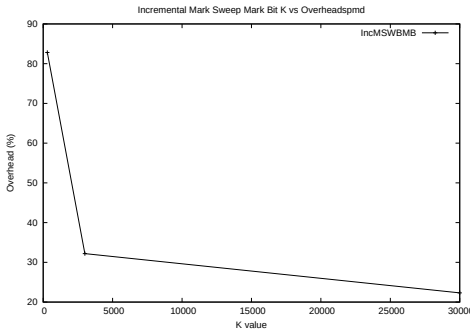
(d) jython overhead for different k's



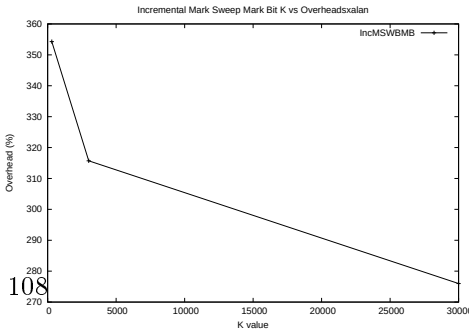
(e) luindex overhead for different k's



(f) lusearch overheads for different k's

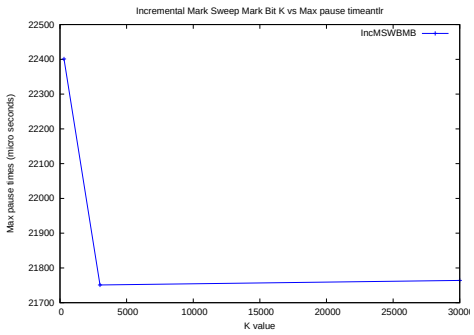


(g) pmd overhead for different k's

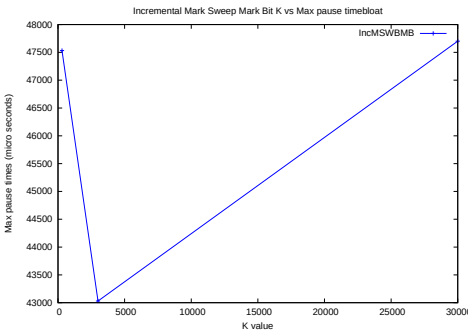


(h) xalan overhead for different k's

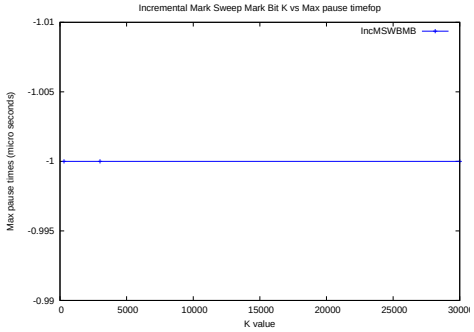
Figure B.4: Graphed results for overhead on single core



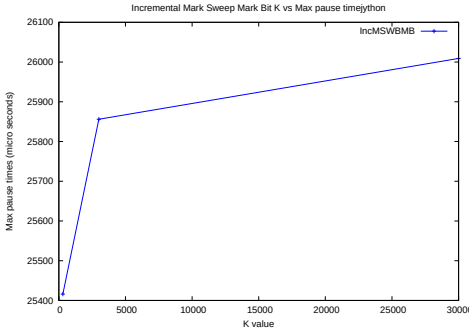
(a) antlr maximum pause time graph



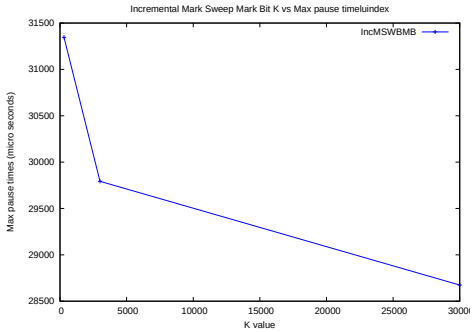
(b) bloat maximum pause times



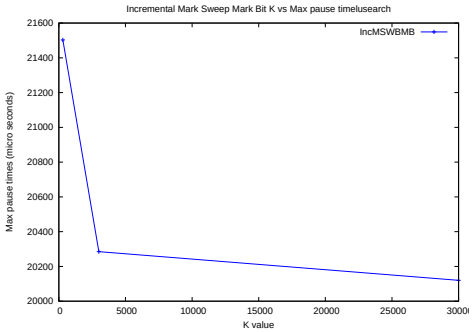
(c) fop maximum pause times



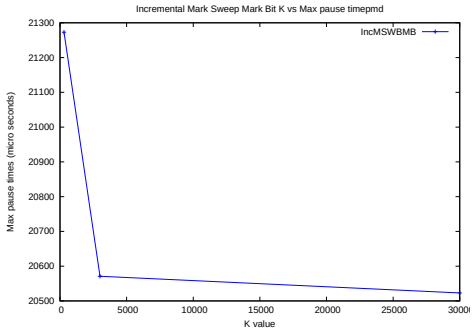
(d) jython maximum pause times



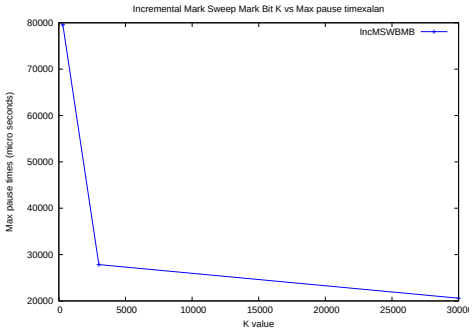
(e) luindex maximum pause times



(f) lusearch maximum pause times

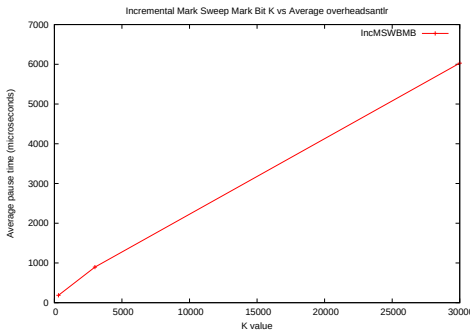


(g) pmd maximum pause times

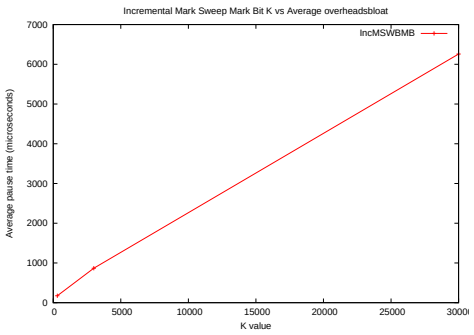


(h) xalan maximum pause times

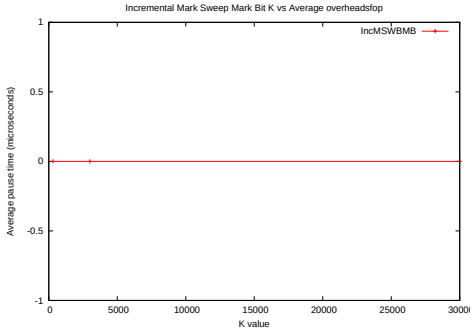
Figure B.5: Graphed results for maximum pause time on single core



