ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Performance analysis of heap managers using binary instrumentation

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΓΕΩΡΓΙΟΣ ΓΕΡΑΣΙΜΟΣ Α.
ΚΡΑΝΗΣ

Επιβλέπων: Δημήτριος Σούντρης
Καθηγητής ΕΜΠ

Αθήνα, Ιούλιος 2013

Εθνικο Μετσοβιο Πολυτεχνειο
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Performance analysis of heap managers using binary instrumentation

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΓΕΩΡΓΙΟΣ ΓΕΡΑΣΙΜΟΣ Α. ΚΡΑΝΗΣ

Επιβλέπων: Δημήτριος Σούντρης
Καθηγητής ΕΜΠ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την _______________.

Δημήτριος Σούντρης	Κιαμάλ Πεκμεστζή	Γεώργιος Οικονομάκος
Καθηγητής ΕΜΠ	Καθηγητής ΕΜΠ	Λέκτορας ΕΜΠ

Αθήνα, Ιούλιος 2013.

ΓΕΩΡΓΙΟΣ ΓΕΡΑΣΙΜΟΣ Α. ΚΡΑΝΗΣ
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Abstract

The scope of this thesis was the development of a performance analysis methodology of heap managers using Pin from Intel, a dynamic binary instrumentation framework. This thesis focuses on fragmentation and time cost metrics, though it provides a generic framework through which additional measurements can be derived. The code has been tailored to the specific behavior of regular glibc memory manager as well as NTUA / ICCS dmmlib, a dynamic memory management framework.

Keywords:heap manager,performance, fragmentation, metrics, pin, pintool, dynamic memory manager,glibc,parsec,spec,malloc

Contents

Chapter 1

Introduction

1.1 Instrumentation

Instrumentation is an essential part of the software development process in all stages, from prototyping to quality assurance. Having tools that extract as much as information as possible with the least overhead and at minimum deployment cost can be proven indispensable.

1.2 Subject

Dynamic memory management is a critical component of an operating system. Its performance can extensively impact the responsiveness of the os, therefore optimal algorithms that efficiently manage the available virtual address space have been an active area of research. This is especially true in the context of embedded systems were such resources are scarce. A developer of an embedded system often has to tailor the heap manager to the target appliance. A heap manager instrumentation tool would enable him to make such decisions accurately without resorting to complex instruction level emulation setups. Specifically, a dynamic binary instrumentation tool would require no knowledge of the memory manager internals or even its source code. The tool would intercept memory management request calls to construct its own representation of the manager's state and ,based on that, output the respective statistics.Such tool could also prove useful while developing a heap manager or in an educational context.

1.3 Document structure

- Chapter 2 introduces us to dynamic binary instrumentation and the pin framework.
- Chapter 3 familiarizes the reader with dynamic memory management.
- Chapter 4 describes the design and implementation of our tool.
- Chapter 5 presents the resultant data.
- Chapter 6 summarizes this document providing areas of future improvement.

Chapter 2

Dynamic binary instrumentation

2.1 Why dynamic binary instrumentation?

Instrumentation is the act of measuring the performance of an application. Dynamic instrumentation refers to such an act at runtime. Static analysis cannot reveal the whole program behavior when it's dependent on dynamic context. Binary instrumentation is done using unaltered compiled code as opposed to source based instrumentation which requires editing the program's source code. Source code based instrumentation is obviously more flexible however instrumenting a binary is more convenient or even the only option when the source code isn't available. The convenience offered by dynamic binary instrumentation has also drawbacks. One must always keep in mind that the very act of instrumenting potentially alters the behavior of the program being measured especially the temporal one. Therefore for such cases where the noise being inserted by instrumentation is comparable to the measured value hardware emulation should be preferred. However such setups are orders of magnitude slower and far from trivial to configure relative to applying a precompiled instrumentation tool.

2.2 Intel pin

Pin's[1] behavior is analogous to the function of a jit compiler. It consumes the original binary assembly and outputs instrumented assembly based on the instrumentation rules (instrumentation routines) that the pin tool provides. Pin instruments basic instruction blocks just before they get executed and caches the resulting code. Instrumented assembly contains replaced or injected routines called analysis routines provided by the pintool.
In detail, when a binary loads, the pin framework injects itself in an appropriate position in the binary's address space. Instead of the original instructions the

framework executes instrumented code from the code cache, passing static and runtime data (state) to analysis routines.

Pin offers various levels of instrumentation granularity, that is, instrumentation routines can be triggered upon entering a code section, a routine, an instruction block or a single instruction. Analysis routines themselves can be placed relative to individual instructions, instruction blocks, or routines of the original binary code.

It is obvious, that since instrumentation routines get executed once, while analysis routines get executed many times, it makes sense to try to shift computationally intensive code from analysis to instrumentation routines whenever possible.

As we mentioned before, analysis routines can be passed runtime data. This data can be CPU state, instruction operands, routine arguments and return values as well as threading environment state, presented in a platform agnostic way.

Analysis routines make the bulk of the computational burden. Therefore pin offers fast call linkages and, even more importantly, analysis routine inlining when a routine has no jump instructions. The later can be effectively combined with the conditional instrumentation API, where an analysis routine is split into an inlineable one that gets called every time and a noninlineable that gets conditionally called.

Great attention must be paid when dealing with multithreaded code. Pin analysis routines are not allowed to use native threading facilities. Instead they must rely on locking primitives, thread management and thread local storage provided by the framework. That's why pin links to special non thread safe standard library routines.

Chapter 3

Dynamic memory management

In all but the simplest of programs the exact size of memory required for data storage is only known at runtime. Therefore most programming languages offer facilities for dynamic memory allocation. In the C language this is accomplished by the malloc family of routines. In simple terms, the program requests more memory by calling malloc and, when the memory block isn't needed any more, calls free to release it. Implementation of these routines is the job of a dynamic memory manager or otherwise called as heap manager. In turn, the heap manager draws its available memory from two sources. The first source is the traditional heap, a contiguous portion of the data segment, the end of which is pointed to by the program break pointer. The traditional heap size can be manipulated by the brk/sbrk system calls. The second source is memory mapped regions. This is accomplished by the mmap system call which maps new memory pages in the process' virtual address space. From now on these sources will be referred to as mapped objects. The heap manager allocates this pool of available memory (from both sources) to the received malloc requests by balancing between low fragmentation of space and quick response time according to various algorithms. The returned chunks will be referred to as memory objects in this paper.
The following sections list excerpts from the official documentation. The listed functions and its arguments/options are extensively used in our code.

3.1 malloc() manpage - Excerpts[10] of interest

3.1.1 Name

malloc, free, calloc, realloc - allocate and free dynamic memory

3.1.2 Synopsis

#include <stdlib.h>
void *malloc(size_t *size);*
void free(void **ptr);*
void *calloc(size_t *nmemb, size_t size);* **void *realloc(void** **ptr, size_t size);*

3.1.3 Description

The **malloc***() function allocates size bytes and returns a pointer to the allocated memory. The memory is not initialized. If size is 0, then* **malloc***() returns either* NULL, *or a unique pointer value that can later be successfully passed to* **free***().*

The **free***() function frees the memory space pointed to by ptr, which must have been returned by a previous call to* **malloc***(),* **calloc***() or* **realloc***(). Otherwise, or if* free(ptr) *has already been called before, undefined behavior occurs. If ptr is* NULL, *no operation is performed.*

The **calloc***() function allocates memory for an array of nmemb elements of size bytes each and returns a pointer to the allocated memory. The memory is set to zero. If nmemb or size is 0, then* **calloc***() returns either* NULL, *or a unique pointer value that can later be successfully passed to* **free***().*

The **realloc***() function changes the size of the memory block pointed to by ptr to size bytes. The contents will be unchanged in the range from the start of the region up to the minimum of the old and new sizes. If the new size is larger than the old size, the added memory will not be initialized. If ptr is* NULL, *then the call is equivalent to mal- loc(size), for all values of size; if size is equal to zero, and ptr is not* NULL, *then the call is equivalent to* free(ptr). *Unless ptr is* NULL, *it must have been returned by an earlier call to* **malloc***(),* **cal- loc***() or* **realloc***(). If the area pointed to was moved, a* free(ptr) *is done.*

3.1.4 Return Value

The **malloc***() and* **calloc***() functions return a pointer to the allocated memory that is suitably aligned for any kind of variable. On error, these functions return* NULL. NULL *may also be returned by a successful call to* **malloc***() with a size of zero, or by a successful call to* **cal- loc***() with nmemb or size equal to zero.*

The **free**() *function returns no value.*

The **realloc**() *function returns a pointer to the newly allocated memory, which is suitably aligned for any kind of variable and may be different from ptr, or* NULL *if the request fails. If size was equal to 0, either* NULL *or a pointer suitable to be passed to* **free**() *is returned. If* **realloc**() *fails the original block is left untouched; it is not freed or moved.*

3.2 mmap() manpage - Excerpts[10] of interest

3.2.1 Name

mmap, munmap - map or unmap files or devices into memory

3.2.2 Synopsis

#include <sys/mman.h>
void *mmap(void ****addr, size_t length, int prot, int flags,* **int *fd, off_t offset);*
**int munmap(void ****addr, size_t length);*

3.2.3 Description

mmap() creates a new mapping in the virtual address space of the call- ing process. The starting address for the new mapping is specified in addr. The length argument specifies the length of the mapping.