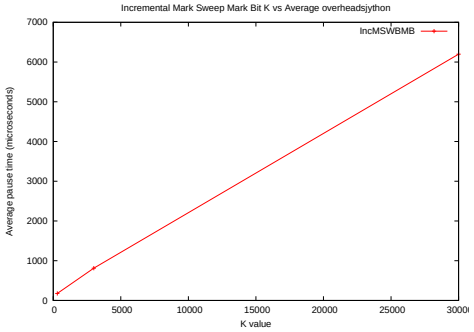
(a) antlr average pause times



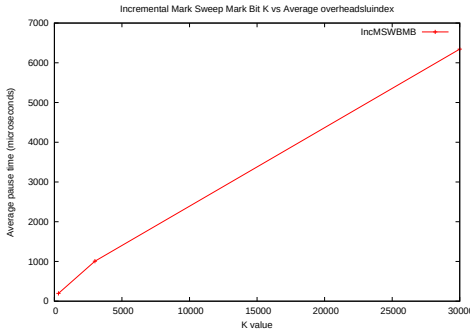
(b) bloat average pause times



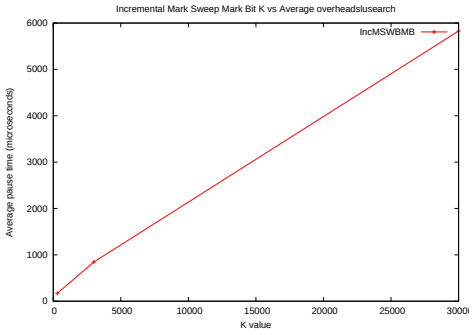
(c) fop average pause times



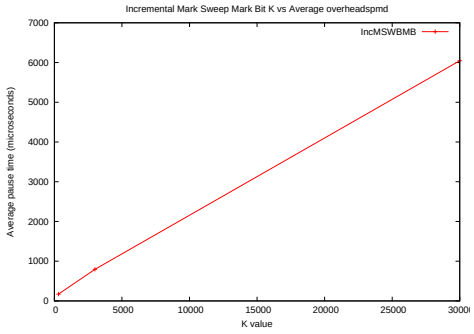
(d) jython average pause times



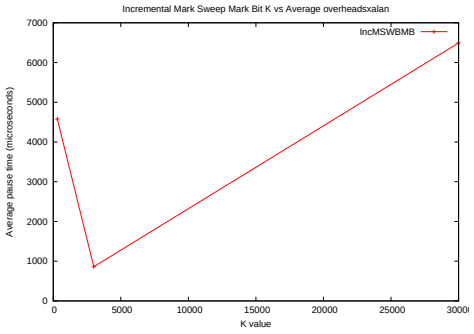
(e) luindex average pause times



(f) lusearch average pause times



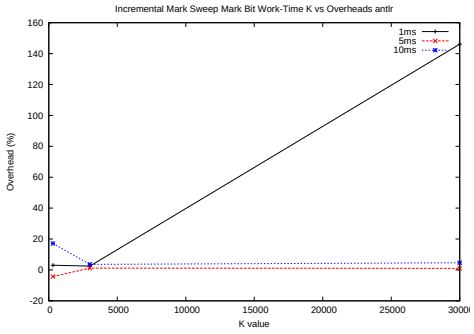
(g) pmd average pause times



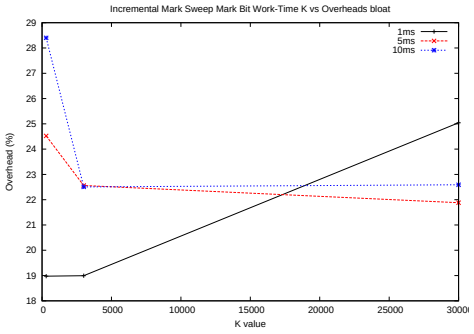
(h) xalan average pause times

Figure B.6: Graphed results for average pause times

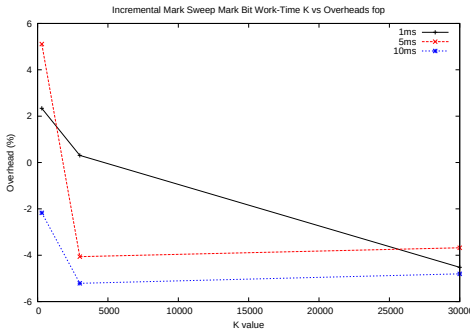
B.1.3 Incremental Mark-Sweep time-work



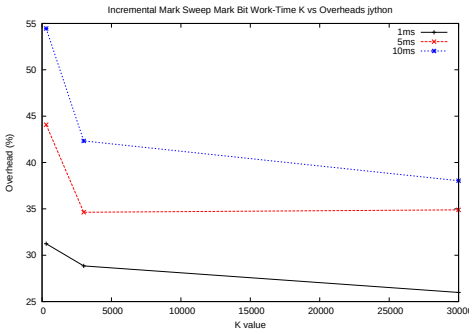
(a) antlr overhead work-time approach



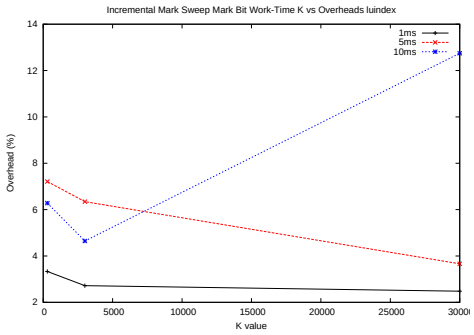
(b) bloat overhead work-time approach



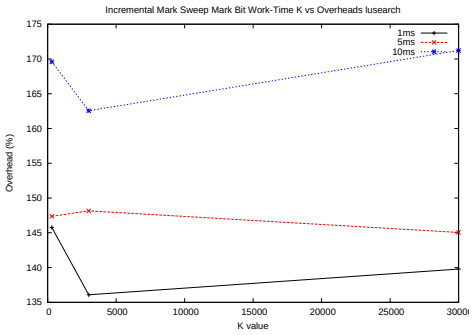
(c) fop overhead work-time approach



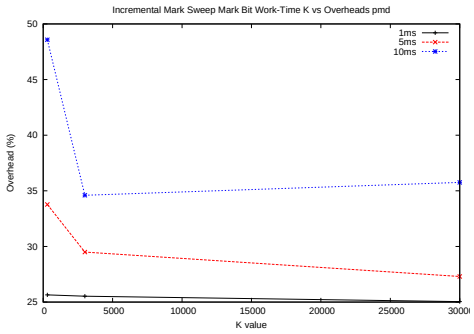
(d) jython overhead work-time approach



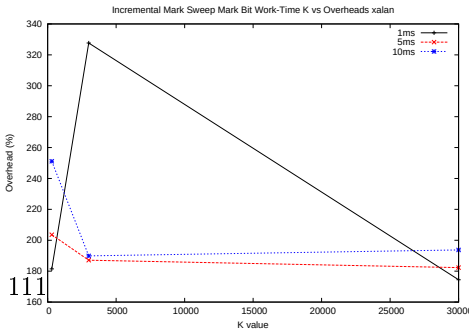
(e) luindex overhead work-time approach



(f) lusearch overhead work-time approach

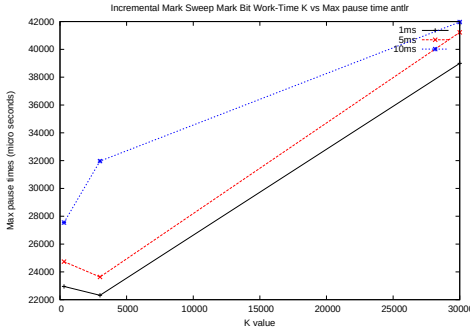


(g) pmd overhead work-time approach

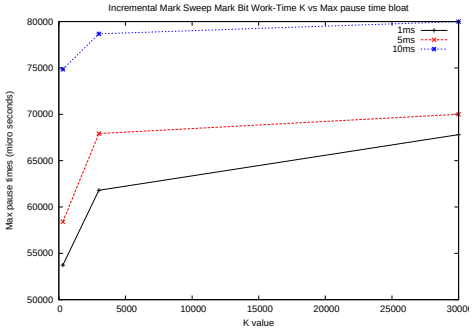


(h) xalan overhead work-time approach

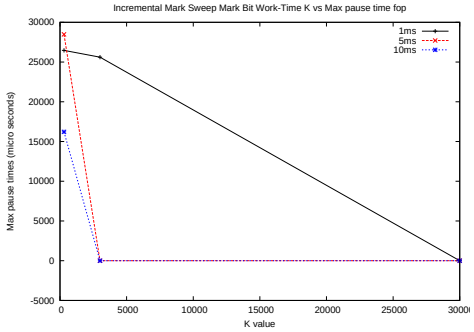
Figure B.7: Graphed results for the overhead of the time-work based approach



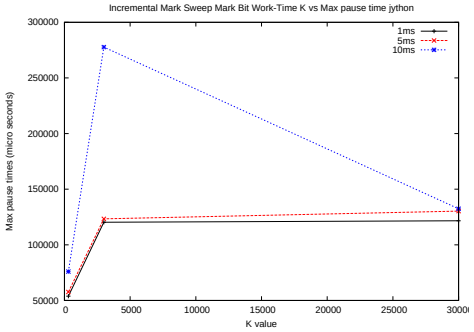
(a) antlr maximum pause times



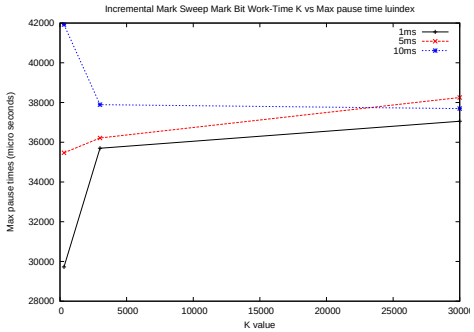
(b) bloat maximum pause times



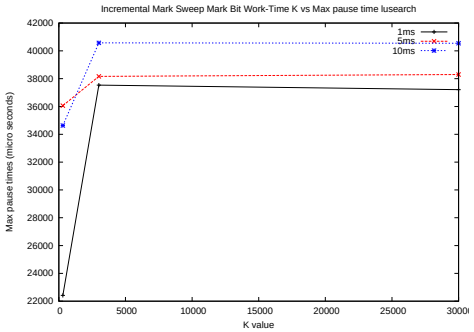
(c) fop maximum pause times



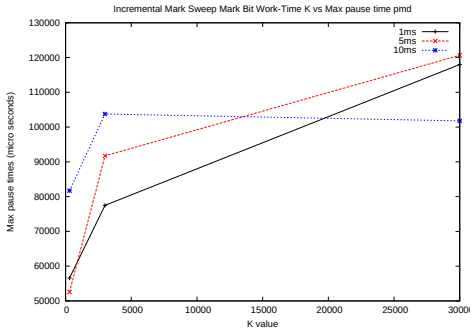
(d) jython maximum pause times



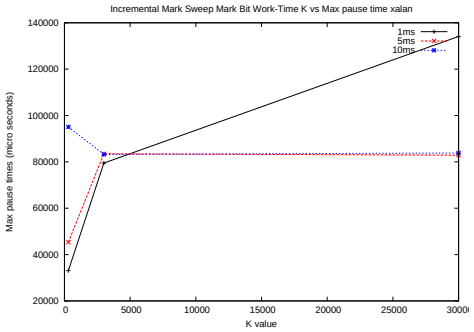
(e) luindex maximum pause times



(f) lusearch maximum pause times

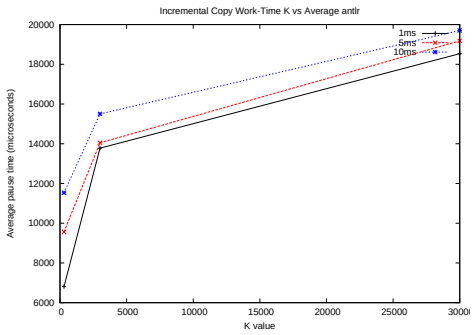


(g) pmd maximum pause times

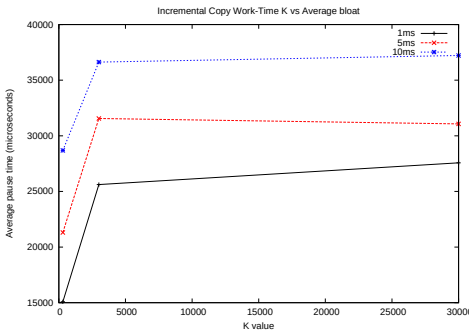


(h) xalan maximum pause times

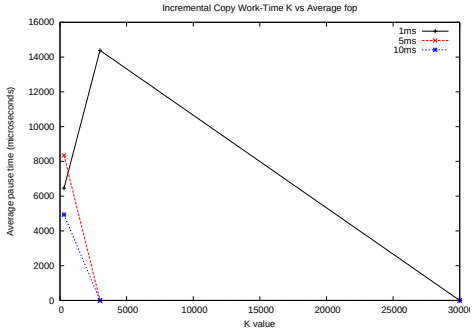
Figure B.8: Graphed results for maximum pause time of the time-work based approach



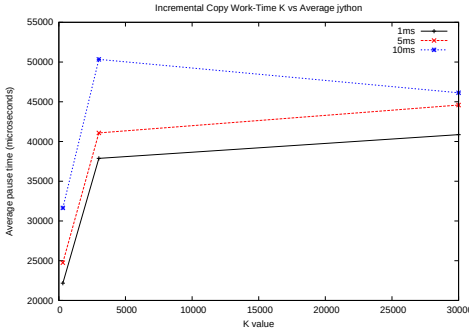
(a) antlr maximum pause times



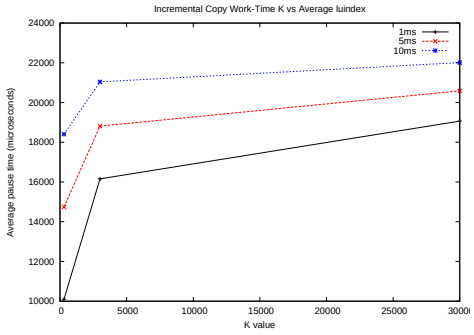
(b) bloat maximum pause times



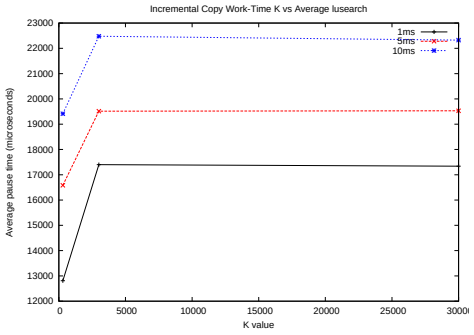
(c) fop maximum pause times



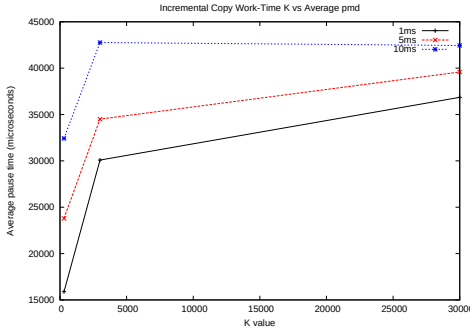
(d) jython maximum pause times



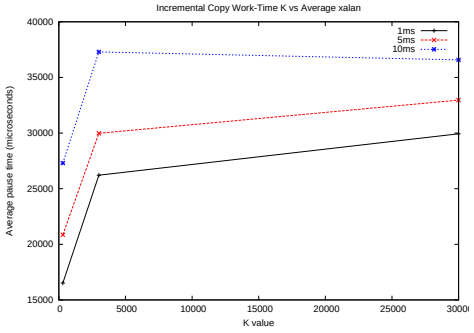
(e) luindex maximum pause times



(f) lusearch maximum pause times



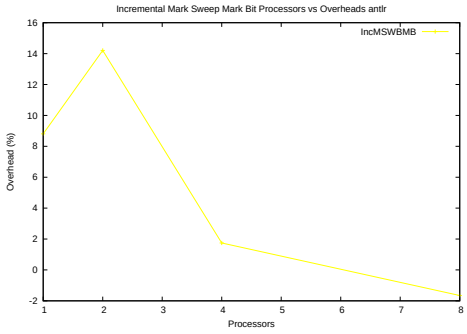
(g) pmd maximum pause times



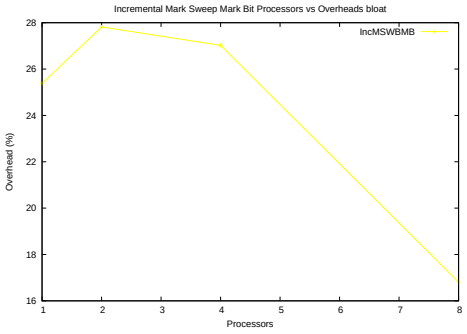
(h) xalan maximum pause times

Figure B.9: Graphed results for average pause time of the time-work based approach

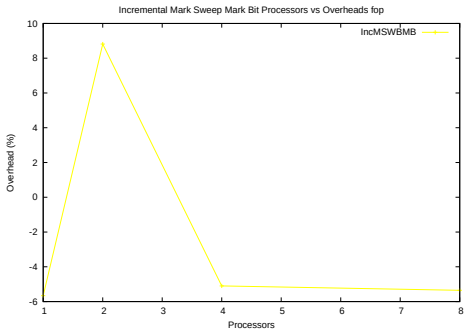
B.1.4 Parallel Incremental Mark-Sweep



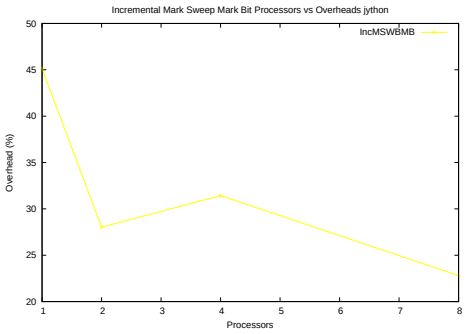
(a) antlr overheads for different p's



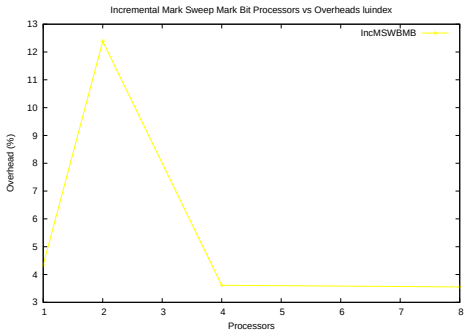
(b) bloat overheads for different p's



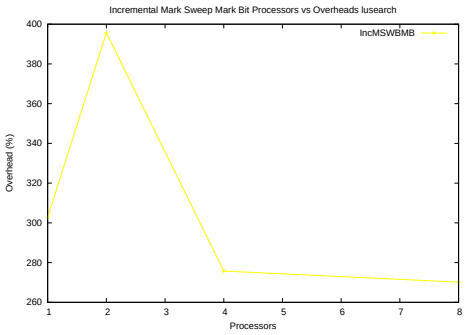
(c) fop overheads for different p's



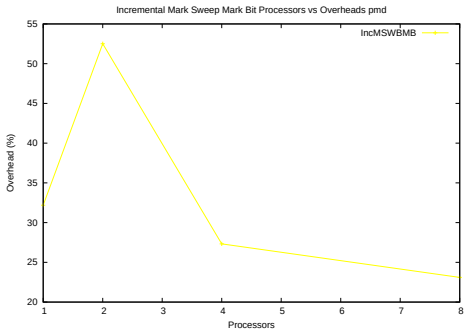
(d) jython overheads for different p's



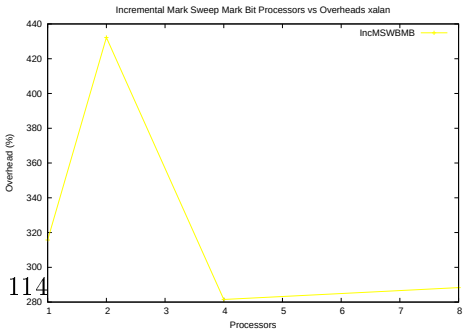
(e) luindex overheads for different p's



(f) lusearch overheads for different p's

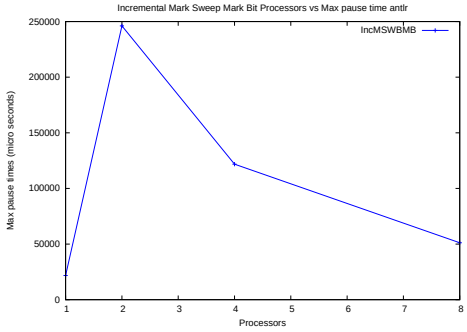


(g) pmd overheads for different p's

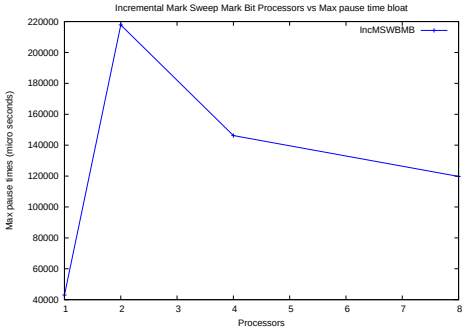


(h) xalan overheads for different p's

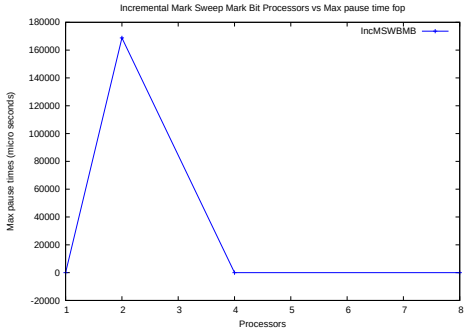
Figure B.10: Graphed total overhead for varying core work based k = 3000



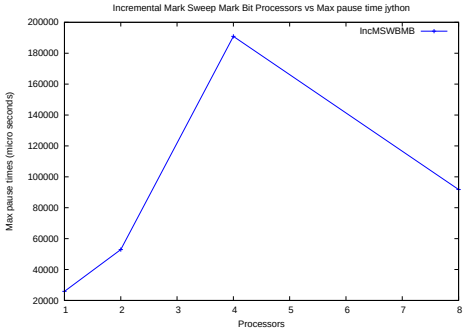
(a) antlr max pause times for different p's



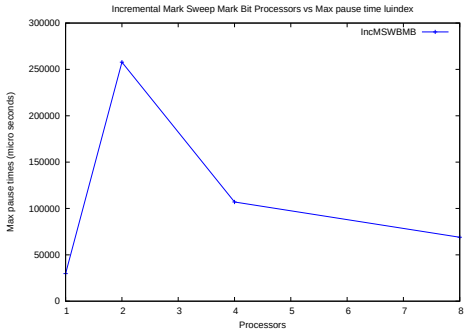
(b) bloat max pause times for different p's



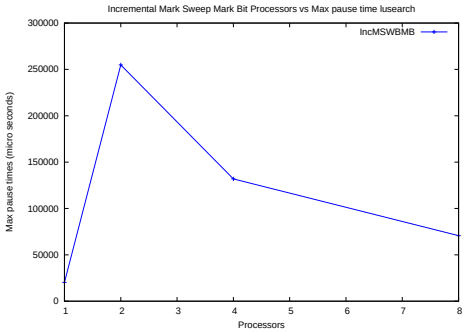
(c) fop max pause times for different p's



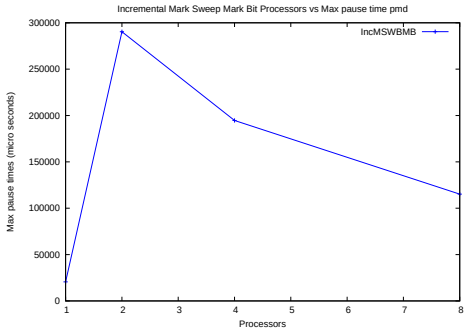
(d) jython max pause times for different p's



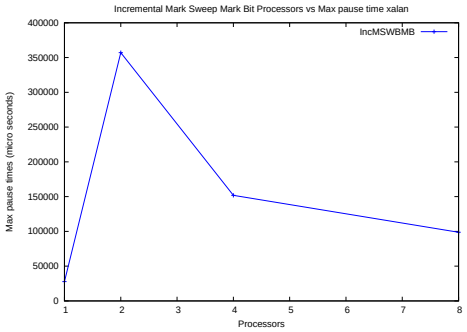
(e) luindex max pause times for different p's



(f) lusearch max pause times for different p's

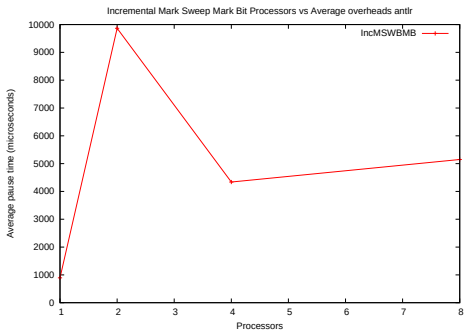


(g) pmd max pause times for different p's

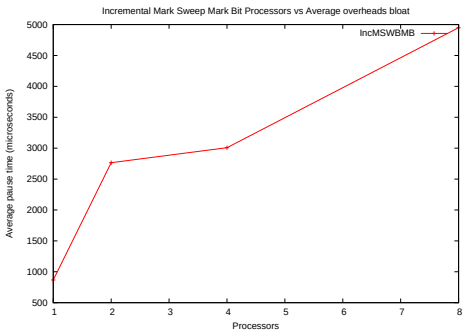


(h) xalan max pause times for different p's

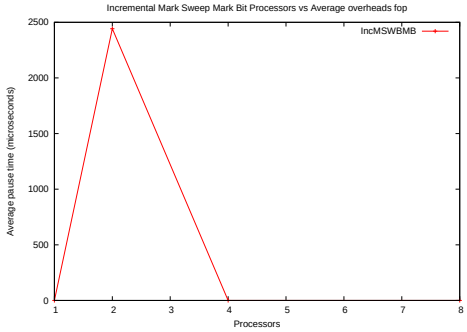
Figure B.11: Graphed max pause times for varying core work based k = 3000



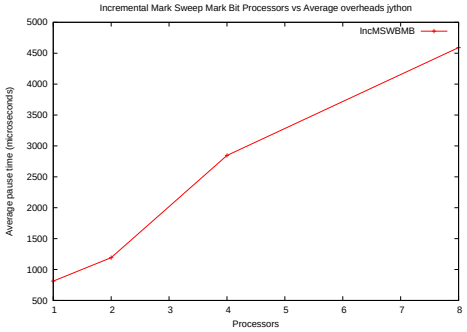
(a) antlr average pause times for different p's



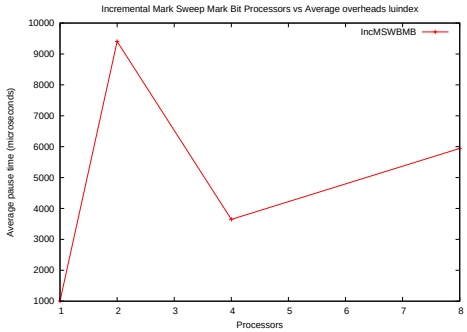
(b) bloat average pause times for different p's



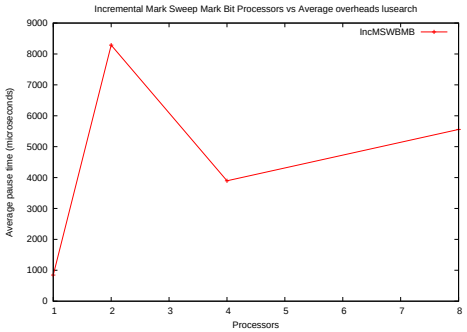
(c) fop average pause times for different p's



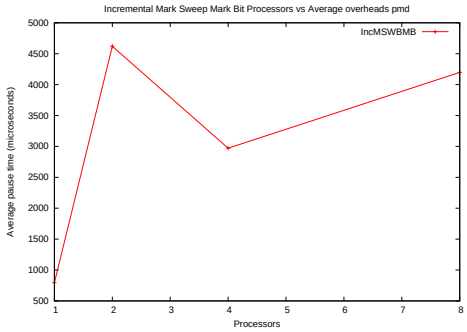
(d) jython average pause times for different p's



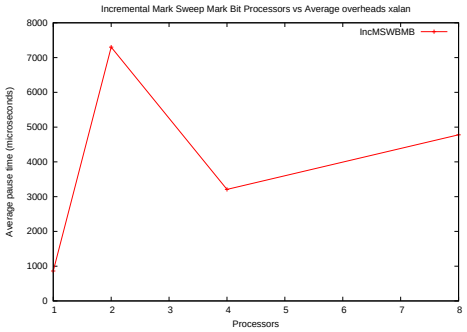
(e) luindex average pause times for different p's



(f) lusearch average pause times for different p's



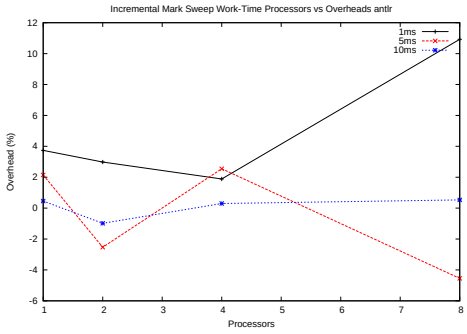
(g) pmd average pause times for different p's



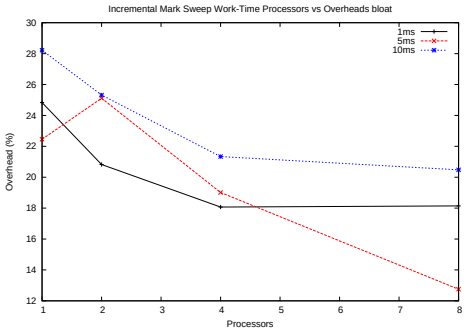
(h) xalan average pause times for different p's

Figure B.12: Graphed avgerage pause times for varying core work based k = 3000

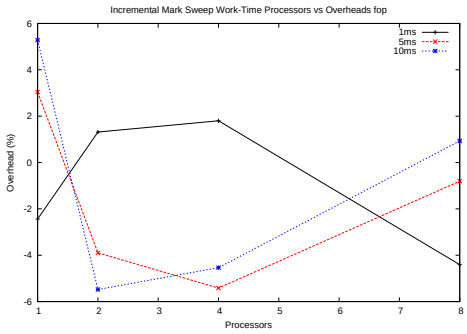
B.1.5 Parallel Incremental Mark-Sweep Work Time