If addr is NULL, then the kernel chooses the address at which to create the map- ping; this is the most portable method of creating a new map- ping. If addr is not NULL, then the kernel takes it as a hint about where to place the mapping; on Linux, the mapping will be created at a nearby page boundary. The address of the new mapping is returned as the result of the call.

The contents of a file mapping (as opposed to an anonymous mapping; see MAP_ANONYMOUS below), are initialized using length bytes starting at offset offset in the file (or other object) referred to by the file descriptor fd. offset must be a multiple of the page size as returned by sysconf(_SC_PAGE_SIZE).

The flags argument determines whether updates to the mapping are visi- ble to other processes mapping the same region, and whether updates are carried through to the underlying file. This behavior is determined by including exactly one of the following values in flags:

MAP_PRIVATE] Create a private copy-on-write mapping. Updates to the map- ping are not visible to other processes mapping the same file, and are not carried through to the underlying file. It is unspecified whether changes made to the file after the **mmap**() call are visible in the mapped region.

In addition, zero or more of the following values can be ORed in flags:

MAP_ANONYMOUS] The mapping is not backed by any file; its contents are initial- ized to zero. The fd and offset arguments are ignored; however, some implementations re- quire fd to be -1 if MAP_ANONYMOUS (or MAP_ANON) is specified, and portable applications should ensure this. The use of MAP_ANONYMOUS in conjunction with MAP_SHARED is supported on Linux only since kernel 2.4.

munmap()

The **munmap**() *system call deletes the mappings for the specified address range, and causes further references to addresses within the range to generate invalid memory references. The region is also automatically unmapped when the process is terminated. On the other hand, closing the file descriptor does not unmap the region.*

The address addr must be a multiple of the page size. All pages con- taining a part of the indicated range are unmapped, and subsequent ref- erences to these pages will generate **SIGSEGV**. *It is not an error if the indicated range does not contain any mapped pages.*

3.2.4 Return Value

On success, **mmap**() *returns a pointer to the mapped area. On error, the value* **MAP_FAILED** *(that is, (void *) -1) is returned, and errno is set appropriately. On success,* **munmap**() *returns 0, on failure -1, and errno is set (probably to* **EINVAL***).*

3.3 sbrk() manpage - Excerpts[10] of interest

3.3.1 Name

brk, sbrk - change data segment size

3.3.2 Synopsis

#include <unistd.h>
int brk(void **addr);*
void *sbrk(intptr_t *increment);*

3.3.3 Description

brk*() and* **sbrk***() change the location of the program break, which defines the end of the process's data segment (i.e., the program break is the first location after the end of the uninitialized data segment). Increasing the program break has the effect of allocating memory to the process; decreasing the break deallocates memory.*

brk*() sets the end of the data segment to the value specified by addr, when that value is reasonable, the system has enough memory, and the process does not exceed its maximum data size (see* setrlimit(2)*).*

sbrk*() increments the program's data space by increment bytes. Calling* **sbrk***() with an increment of 0 can be used to find the current location of the program break.*

3.3.4 Return Value

On success, **brk***() returns zero. On error, -1 is returned, and errno is set to* **ENOMEM***. (But see Linux Notes below.)*

On success, **sbrk***() returns the previous program break. (If the break was increased, then this value is a pointer to the start of the newly allocated memory). On error, (void *) -1 is returned, and errno is set to* **ENOMEM***.*

Chapter 4

Design and implementation

We used image instrumentation mode to wrap with analysis code the malloc and mmap family of routines. The analysis code before each routine captures the call arguments and stores them into thread local storage, while the code after uses these arguments plus the return value to deduce the exact event details. It then proceeds to update the global representation of dynamic memory state. For storing said state, we use two binary search trees, one for the areas allocated by malloc referred to as mem objects(Listing 4.1) and one for the areas allocated by the Mmap family referred to as mmap objects(Listing 4.2). Each object contains the starting address (which is the key of the object) and the respective size. The binary search tree, even though slower on average than a hash map in search and insertion, is advantageous in traversal and range searches, operations that are used the most by our pin tool. We also monitor sbrk calls to keep up to date info on heap extents(Listing 4.3).
The variants of malloc, realloc and calloc are often implemented by dynamic memory managers as calls to malloc itself. Therefore, depending on the memory manager to be instrumented, we emulate realloc and calloc as malloc events, suppressing the respective analysis code due to internal calls to malloc by the memory manager(Listing 4.4). The same technique has also been used for the mremap function.
Our image instrumentation code ignores the LINUX loader library, because the contained internal bootstrapping malloc functions would be seen as regular dynamic memory management calls (Listing 4.5).
We have also wrapped the memory management routines around timing analysis functions using the TSC CPU register (Listing 4.6). In CPUs with the constant TSC feature, that register can be used as a wall time clock.[8] If power management is turned off (frequency scaling) that in turn can be used as a total cycle count. However the routine to be measured will be intertwined with idle user processes as well as kernel mode cycles. The actual usefulness of this metric will be checked in the results.
We also implemented optional instrumentation of memory access instructions that

Listing 4.1: mem objects

```
class memobject{
public:
  memobject (ADDRINT retaddr, int reqsize, int thread_id)
    :addr(retaddr),size(reqsize),reads(0),writes(0),
     is_critical(0),creator_thread(thread_id),owner(NULL)
  {

  }
  memobject()
  {
    //      InitLock(&(this->lock));
  }

  inline int is_parent(ADDRINT addr){
    return ((this->addr<=addr)&&(addr<(this->addr+this->size)));
  }

  ADDRINT addr;
  int size;
  int reads;
  int writes;
  int is_critical;
  unsigned int creator_thread;
  PIN_LOCK lock;
  mmapobject * owner;
};
```

Listing 4.2: mmap objects

```
class mmapobject{
public:
  mmapobject (ADDRINT retaddr,int reqsize,int thread_id)
    :addr(retaddr),size(reqsize),reads(0),writes(0),
     is_critical(0),creator_thread(thread_id)
  {

  }
  mmapobject()
  {
    //      InitLock(&(this->lock));
  }
  ADDRINT addr;
  size_t size;
  int reads;
  int writes;
  int is_critical;
  int creator_thread;
  PIN_LOCK lock;
};
```

correspond to dynamically managed memory. In this mode, analysis code is placed before executing an instruction that operates on a memory address that tries to match it to a memory object before updating its total read and write counts. This code is guarded by a pin specific in-line conditional check that returns true if this memory address might be related to dynamic memory. This fast guard function ensures that there is minimal slowdown from certainly unrelated memory operations. Since the guard function must contain no jump instructions in order to be inlined we had to manually tweak the returned Boolean expression to nudge the compiler into not using jump instructions (Listing 4.7)(Listing 4.8). Another optimization we applied was the implementation of a cache structure in front of the tree of memory objects.We also used atomic increments where possible(Listing 4.9). However it was quickly realized that the overhead of locking the data structures for multithreaded access led to a slowdown in the crude order of 50x thus rethinking was needed. That, plus the fact that this function was borderline in scope with the rest of the project means our results won't deal with it even though it's mostly functional.
Our heap profiler offers full multi-threading support. On thread start we allocate space in the TLS. On thread finish we aggregate the accumulated thread metrics into the global process metrics. Since our data structures are not inherently thread safe(and even if they were pin would link to non-thread safe variants) we protect

Listing 4.3: Monitoring sbrk

```
VOID AfterSbrk(ADDRINT ret,THREADID threadid)
{
  thread_data_t* tdata = get_tls(threadid);
  if ((void*)ret==MAP_FAILED){
    if (debug)
      fprintf(out,"thread %d AfterSbrk FAILED\n",threadid);
    ReleaseLock(&Heap::lock);
    return;
  }
  Heap::size+=(tdata->sbrk_size);
  if (Heap::address==0){
    if (debug)
      fprintf(out,"thread %d AfterSbrk First exec\n",threadid);
    Heap::address=ret;
  }
  if (debug){
    fprintf(out,"thread %d AfterSbrk addr (%p)\n",threadid,(void
        *)ret);
  }
  ReleaseLock(&Heap::lock);
}
```

Listing 4.4: Emulating realloc by using free/malloc analysis code

```
VOID BeforeRealloc(ADDRINT ptr, size_t size, THREADID threadid)
{
  if (((void*)ptr==NULL)||(size==0))
    return; //Nothing to see here!
  thread_data_t* tdata = get_tls(threadid);
  tdata->realloc_size=size;
  tdata->realloc_ptr=ptr;
  tdata->inbetween_realloc=1;
}

VOID AfterRealloc(ADDRINT ret, THREADID threadid)
{
  thread_data_t* tdata = get_tls(threadid);
  if (tdata->inbetween_realloc==0)
    return;
  tdata->inbetween_realloc=0;
  if ((void *)ret==NULL)
    return; //Nothing to see here!
  BeforeFree(tdata->realloc_ptr, threadid);
  BeforeMalloc(tdata->realloc_size, threadid);
  AfterMalloc(ret, threadid);
}
```

Listing 4.5: Ignoring unwanted images

```
VOID ImageLoad(IMG img, VOID *)
{
  fprintf(out, "Image %s loaded\n", IMG_Name(img).c_str());
  // if (!IMG_IsMainExecutable(img))
  //     return;
  string img_str=(IMG_Name(img).c_str());
  //if (img_str.find("ld-linux")!= string::npos)
  //   return;
  if ((!(img_str.find("libdmm")!= string::npos))&&(!(img_str.
      find("libc.so")!= string::npos)))
       return;
```

Listing 4.6: Time wrapping

```
VOID time(RTN rtn, uint32_t index)
{
  RTN_InsertCall(rtn,IPOINT_BEFORE,AFUNPTR(Before_time),
                  IARG_TSC,IARG_UINT32, index,IARG_ADDRINT,
                     UINT64_MAX,IARG_THREAD_ID,
                  IARG_CALL_ORDER,CALL_ORDER_LAST,IARG_END);
  RTN_InsertCall(rtn,IPOINT_AFTER,AFUNPTR(After_time),
                  IARG_TSC,IARG_UINT32, index,IARG_ADDRINT,
                     UINT64_MAX,IARG_THREAD_ID,
                  IARG_CALL_ORDER,CALL_ORDER_FIRST,IARG_END);
}
```

Listing 4.7: Nudging the compiler to produce the desired expression without jmp instructions

```
  ADDRINT _seglow=__sync_fetch_and_add(&seglow,0);
  ADDRINT _seghigh=__sync_fetch_and_add(&seghigh,0);
  ADDRINT diff1=addr-_seglow;
  ADDRINT diff2=_seghigh-addr;
  ADDRINT comp1=(_seglow-1);
  ADDRINT comp2=(_seghigh-1);
  ADDRINT bor=diff1|diff2|comp1|comp2;
  ADDRINT result2=(bor>>63)^1;//FIXME x86_64 specific
  /*(_seglow!=0)&&(_seghigh!=0)&&(addr>=_seglow)&&(addr<=
     _seghigh)
```

Listing 4.8: Resulting asm

```
00000000000668b0 <_Z7LSCheckm>:
   668b0:        mov    0x54b0c9(%rip),%rax
   668b7:        xor    %ecx,%ecx
   668b9:        lock xadd %rcx,(%rax)
   668be:        mov    0x54e1e3(%rip),%rax
   668c5:        xor    %edx,%edx
   668c7:        lock xadd %rdx,(%rax)
   668cc:        mov    %rdx,%rax
   668cf:        sub    $0x1,%rdx
   668d3:        sub    %rdi,%rax
   668d6:        sub    %rcx,%rdi
   668d9:        sub    $0x1,%rcx
   668dd:        or     %rdi,%rax
   668e0:        or     %rcx,%rax
   668e3:        or     %rdx,%rax
   668e6:        shr    $0x3f,%rax
   668ea:        xor    $0x1,%rax
   668ee:        retq
   668ef:        nop
```

Listing 4.9: Using atomic increment

```
inline void mem_access(memobject *obj,int isstore,THREADID
    threadid){
  if (isstore)
    obj->writes++;//From program correctness;
    //__sync_fetch_and_add(&obj->writes,1);
  else
    //obj->reads++;
    __sync_fetch_and_add(&obj->reads,1);
  if ((obj->creator_thread)!=threadid)
    //obj->is_critical=1;
    __sync_lock_test_and_set(&obj->is_critical,1);
}/*
```

Listing 4.10: Wrapping fprintf for thread safe access

```
inline void wrap_fprintf(FILE * stream, const char *fmt, ...)
{
  if (debug)
    PIN_MutexLock(&printlock);
  va_list args;
  va_start(args, fmt);
  vfprintf(stream,fmt, args);
  va_end(args);
  if (debug){
    fflush(stream);
    PIN_MutexUnlock(&printlock);
  }
}
```

them with pin provided locks. The same is true for the fprintf function(Listing 4.10). At a user configurable interval of memory management events we iterate over the entire tree of mmap objects (Listing 4.11). For each object we find the corresponding memory objects and calculate the respective spatial metrics. If the memory manager uses the traditional heap we calculate the spatial metrics using the unmatched memory objects of the previous step and the known heap extents. Finally we calculate the aggregate metrics for the whole allocated space.

Listing 4.11: Iterating over mapped regions

```
std::map<ADDRINT, mmapobject >::iterator next=mmapobjectmap.
    begin();
for (next=mmapobjectmap.begin();next!=(mmapobjectmap.end());
    next++)
  {
    float fragmentation;
    int allocedsize=0;
    int memobjnum=0;
    int totalspace=0;
    mmapobject * mmobj=&next->second;
    int isempty=1;
    int foundbegin=0;
    int foundend=0;
    memobjectmap_class::iterator memit,membegin,memend,
        membeforebegin;
    memit=memobjectmap.upper_bound(mmobj->addr+mmobj->size);
        //FIX_ME provide hints
```

```
    if (memit!=memobjectmap.end()){ //find end
        memobject *mobj=&(memit)->second;
        if ((mobj->addr+mobj->size)<=(mmobj->addr+mmobj->size
            )){
            memend=memit;
            foundend=1;
        }
    }
    memit=memobjectmap.upper_bound(mmobj->addr);
    if (memit!=memobjectmap.begin()){ //begin
        memobject *mobj=&(--memit)->second;
        if ((mobj->addr+mobj->size)<=((mmobj->addr)+(mmobj->
            size))){
            membegin=memit;
            foundbegin=1;
        }
    }
    if ((foundbegin==1)&&(foundend==1))
        isempty=0;
    if (isempty)
        fragmentation=0;
    else{
        membeforebegin=membegin; membeforebegin++;

        for (memit=memend;memit!=membeforebegin;memit++){
            allocedsize+=memit->second.size;
            memobjnum++;
        }
        totalspace=(memend->second.addr+memend->second.size-1)
            -mmobj->addr+1;
        fragmentation=(totalspace-allocedsize)/(1.0*totalspace
            );
        emptyspace_aggr+=totalspace-allocedsize;
        totalspace_aggr+=totalspace;
    }
    totalmmapedspace+=(mmobj->size);
    totalmmapobj++;
    totalfragmentation_aggr+=fragmentation;
    totalmemobj+=memobjnum;
    /*      fprintf(report, "(%p)=mmap(%zu) objnum(%d)
        totalspace(%d) allocedsize(%d)"
              "fragmentation (%f) percent\n",
              (void *)mmobj->addr,mmobj->size,
              memobjnum,totalspace,allocedsize,fragmentation
                  *100);*/
}
```

Chapter 5

Experimental Evaluation

5.1 Metrics

We calculated the following metrics:

- *Number of currently allocated memory objects,mmap objects*
- *Total size of currently allocated memory objects, total size of currently allocated mapped objects (traditional heap and mmaped)*
- *Mean and total fragmentation. Mean fragmentation gives equal weight to each mapped object regardless of size while total weights them by size. Both metrics ignore trailing free space. The above metrics are plotted in relation to heap events.*
- *Total number of malloc, free, calloc, realloc,mmap events, as well as total cycle cost, mean cycle cost and the respective standard deviation.*
- *Mean requested size of malloc, Mmap calls. Since we emulate calloc and realloc with the malloc analysis routine, their size arguments add to the malloc mean requested size metric.*
- *Lastly we calculate the instrumentation overhead.*

5.2 Benchmarks

We used select benchmarks from the parsec[3] and spec[1] CPU 2006 [12] benchmark suites and profiled the performance of both libc and dmmlib[9] dynamic