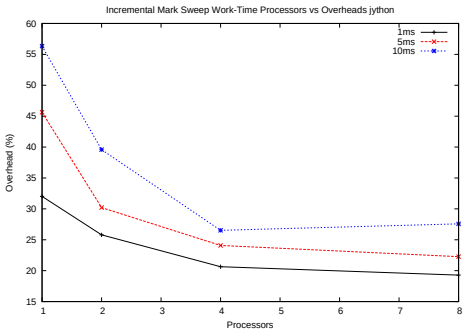
(a) antlr overheads for different p's



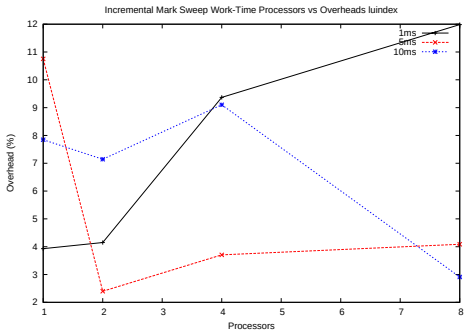
(b) bloat overheads for different p's



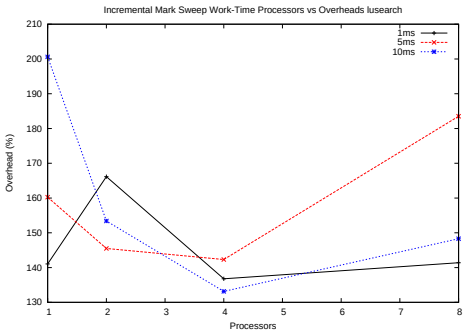
(c) fop overheads for different p's



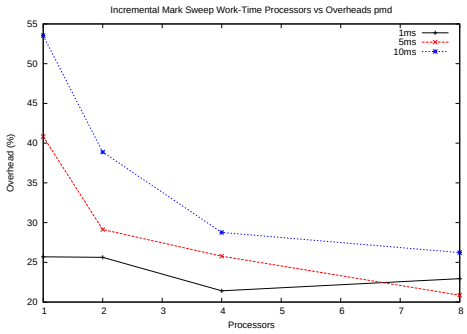
(d) jython overheads for different p's



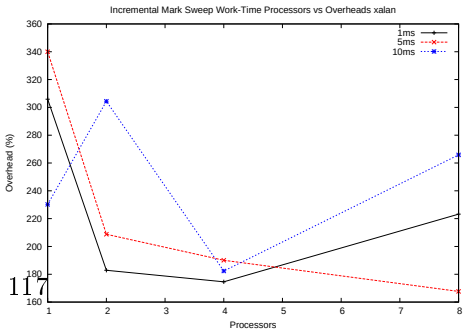
(e) luindex overheads for different p's



(f) lusearch overheads for different p's

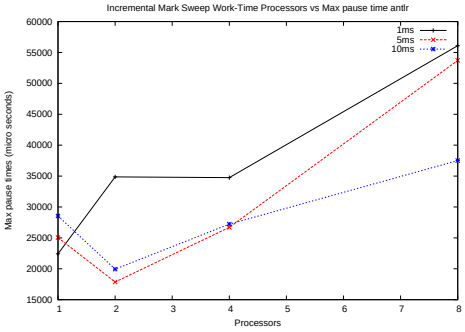


(g) pmd overheads for different p's

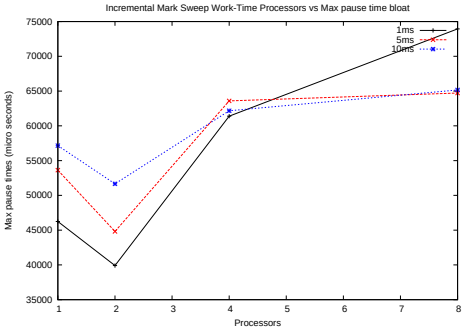


(h) xalan overheads for different p's

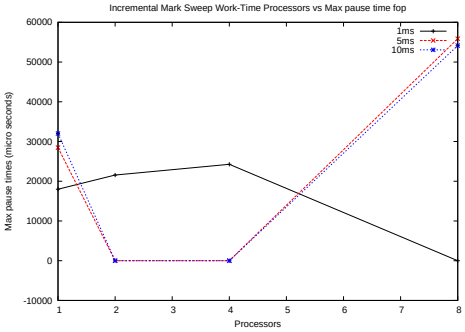
Figure B.13: Graphed total overhead for varying core work based k = 3000



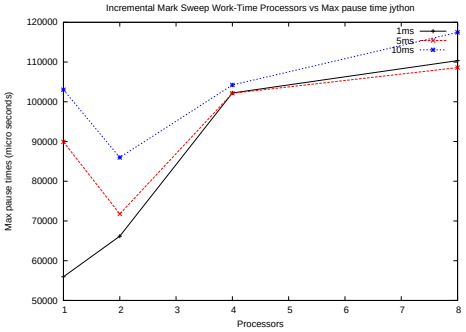
(a) antlr max pause times for different p's



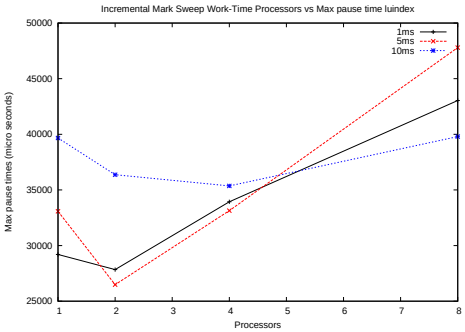
(b) bloat max pause times for different p's



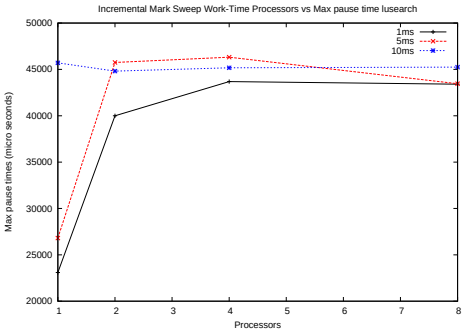
(c) fop max pause times for different p's



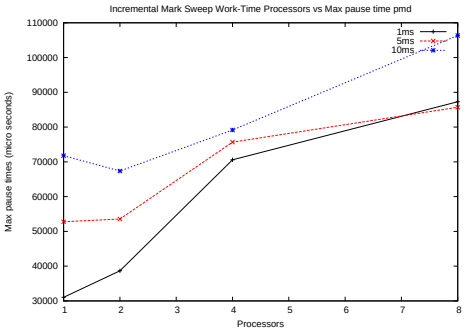
(d) jython max pause times for different p's



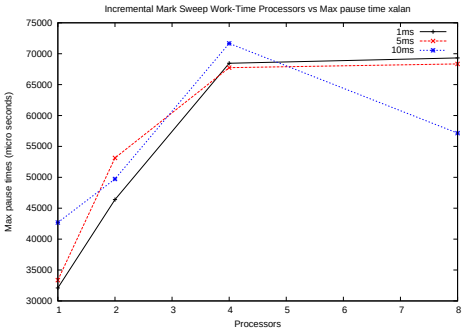
(e) luindex max pause times for different p's



(f) lusearch max pause times for different p's

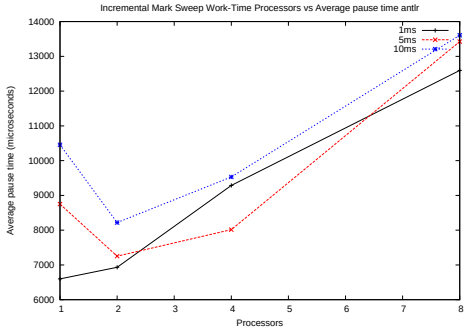


(g) pmd max pause times for different p's

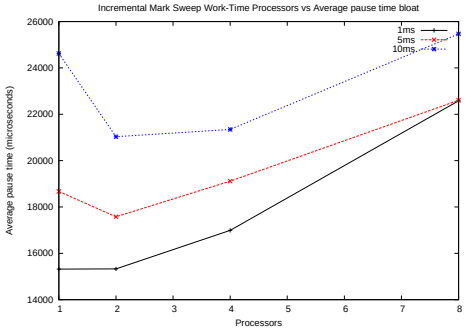


(h) xalan max pause times for different p's

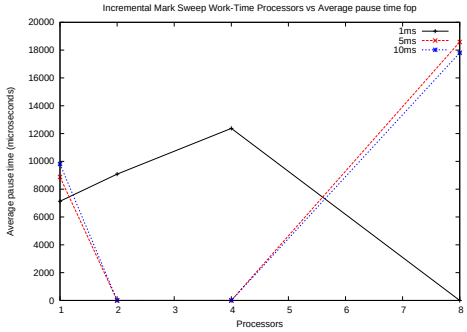
Figure B.14: Graphed max pause time for varying core's k=3000



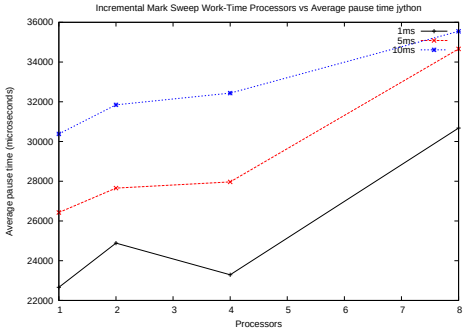
(a) antlr average pause times for different p's



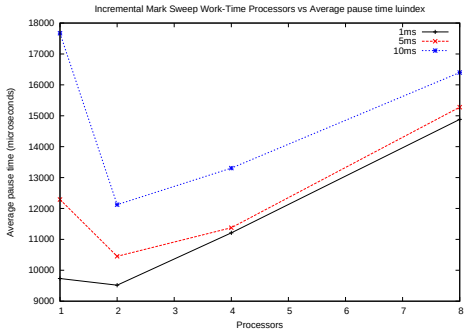
(b) bloat average pause times for different p's



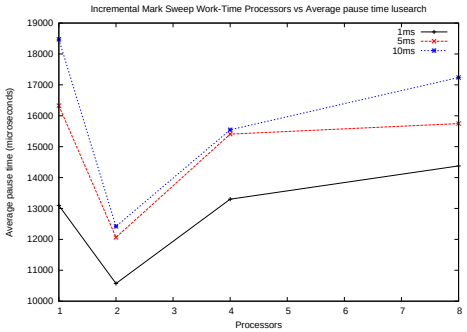
(c) fop average pause times for different p's



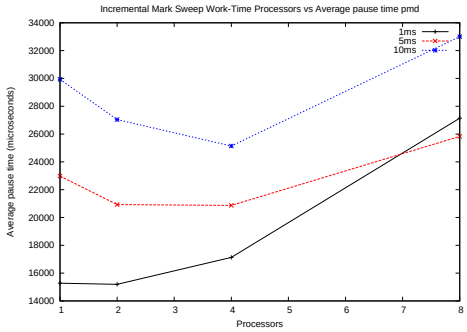
(d) jython average pause times for different p's



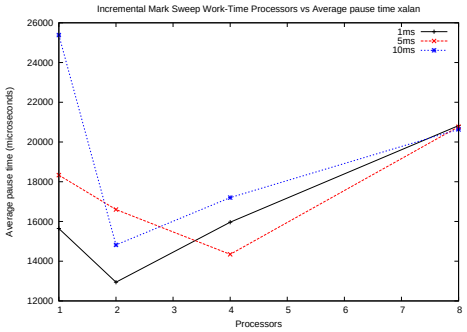
(e) luindex average pause times for different p's



(f) lusearch average pause times for different p's



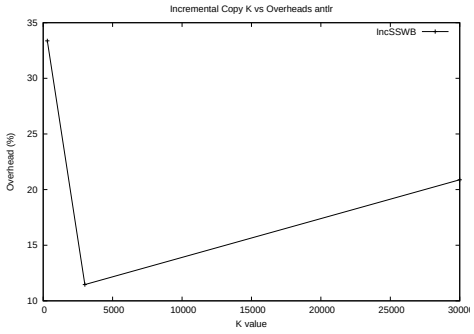
(g) pmd average pause times for different p's



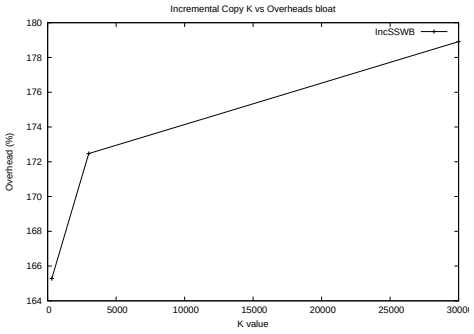
(h) xalan average pause times for different p's