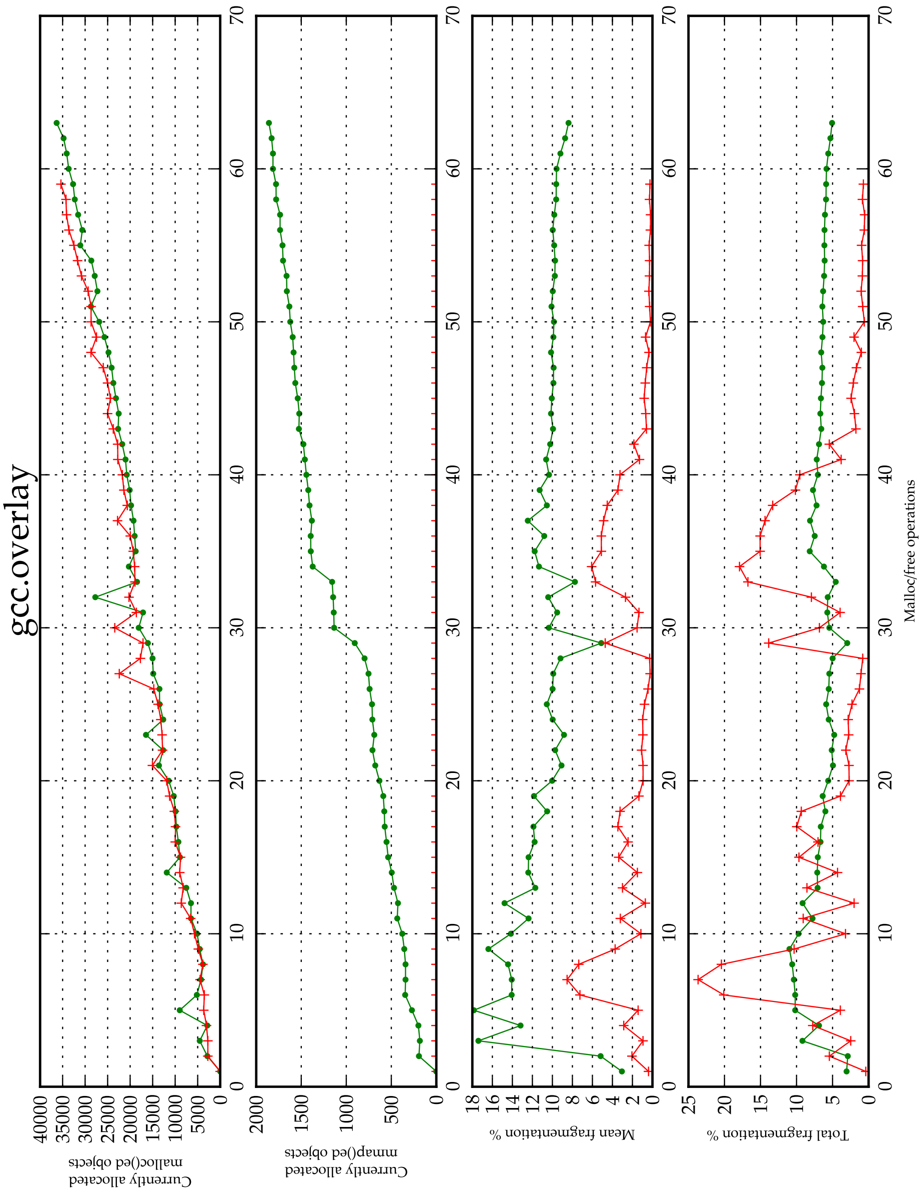
[1]The spec benchmarks have been used in a non-compliant manner

memory managers. Parsing of the raw text data was done with numpy[11, 4, 5, 2], a scientific python[13] library and Matplotlib[7] was used to plot them. If the available samples are more than 200 we subsample them by simply discarding in between data to improve rendering speed of the resulting encapsulated PostScript files. In some benchmarks with big numbers of allocated objects or where memory management events happen frequently enough, the calculation of fragmentation metrics or the frequency at which said calculation was called respectively created a prohibitively high instrumentation overhead. In these instances the calculation rate was manually tweaked to produce acceptable results.

5.3 Functionality evaluation[2] [3] [4]

5.3.1 gcc

Figure 5.1: gcc plot *a*



---

[2] dmmlib data represented by dots

[3] glibc data represented by crosses

[4] for the sake of brevity only the results referenced later are listed

Figure 5.2: gcc plot *b*

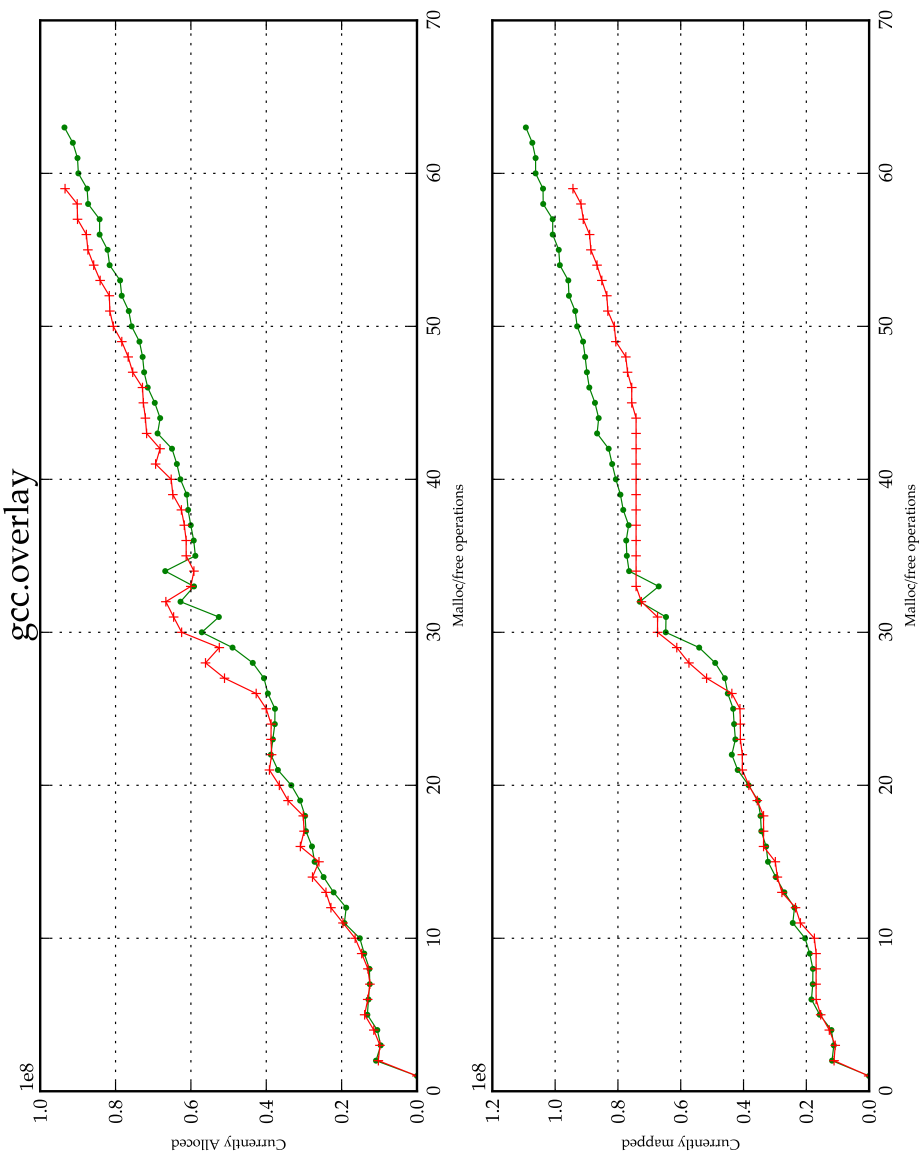


Table 5.1: gcc.dmmlib statistics

function	Total events	Mean requested size	Total cost	Mean cost	Std deviation
malloc	2924051	2388.517843	54819046854	18822.41687	47649.964231
free	2919547	0.0	14406047063	4328.69718	21180.73488
calloc	374890	0.0	19331482676	51565.746422	106061.245195
realloc	11617	0.0	501792465	43194.668589	173292.453388
mmap	42411	76700.596355	1565265969	36906.20506	17025.051259
Instrumentation slowdown	1.251000				
Base running time(s)	2398.678401				
Data downsampling factor	1				

Table 5.2: gcc.dmmlib debug statistics

function	Entries	Exits
malloc	2912434	2912434
free	3328033	3328033
mmap	42412	42412
realloc	11617	11617
calloc	374890	374890
munmap	40561	40561
mremap	0	0
sbrk	0	0

Table 5.3: gcc.libc statistics

function	Total events	Mean requested size	Total cost	Mean cost	Std deviation
malloc	2919539	2382.704128	732187999	289.055619	19178.099256
free	2919539	0.0	38618952	84.735468	794.961157
calloc	374887	0.0	1182072027	3153.142219	25378.265279
realloc	11617	0.0	90508735	7791.059224	110345.162181
mmap	13	1381612.30769	2273206	162371.857143	191054.844442
Instrumentation slowdown	1.487182				
Base running time(s)	524.489747				
Data downsampling factor	1				

Table 5.4: gcc.libc debug statistics

function	Entries	Exits
malloc	2533036	2533035
free	3328028	455759
mmap	14	14
realloc	11618	11617
calloc	374887	374887
munmap	12	12
mremap	0	0
sbrk	524	524