Figure B.15: Graphed avgerage pause times for varying core work based k = 3000

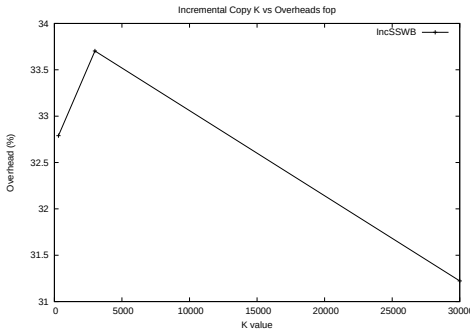
B.1.6 Incremental Copying Collector



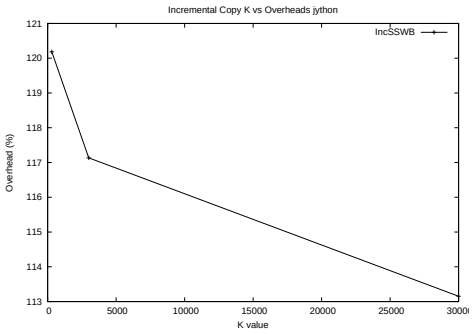
(a) antlr overheads for different p's



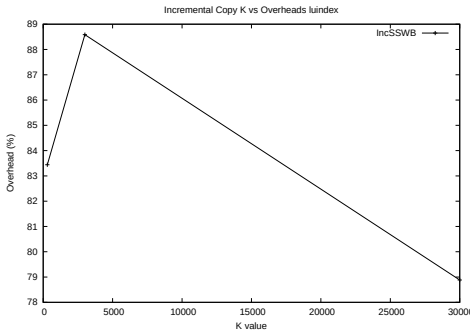
(b) bloat overheads for different p's



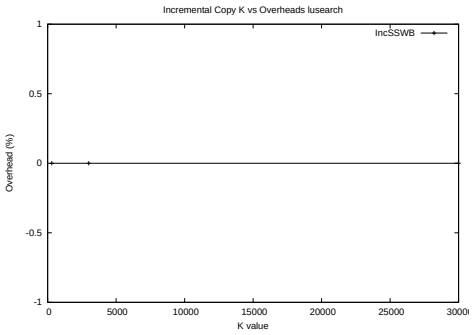
(c) fop overheads for different p's



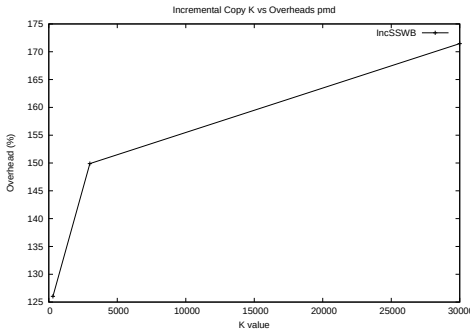
(d) jython overheads for different p's



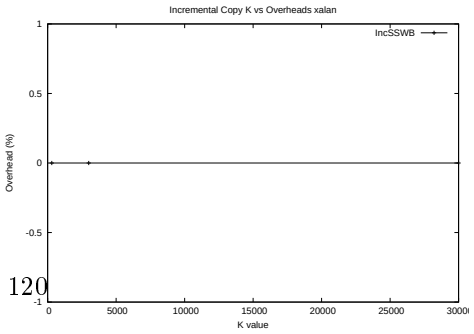
(e) luindex overheads for different p's



(f) lusearch overheads for different p's

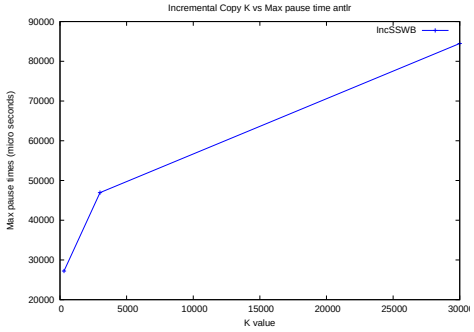


(g) pmd overheads for different p's

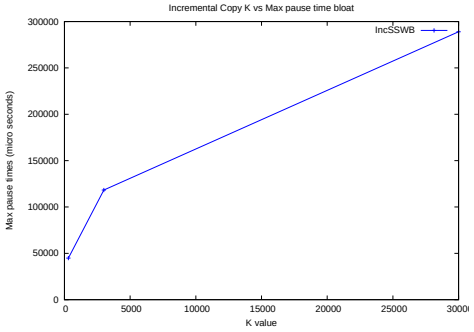


(h) xalan overheads for different p's

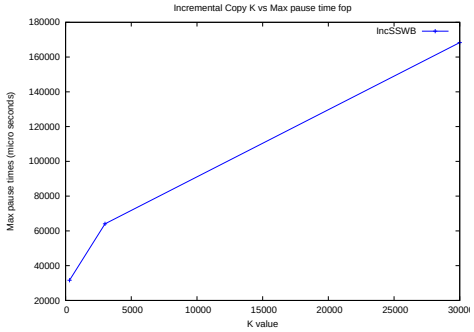
Figure B.16: Graphed total overhead for varying core work based k = 3000



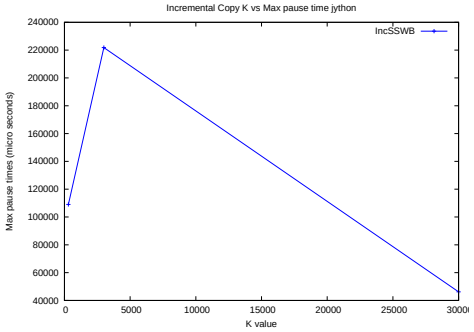
(a) antlr max pause times for different p's



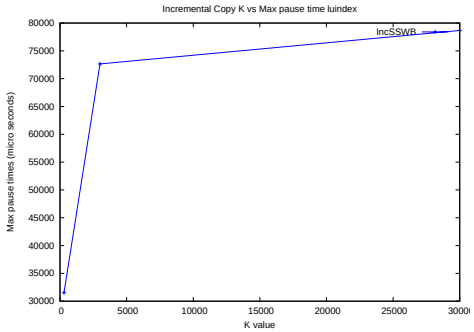
(b) bloat max pause times for different p's



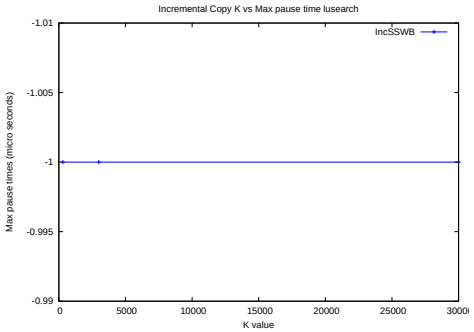
(c) fop max pause times for different p's



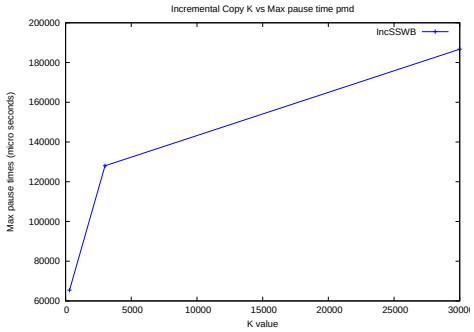
(d) jython max pause times for different p's



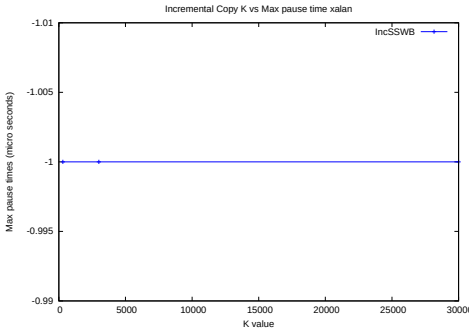
(e) luindex max pause times for different p's



(f) lusearch max pause times for different p's

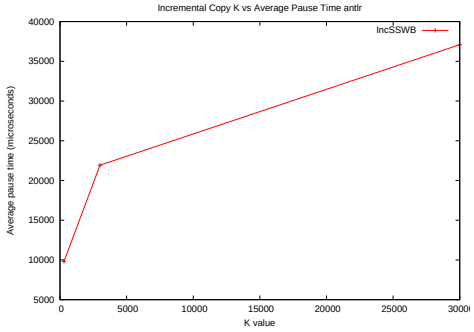


(g) pmd max pause times for different p's

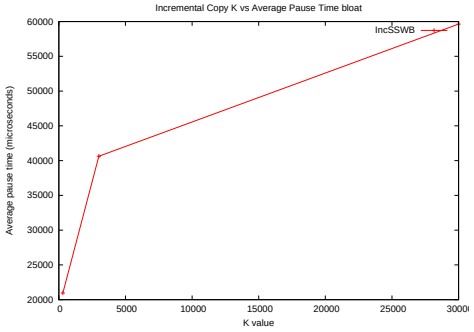


(h) xalan max pause times for different p's

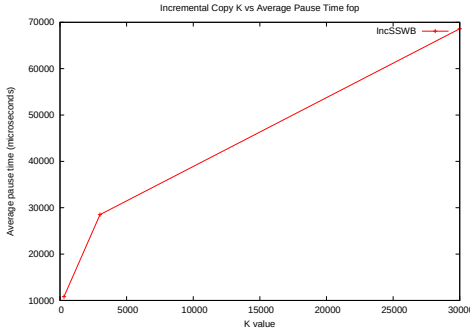
Figure B.17: Graphed max pause times for varying core work based k = 3000



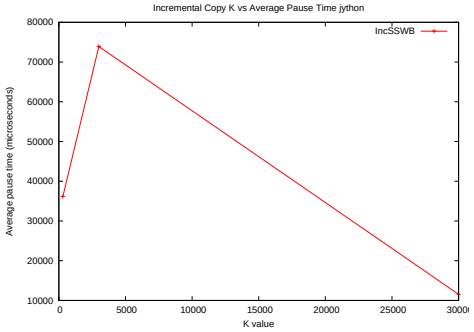
(a) antlr average pause times for different p's



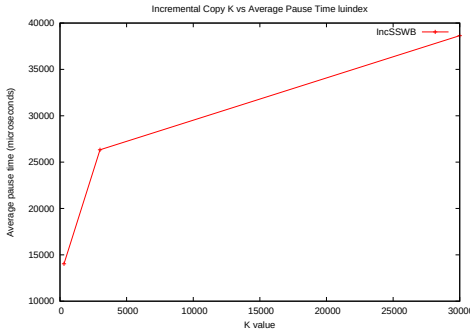
(b) bloat average pause times for different p's



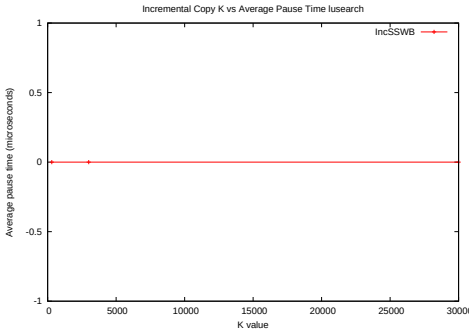
(c) fop average pause times for different p's



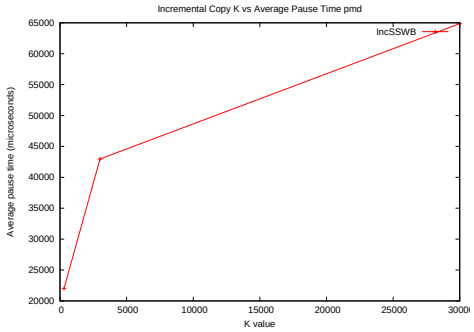
(d) jython average pause times for different p's



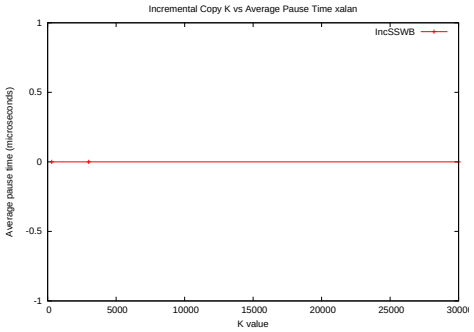
(e) luindex average pause times for different p's



(f) lusearch average pause times for different p's



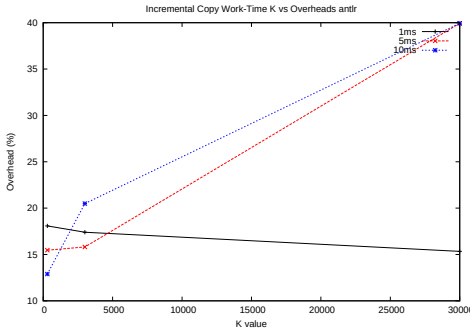
(g) pmd average pause times for different p's



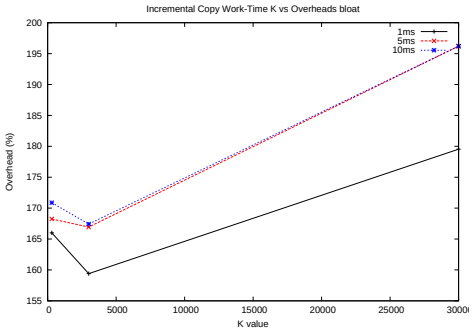
(h) xalan average pause times for different p's

Figure B.18: Graphed avgerage pause times for varying core work based k = 3000

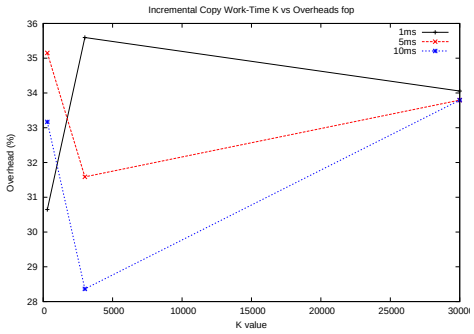
B.1.7 Incremental Copying Collector work-time