5.3.2 parsec.bodytrack

Figure 5.3: parsec.bodytrack plot *a*

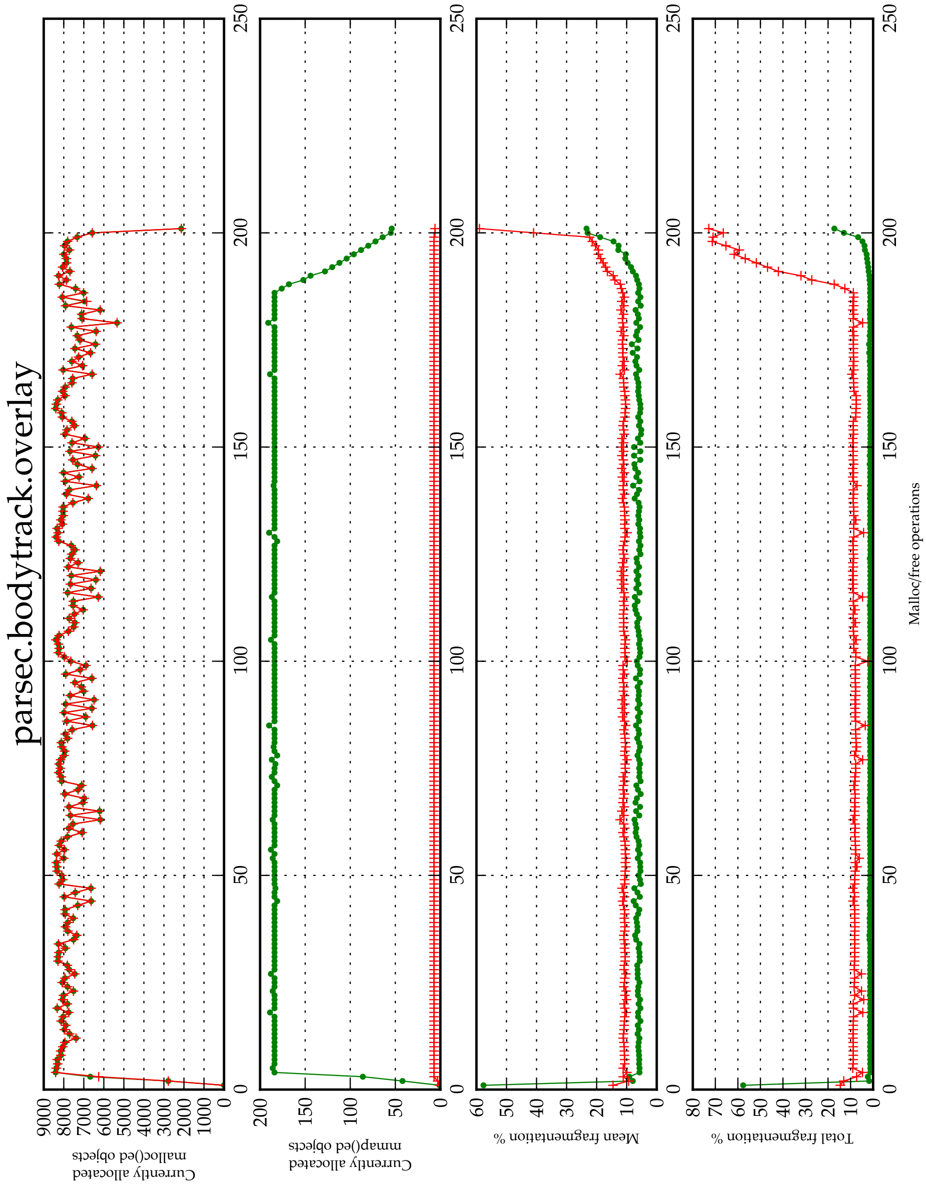


Figure 5.4: parsec.bodytrack plot b

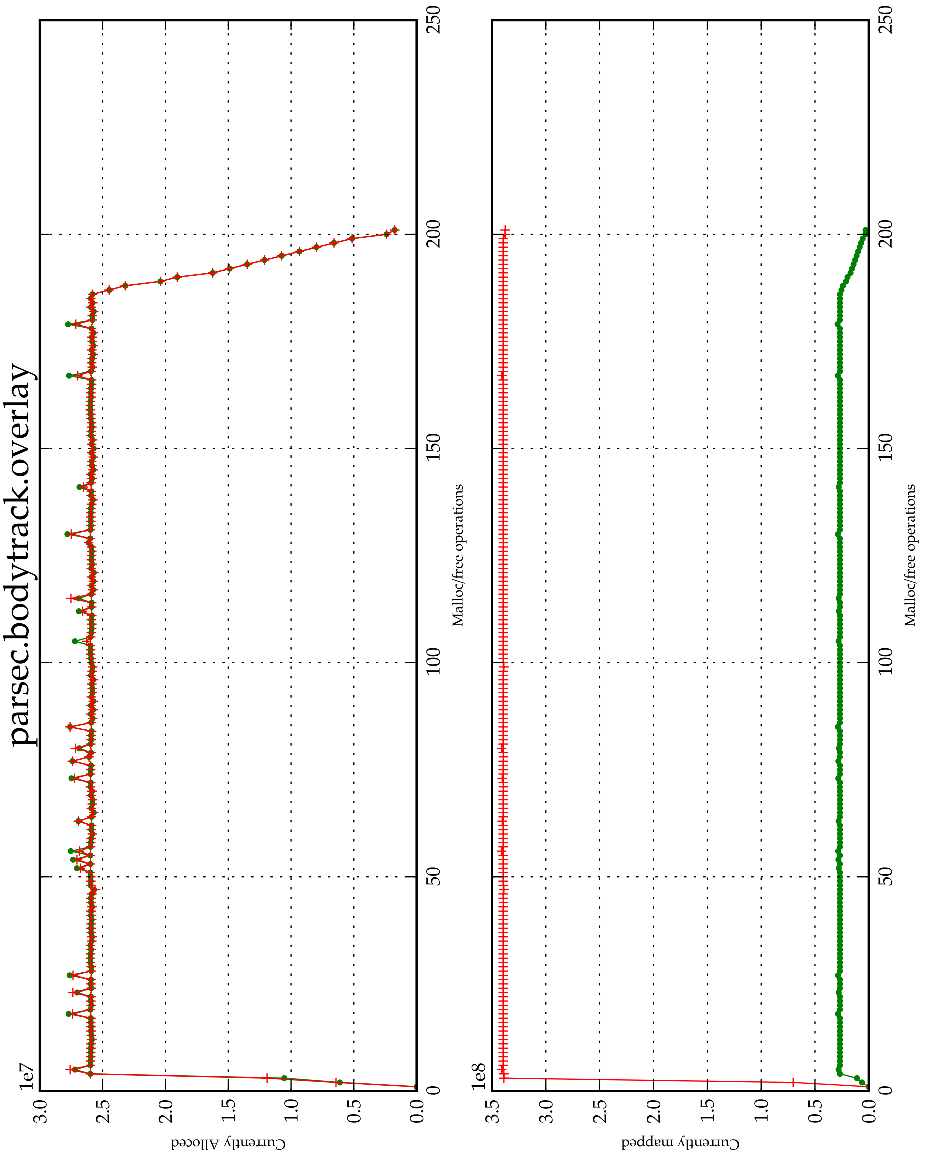


Table 5.5: parsec.bodytrack.dmmlib statistics

function	Total events	Mean requested size	Total cost	Mean cost	Std deviation
malloc	437368	9581.471982	15680764694	35852.565103	271484.761486
free	437368	0.0	3468108054	7646.330854	91404.240026
calloc	5	0.0	17316441	3463288.2	3237451.10571
realloc	0	0.0	0	nan	nan
mmap	14142	295927.455805	1424173944	68890.530837	261930.036934
Instrumentation slowdown	1.470006				
Base running time(s)	81.206158				
Data downsampling factor	44				

Table 5.6: parsec.bodytrack.dmmlib debug statistics

function	Entries	Exits
malloc	437368	437368
free	453565	453565
mmap	20673	20673
realloc	0	0
calloc	5	5
munmap	20623	20623
mremap	0	0
sbrk	0	0

Table 5.7: parsec.bodytrack.libc statistics

function	Total events	Mean requested size	Total cost	Mean cost	Std deviation
malloc	437368	9581.471982	655971136	1499.832258	140490.459677
free	437368	0.0	2198642	135.710265	4302.819463
calloc	5	0.0	15100623	3020124.6	3902492.38801
realloc	0	0.0	0	nan	nan
mmap	10	54053273.6	305847196	46758.476686	116175.145419
Instrumentation slowdown	1.477756				
Base running time(s)	77.579931				
Data downsampling factor	44				

Table 5.8: parsec.bodytrack.libc debug statistics

function	Entries	Exits
malloc	437364	437363
free	453565	16201
mmap	6541	6541
realloc	0	0
calloc	5	5
munmap	6535	6535
mremap	0	0
sbrk	2564	2564

5.3.3 parsec.fluidanimate

Figure 5.5: parsec.fluidanimate plot *a*

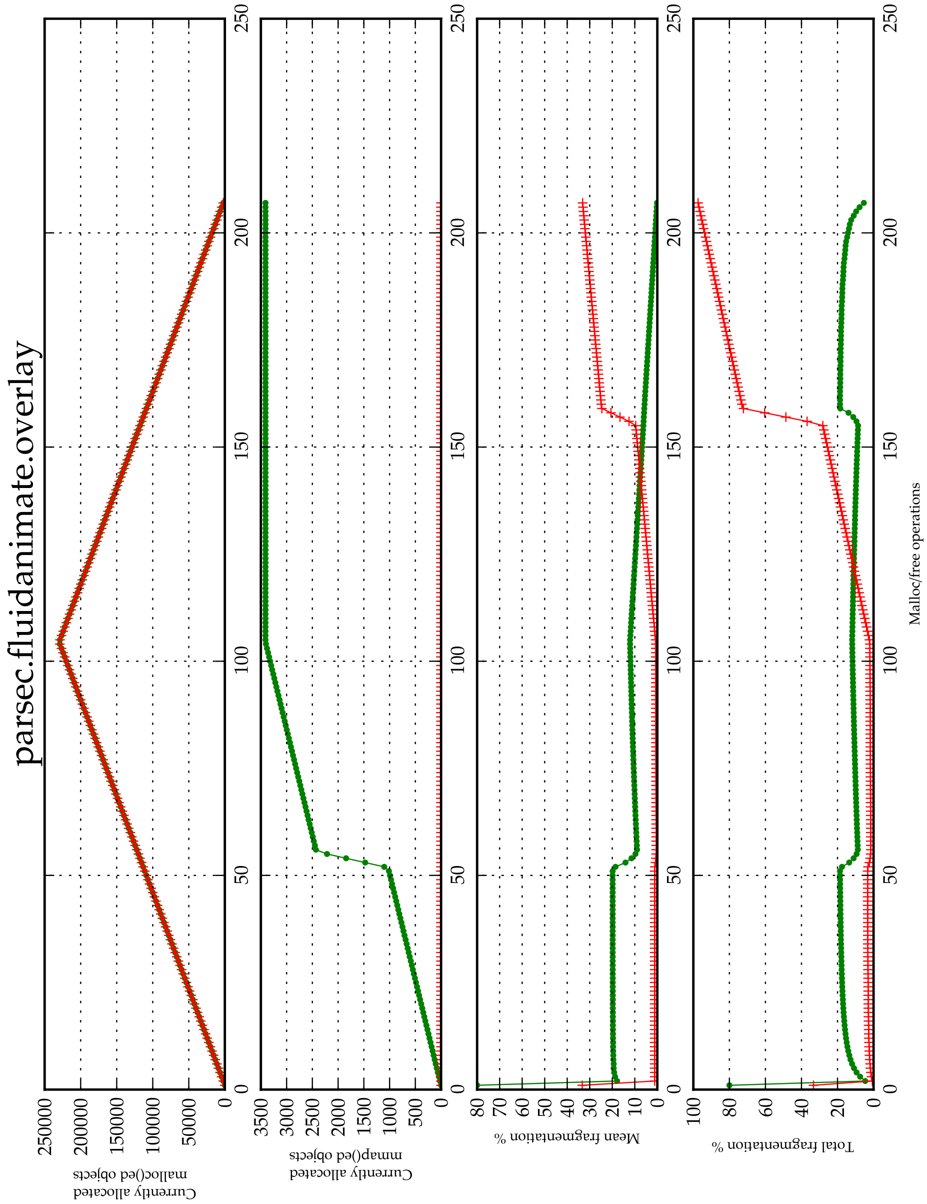


Figure 5.6: parsec.fluidanimate plot *b*

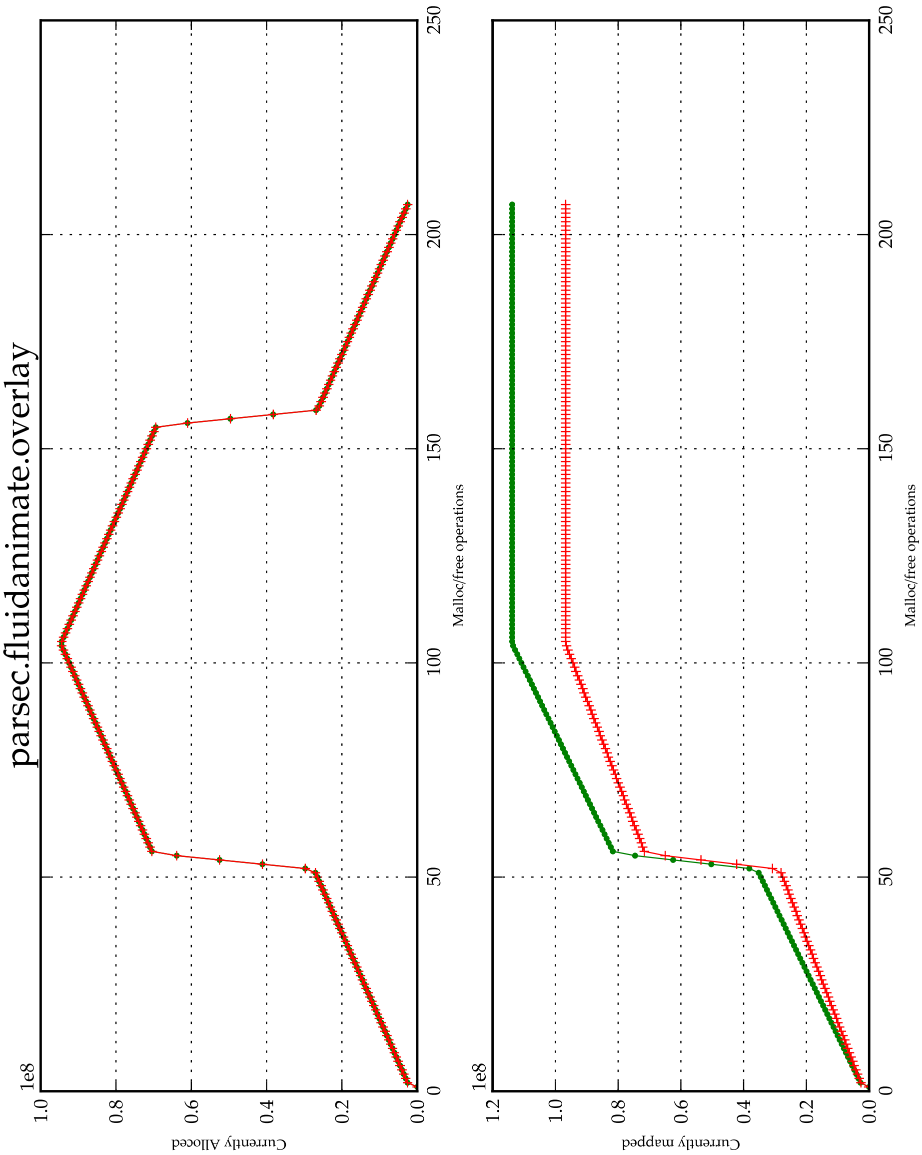


Table 5.9: parsec.fluidanimate.dmmlib statistics

function	Total events	Mean requested size	Total cost	Mean cost	Std deviation
malloc	229913	411.797906	5490924562	23882.618912	253081.72558
free	229913	0.0	105341732349	458150.95312	437517.325325
calloc	4	0.0	41954641	10488660.25	16054176.8377
realloc	0	0.0	0	nan	nan
mmap	3410	33357.775953	339735582	99221.840537	66617.999487
Instrumentation slowdown	1.240439				
Base running time(s)	188.319599				
Data downsampling factor	22				

Table 5.10: parsec.fluidanimate.dmmlib debug statistics

function	Entries	Exits
malloc	229913	229913
free	229928	229928
mmap	3424	3424
realloc	0	0
calloc	4	4
munmap	11	11
mremap	0	0
sbrk	0	0

Table 5.11: parsec.fluidanimate.libc statistics

function	Total events	Mean requested size	Total cost	Mean cost	Std deviation
malloc	229913	411.797906	263417009	1145.74466	58083.457738
free	229913	0.0	697688	63426.181818	151462.935228
calloc	4	0.0	16805138	4201284.5	6805182.2156
realloc	0	0.0	0	nan	nan
mmap	2	1038336.0	2006880	125430.0	206083.479554
Instrumentation slowdown	1.182512				
Base running time(s)	176.829302				
Data downsampling factor	22				

Table 5.12: parsec.fluidanimate.libc debug statistics

function	Entries	Exits
malloc	229910	229909
free	229928	11
mmap	16	16
realloc	0	0
calloc	4	4
munmap	11	11
mremap	0	0
sbrk	697	697