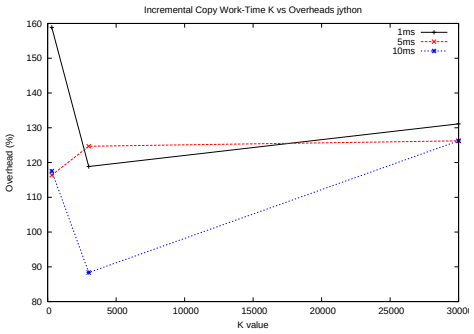
(a) antlr overheads for different p's



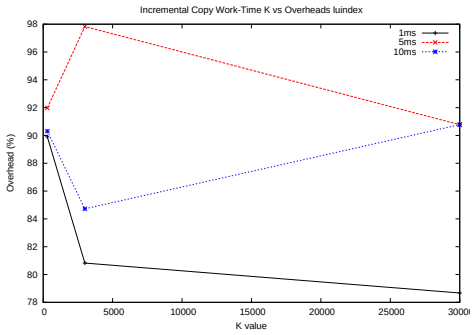
(b) bloat overheads for different p's



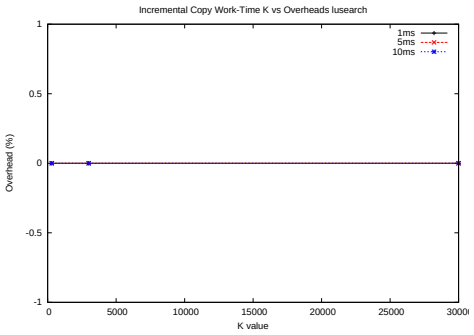
(c) fop overheads for different p's



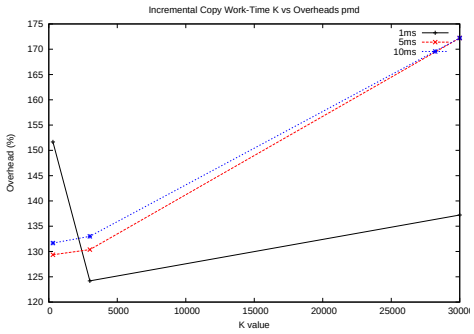
(d) jython overheads for different p's



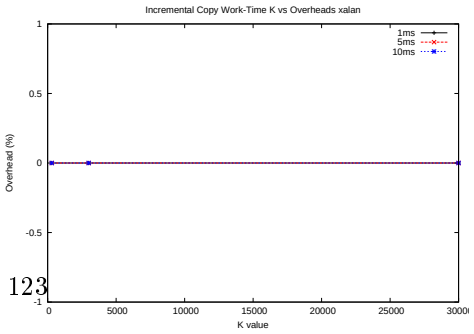
(e) luindex overheads for different p's



(f) lusearch overheads for different p's

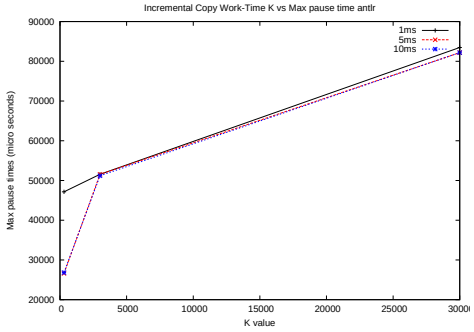


(g) pmd overheads for different p's

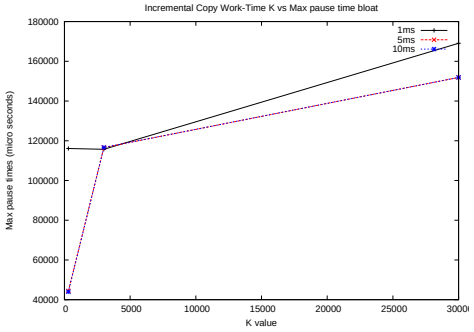


(h) xalan overheads for different p's

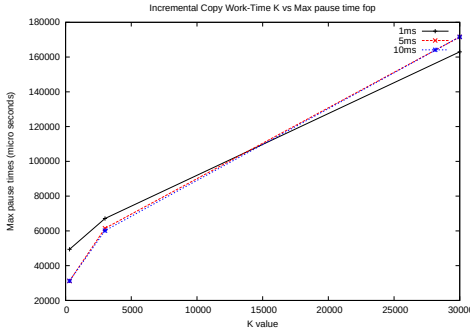
Figure B.19: Graphed total overhead for varying core work based k = 3000



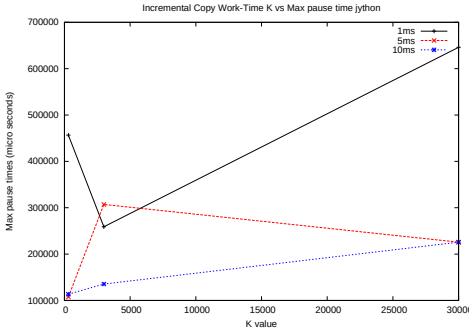
(a) antlr max pause times for different p's



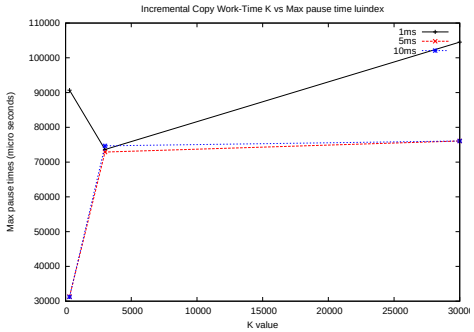
(b) bloat max pause times for different p's



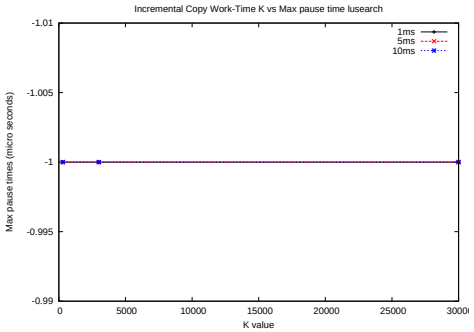
(c) fop max pause times for different p's



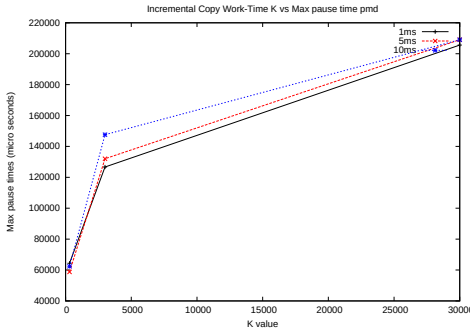
(d) jython max pause times for different p's



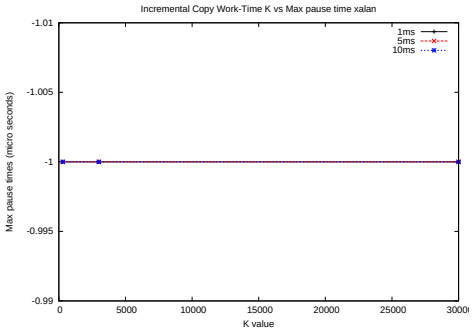
(e) luindex max pause times for different p's



(f) lusearch max pause times for different p's

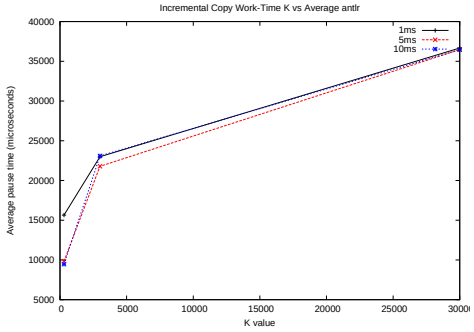


(g) pmd max pause times for different p's

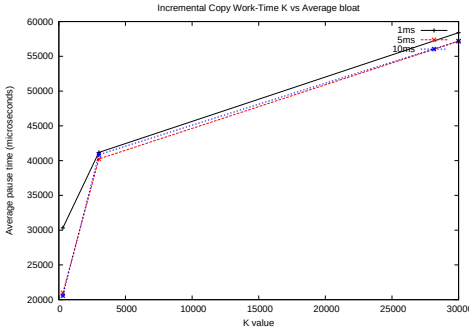


(h) xalan max pause times for different p's

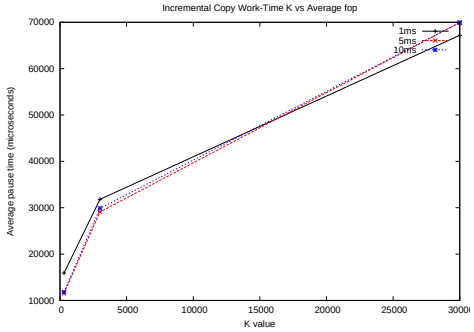
Figure B.20: Graphed max pause times for varying core work based k = 3000



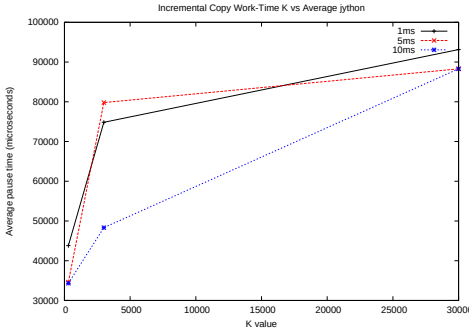
(a) antlr average pause times for different p's



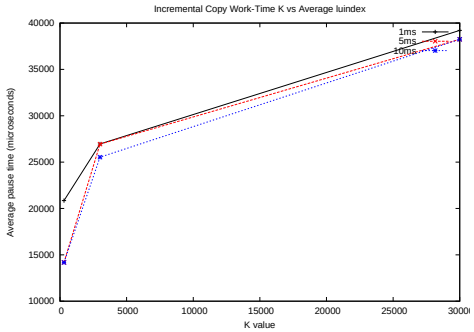
(b) bloat average pause times for different p's



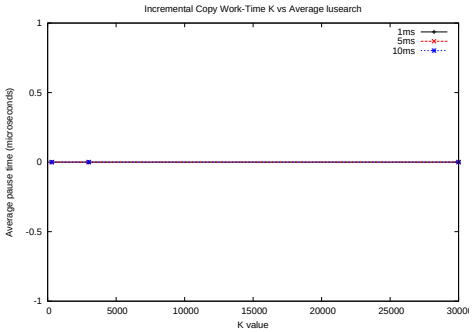
(c) fop average pause times for different p's



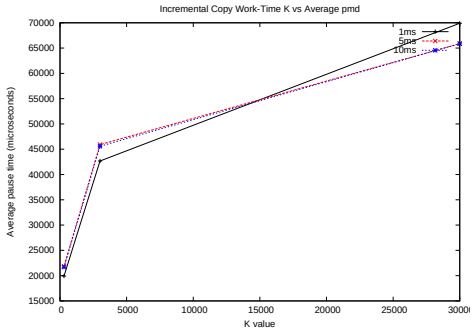
(d) jython average pause times for different p's



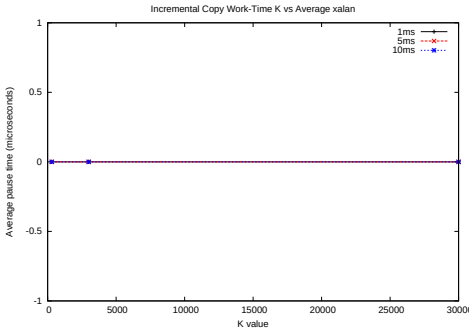
(e) luindex average pause times for different p's



(f) lusearch average pause times for different p's



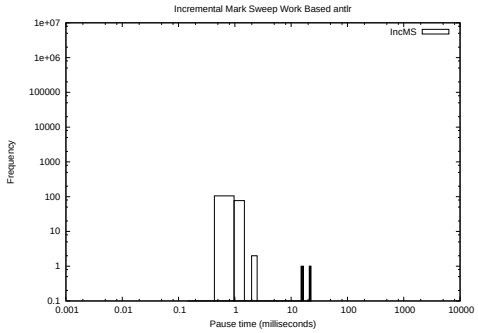
(g) pmd average pause times for different p's



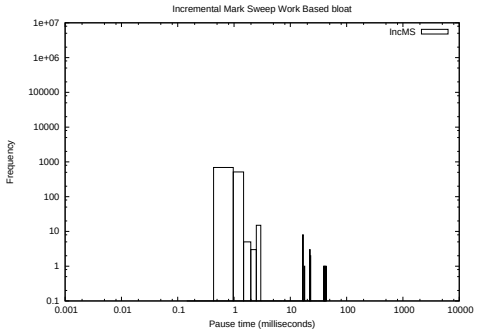
(h) xalan average pause times for different p's

Figure B.21: Graphed avgerage pause times for varying core work based k = 3000

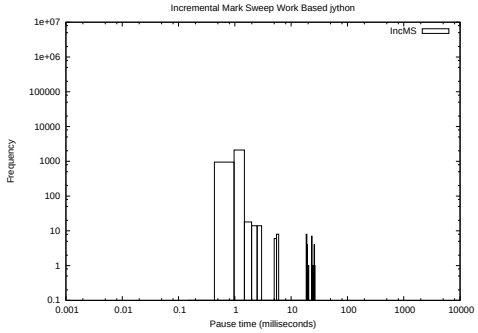
B.2 MMU, Average CPU utilistaion and Pause Time Distribution



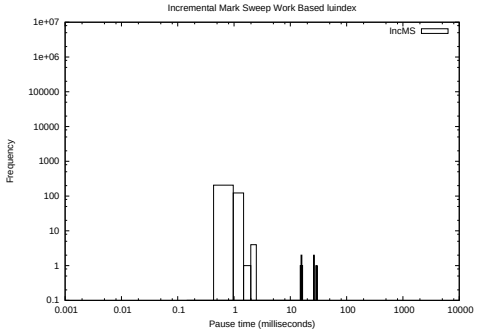
(a) antlr pause time distribution



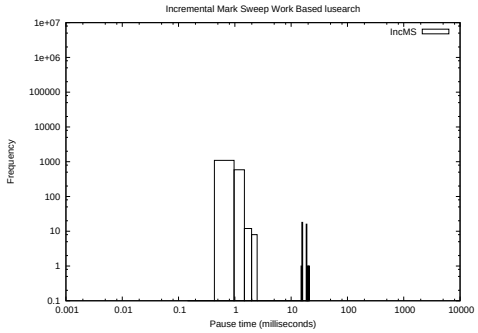
(b) bloat pause time distribution



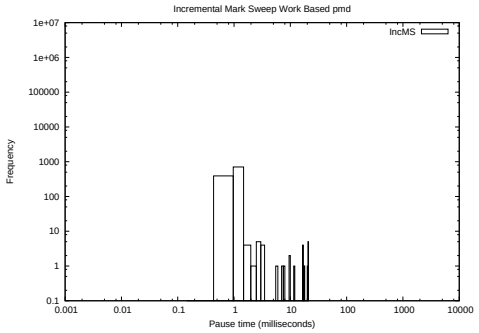
(c) jython pause time distribution



(d) luindex pause time distribution

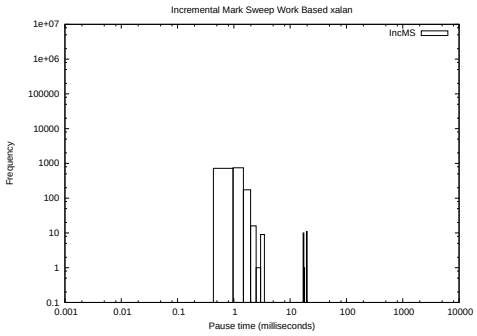


(e) lusearch pause time distribution



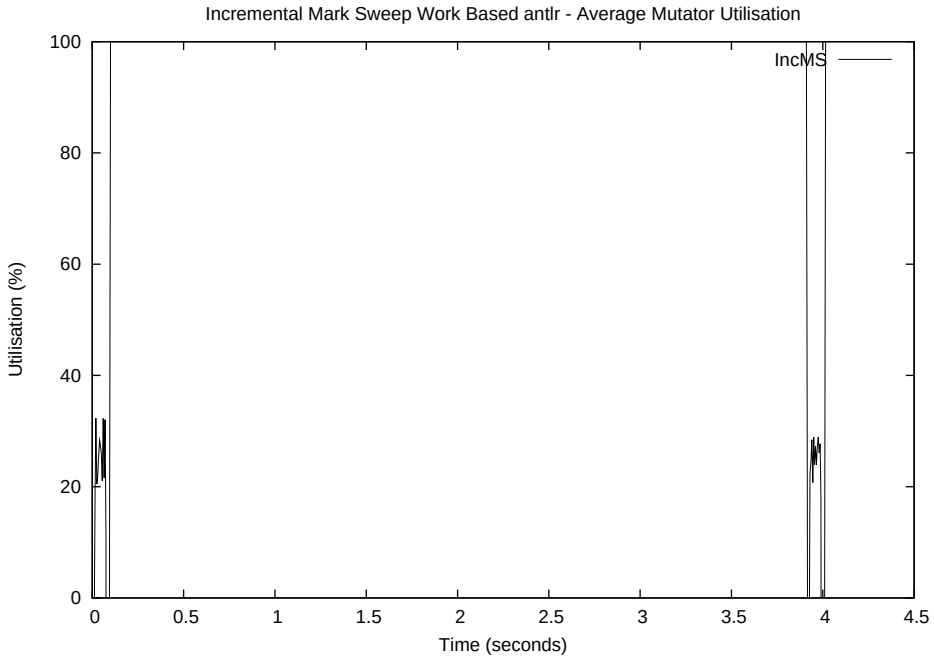
(f) pmd pause time distribution

Figure B.22: Pause time distibution figure for IncMS Work based k = 3000

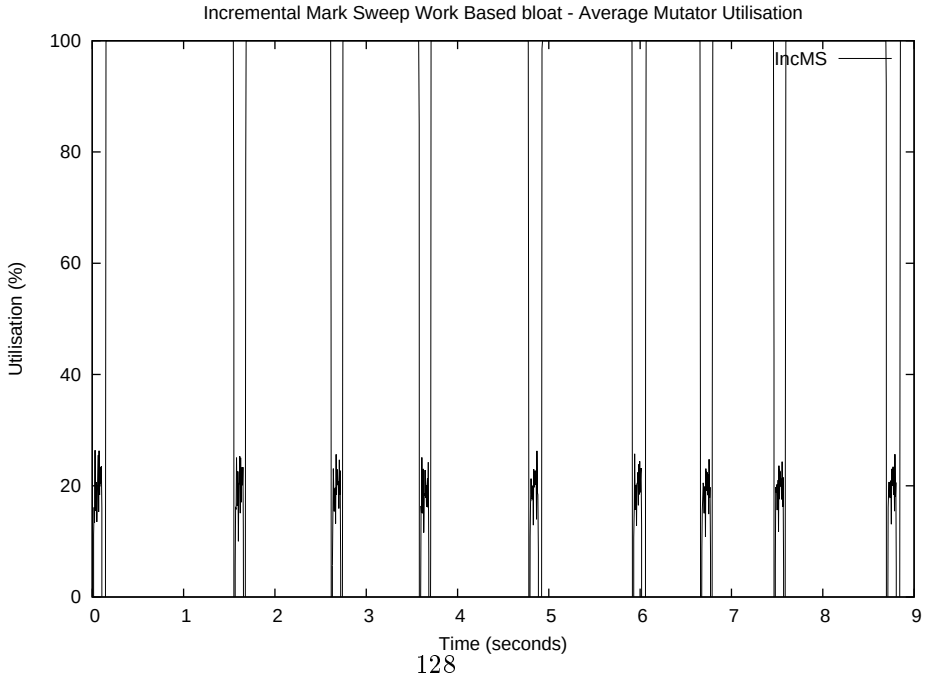


(a) xalan pause time distribution

Figure B.23: Pause time distribution figures for IncMS Work based k = 3000 continued