5.3.4 parsec.swaptions

Figure 5.7: parsec.swaptions plot *a*

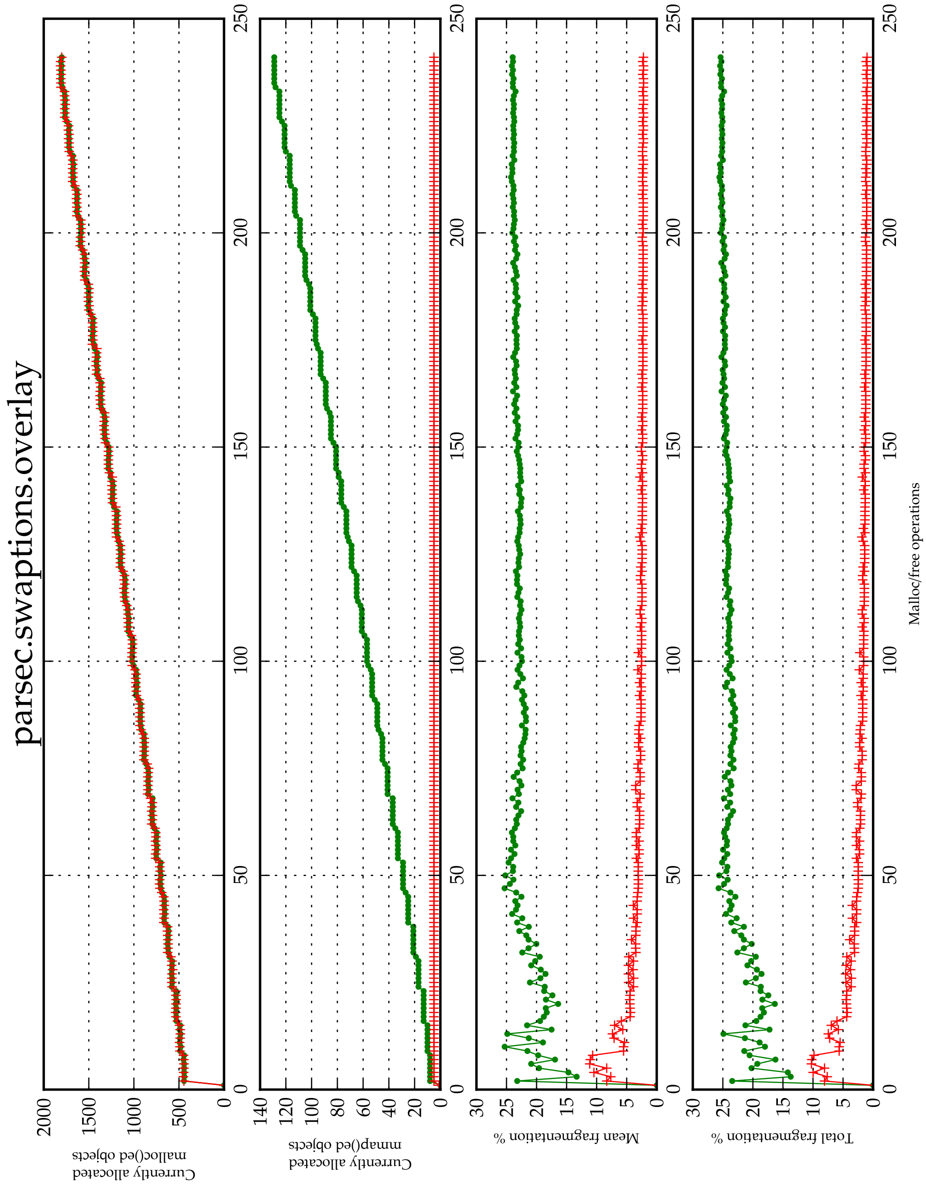


Figure 5.8: parsec.swaptions plot b

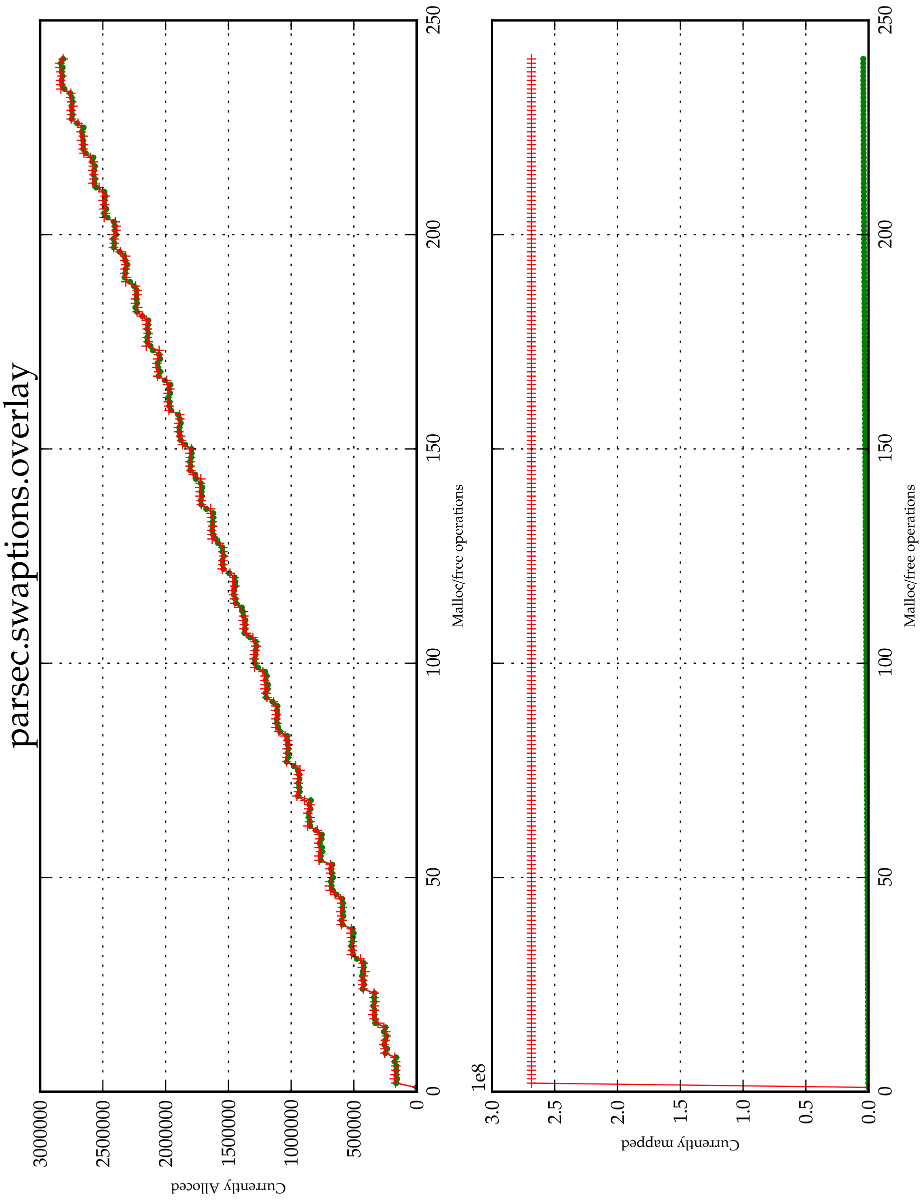


Table 5.13: parsec.swaptions.dmmlib statistics

function	Total events	Mean requested size	Total cost	Mean cost	Std deviation
malloc	48001800	1825.823532	964763466572	20098.485194	289645.086764
free	48001800	0.0	939548057609	19573.653623	278311.421185
calloc	4	0.0	25766425	6441606.25	3898758.51458
realloc	0	0.0	0	nan	nan
mmap	129	32768.0	34674365	258763.91791	1130749.31667
Instrumentation slowdown	2.595109				
Base running time(s)	143.764783				
Data downsampling factor	4				

Table 5.14: parsec.swaptions.dmmlib debug statistics

function	Entries	Exits
malloc	48001800	48001800
free	48000648	48000648
mmap	134	134
realloc	0	0
calloc	4	4
munmap	0	0
mremap	0	0
sbrk	0	0

Table 5.15: parsec.swaptions.libc statistics

function	Total events	Mean requested size	Total cost	Mean cost	Std deviation
malloc	48001800	1825.823532	15323435297	319.226291	26987.457811
free	48001800	0.0	737849	2816.217557	34172.640923
calloc	4	0.0	9855044	2463761.0	4259621.60378
realloc	0	0.0	0	nan	nan
mmap	4	134217728.0	1425694	158410.444444	272018.304786
Instrumentation slowdown	1.997918				
Base running time(s)	111.489341				
Data downsampling factor	4				

Table 5.16: parsec.swaptions.libc debug statistics

function	Entries	Exits
malloc	48001797	48001796
free	48000648	262
mmap	9	9
realloc	0	0
calloc	4	4
munmap	7	7
mremap	0	0
sbrk	2	2