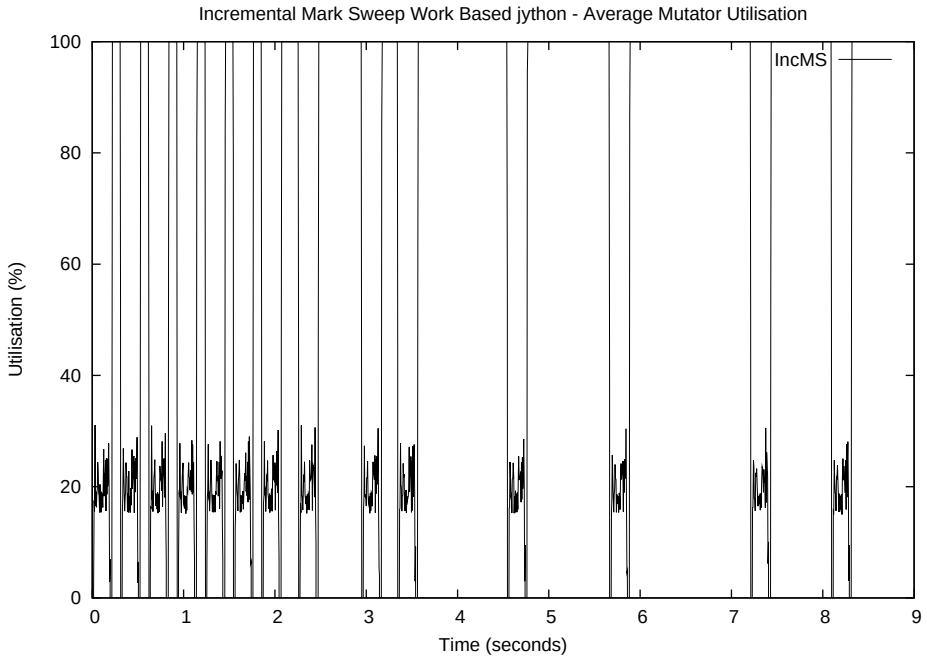


(a) antlr Average CPU utilisation

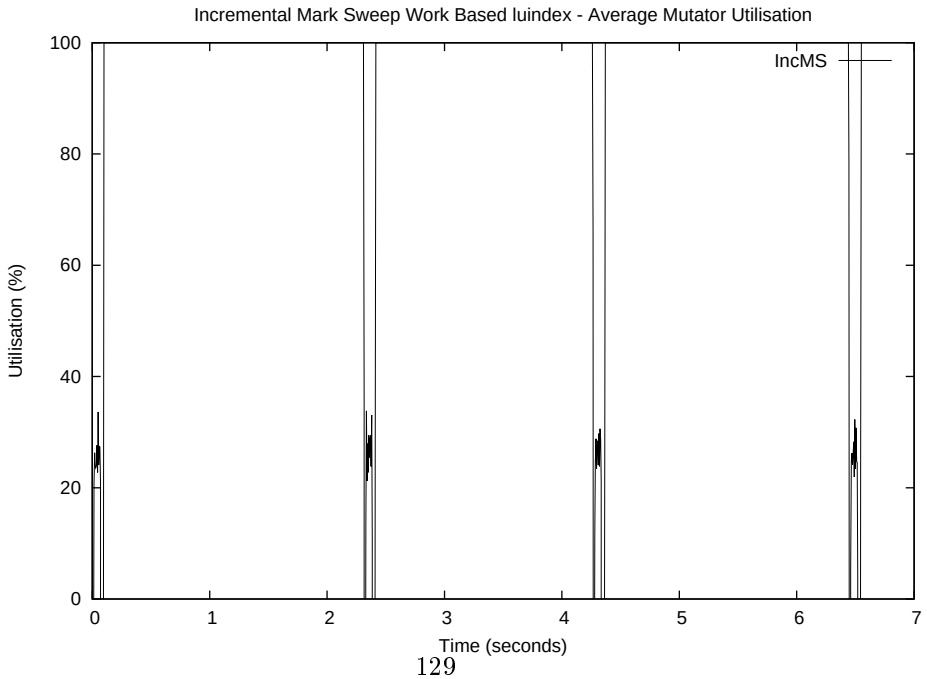


(b) bloat Average CPU utilisation

Figure B.24: Average CPU utilisation for IncMS Work based k = 3000

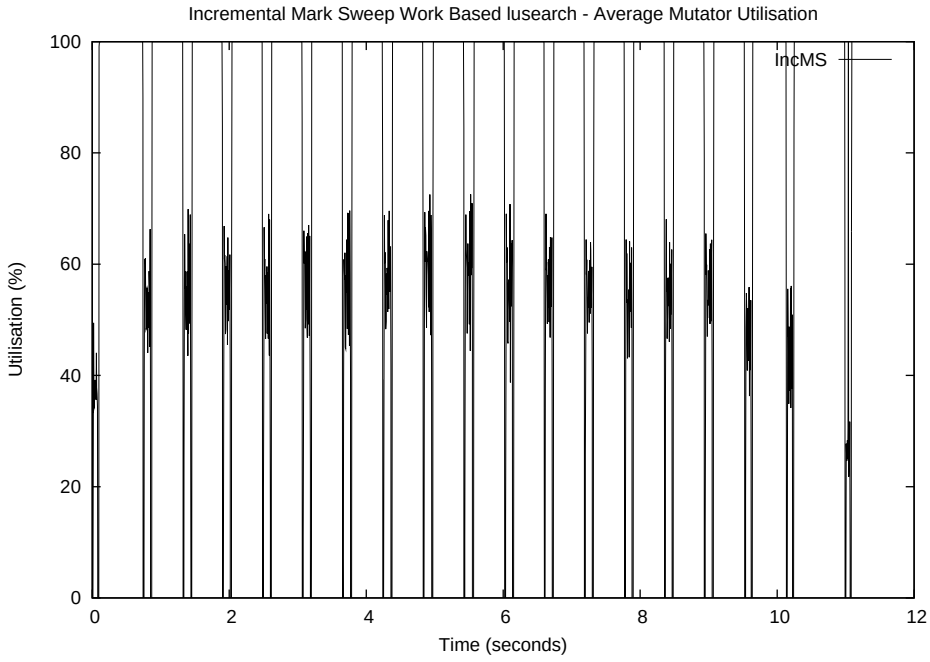


(a) jython Average CPU utilisation

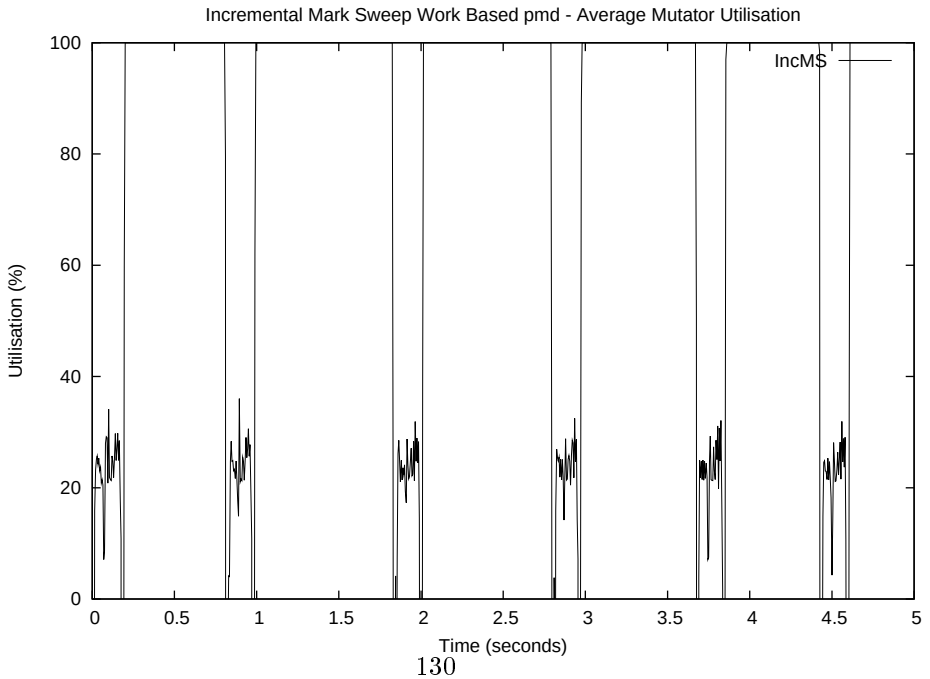


(b) luindex Average CPU utilisation

Figure B.25: Average CPU utilisation for IncMS Work based k = 3000 (Continued)

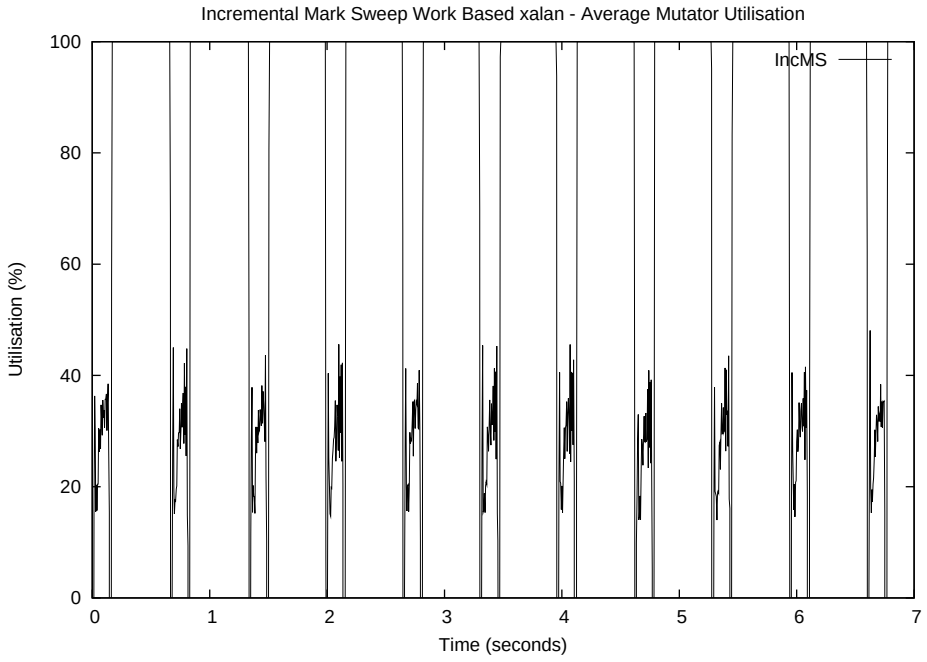


(a) lusearch Average CPU utilisation



(b) pmd Average CPU utilisation

Figure B.26: Average CPU utilisation for IncMS Work based k = 3000 (Continued)



(a) xalan Average CPU utilisation

Figure B.27: Average CPU utilisation for IncMS Work based k = 3000 (Continued)

Bibliography

[1] Gdb homepage. URL `http://www.gnu.org/software/gdb/`.

[2] The java language environment. URL `http://java.sun.com/docs/white/langenv/Intro.doc2.html`.

[3] Specjvm98 homepage. URL `http://www.spec.org/jvm98/`.

[4] Valgrind homepage. URL `http://www.valgrind.org/`.

[5] Bowen Alpern, C. R. Attanasio, Anthony Cocchi, Derek Lieber, Stephen Smith, Ton Ngo, John J. Barton, Susan Flynn Hummel, Janice C. Sheperd, and Mark Mergen. Implementing jalape no in java. *SIGPLAN Not.*, 34(10):314–324, 1999.

[6] A. W. Appel. Simple generational garbage collection and fast allocation. *Softw. Pract. Exper.*, 19(2):171–183, 1989.

[7] Matthew Arnold and Barbara G. Ryder. A framework for reducing the cost of instrumented code. *SIGPLAN Not.*, 36(5):168–179, 2001.

[8] Hezi Azatchi, Yossi Levanoni, Harel Paz, and Erez Petrank. An on-the-fly mark and sweep garbage collector based on sliding views. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 269–281, New York, NY, USA, 2003. ACM.

[9] David F. Bacon, Perry Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of the 30th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 285–298, New Orleans, Louisiana, January 2003.

[10] David F. Bacon, Perry Cheng, and V.T. Rajan. The metronome: A simpler approach to garbage collection in real-time systems. In *In Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES), OTM Workshops*, pages 466–478, 2003.

[11] Henry G. Baker, Jr. List processing in real time on a serial computer. *Commun. ACM*, 21(4):280–294, 1978.

[12] David A. Barrett and Benjamin G. Zorn. Using lifetime predictors to improve memory allocation performance. *SIGPLAN Not.*, 28(6):187–196, 1993.

[13] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage,

and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, New York, NY, USA, October 2006. ACM Press.

[14] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. Oil and water? high performance garbage collection in java with mmtk. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 137–146, Washington, DC, USA, 2004. IEEE Computer Society.

[15] Stephen M. Blackburn and Antony L. Hosking. Barriers: friend or foe? In *ISMM '04: Proceedings of the 4th international symposium on Memory management*, pages 143–151, New York, NY, USA, 2004. ACM.

[16] Stephen M. Blackburn and Kathryn S. McKinley. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 22–32, New York, NY, USA, 2008. ACM.

[17] Rodney A. Brooks. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 256–262, New York, NY, USA, 1984. ACM.

[18] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of self a dynamically-typed object-oriented language based on prototypes. *SIGPLAN Not.*, 24(10):49–70, 1989.

[19] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of self a dynamically-typed object-oriented language based on prototypes. *SIGPLAN Not.*, 24(10):49–70, 1989.

[20] C. J. Cheney. A nonrecursive list compacting algorithm. *Commun. ACM*, 13(11):677–678, 1970.

[21] Perry Cheng. *Scalable real-time parallel garbage collection for symmetric multiprocessors.* PhD thesis, Pittsburgh, PA, USA, 2001. Adviser-Blelloch, Guy and Adviser-Harper, Robert.

[22] Perry Cheng and Guy E. Blelloch. A parallel, real-time garbage collector. *SIGPLAN Not.*, 36(5):125–136, 2001.

[23] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: an exercise in cooperation. *Commun. ACM*, 21(11):966–975, 1978.

[24] Damien Doligez and Georges Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 70–83, New York, NY, USA, 1994. ACM.

[25] Damien Doligez and Xavier Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ml. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 113–123, New York, NY, USA, 1993. ACM.

[26] Tamar Domani, Elliot K. Kolodner, Ethan Lewis, Eliot E. Salant, Katherine Barabash, Itai Lahan, Yossi Levanoni, Erez Petrank, and Igor Yanorer. Implementing an on-the-fly garbage collector for java. *SIGPLAN Not.*, 36(1):155–166, 2001.

[27] Tamar Domani, Elliot K. Kolodner, and Erez Petrank. A generational on-the-fly garbage collector for java. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 274–284, New York, NY, USA, 2000. ACM.

[28] David A. Fisher. Bounded workspace garbage collection in an address order preserving list processing environment. *Information Processing Letters*, 3(1):25–32, July 1974.

[29] Daniel Frampton, Stephen M. Blackburn, Perry Cheng, Robin J. Garner, David Grove, J. Eliot B. Moss, and Sergey I. Salishev. Demystifying magic: high-level low-level programming. In *VEE '09: Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 81–90, New York, NY, USA, 2009. ACM.

[30] Barry Hayes. Using key object opportunism to collect old objects. *SIGPLAN Not.*, 26(11):33–46, 1991.

[31] Antony Hosking, J. Eliot B. Moss, J. Eliot, and B. Moss. Protection traps and alternatives for memory management of an object-oriented language., 1993.

[32] Antony L. Hosking and Richard L. Hudson. Remembered sets can also play cards. In *In OOPSLA'93 Workshop on Garbage Collection and Memory Management*, 1993.

[33] Antony L. Hosking, J. Eliot B. Moss, and Darko Stefanovic. A comparative performance evaluation of write barrier implementation. *SIGPLAN Not.*, 27(10):92–109, 1992.

[34] Richard L. Hudson and J. Eliot B. Moss. Sapphire: copying gc without stopping the world. In *JGI '01: Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, pages 48–57, New York, NY, USA, 2001. ACM.

[35] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. 1996.

[36] H. B. M. Jonkers. A fast garbage compaction algorithm. *Information Processing Letters*, 9(1):25–30, July 1979.

[37] Haim Kermany and Erez Petrank. The compressor: concurrent, incremental, and parallel compaction. *SIGPLAN Not.*, 41(6):354–363, 2006.

[38] Donald E. Knuth. *Lists and Garbage Collection*, volume I: Fundamental Algorithms, chapter 2, pages 408–423. Addison-Wesley, second edition, 1973.

[39] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, 1960.

[40] Matthias Meyer. A true hardware read barrier. In *ISMM '06: Proceedings of the 5th international symposium on Memory management*, pages 3–16, New York, NY, USA, 2006. ACM.

[41] Scott Nettles and James O'Toole. Real-time replication garbage collection. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 217–226, New York, NY, USA, 1993. ACM.

[42] Pekka P. Pirinen. Barrier techniques for incremental tracing. In *ISMM '98: Proceedings of the 1st international symposium on Memory management*, pages 20–25, New York, NY, USA, 1998. ACM.

[43] Filip Pizlo, Daniel Frampton, Erez Petrank, and Bjarne Steensgaard. Stopless: a real-time garbage collector for multiprocessors. In *ISMM '07: Proceedings of the 6th international symposium on Memory management*, pages 159–172, New York, NY, USA, 2007. ACM.

[44] Filip Pizlo, Erez Petrank, and Bjarne Steensgaard. A study of concurrent real-time garbage collectors. *SIGPLAN Not.*, 43(6):33–44, 2008.

[45] Narendran Sachindran, J. Eliot B. Moss, and Emery D. Berger. Mc2: high-performance garbage collection for memory-constrained environments. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 81–98, New York, NY, USA, 2004. ACM.

[46] Patrick M. Sansom and Simon L. Peyton Jones. Generational garbage collection for haskell. In *In Functional Programming Languages and Computer Architecture*, pages 106–116. ACM Press, 1993.

[47] Robert A. Saunders. The LISP system for the Q–32 computer. pages 220–231.

[48] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *SIGPLAN Not.*, 19(5):157–167, 1984.

[49] David Michael Ungar. *The Design and Evaluation of A High Performance Smalltalk System*. PhD thesis, EECS Department, University of California, Berkeley, Feb 1986.

[50] Stuart A. Yeates and Michel De Champlain. Design of a garbage collector using design patterns. In *Proceedings of the Twenty-Fifth Conference of (TOOLS) Pacific*, pages 77–92, 1997.

[51] T. Yuasa. Real-time garbage collection on general-purpose machines. *J. Syst. Softw.*, 11(3):181–198, 1990.

[52] Benjamin Zorn. Barrier methods for garbage collection. Technical report, 1990.

[53] Benjamin Goth Zorn. *Comparative performance evaluation of garbage collection algorithms*. PhD thesis, 1989.