5.3.5 splash2x.barnes

Figure 5.9: splash2x.barnes plot *a*

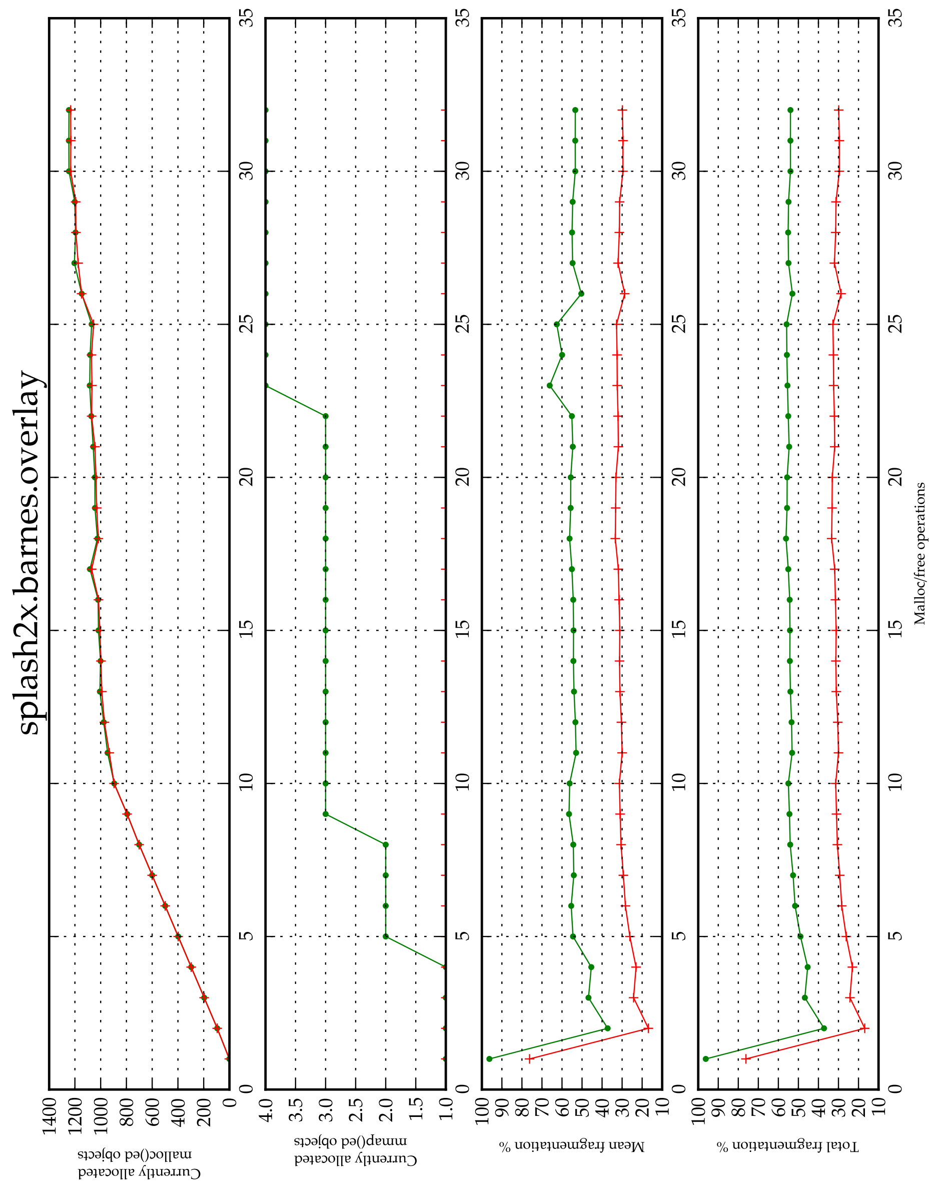


Figure 5.10: splash2x.barnes plot *b*

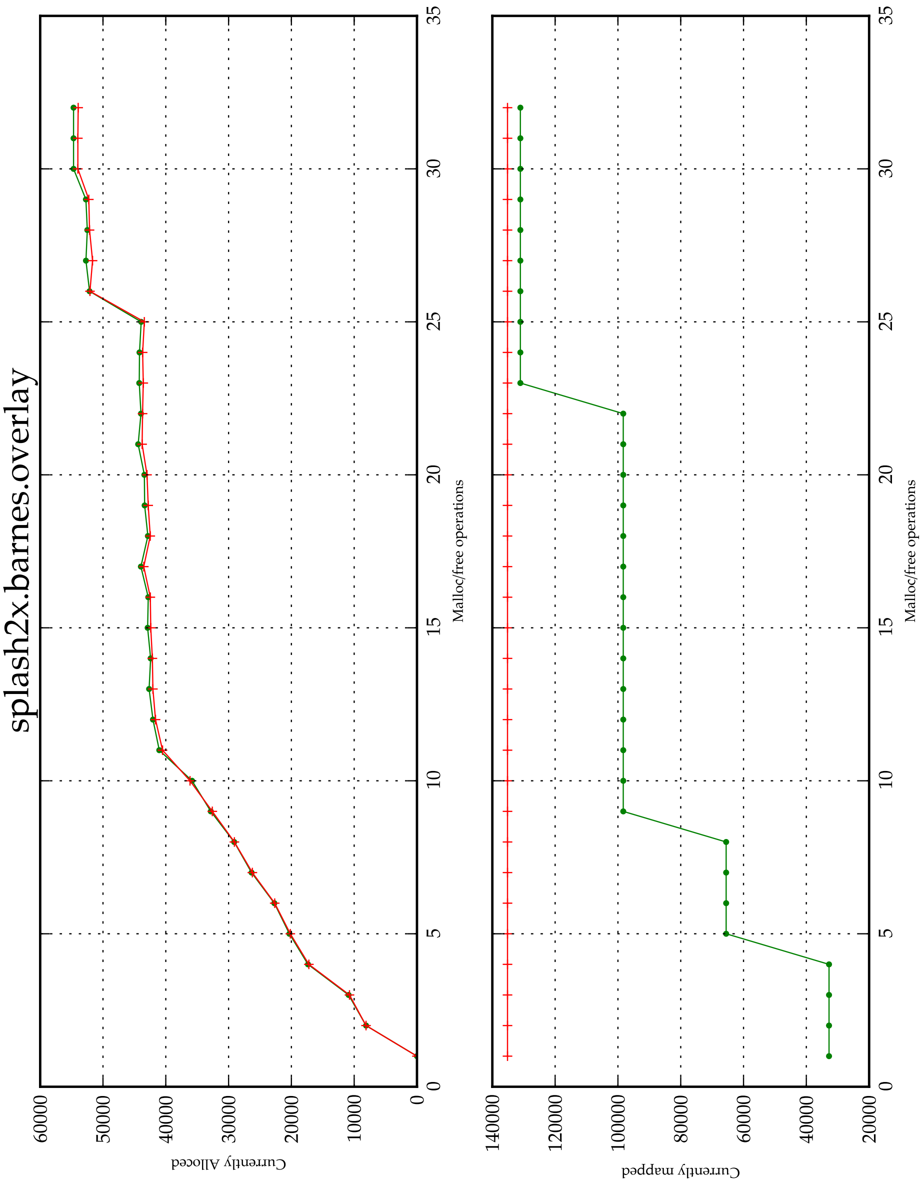


Table 5.17: splash2x.barnes.dmmlib statistics

function	Total events	Mean requested size	Total cost	Mean cost	Std deviation
malloc	2215	39.408578	99710112	45261.058557	1116147.70365
free	2215	0.0	59574780	58008.549172	1000969.55827
calloc	1	0.0	4385546	4385546.0	0.0
realloc	12	0.0	13189078	1099089.83333	3256473.46443
mmap	4	32768.0	1548870	193608.75	272997.625167
Instrumentation slowdown	1.004152				
Base running time(s)	73.037892				
Data downsampling factor	1				

Table 5.18: splash2x.barnes.dmmlib debug statistics

function	Entries	Exits
malloc	2203	2203
free	1027	1027
mmap	8	8
realloc	12	12
calloc	1	1
munmap	1	1
mremap	0	0
sbrk	0	0

Table 5.19: splash2x.barnes.libc statistics

function	Total events	Mean requested size	Total cost	Mean cost	Std deviation
malloc	2201	39.330304	54361232	24845.170018	620355.008778
free	2201	0.0	500145	23816.428571	104924.747414
calloc	1	0.0	6290514	6290514.0	0.0
realloc	12	0.0	12763623	1063635.25	2450234.82229
mmap	0	nan	956137	239034.25	345477.474649
Instrumentation slowdown	1.026259				
Base running time(s)	72.829441				
Data downsampling factor	1				

Table 5.20: splash2x.barnes.libc debug statistics

function	Entries	Exits
malloc	2189	2188
free	1027	21
mmap	4	4
realloc	13	12
calloc	1	1
munmap	1	1
mremap	0	0
sbrk	3	3

5.3.6 splash2x.fmm

Figure 5.11: splash2x.fmm plot *a*

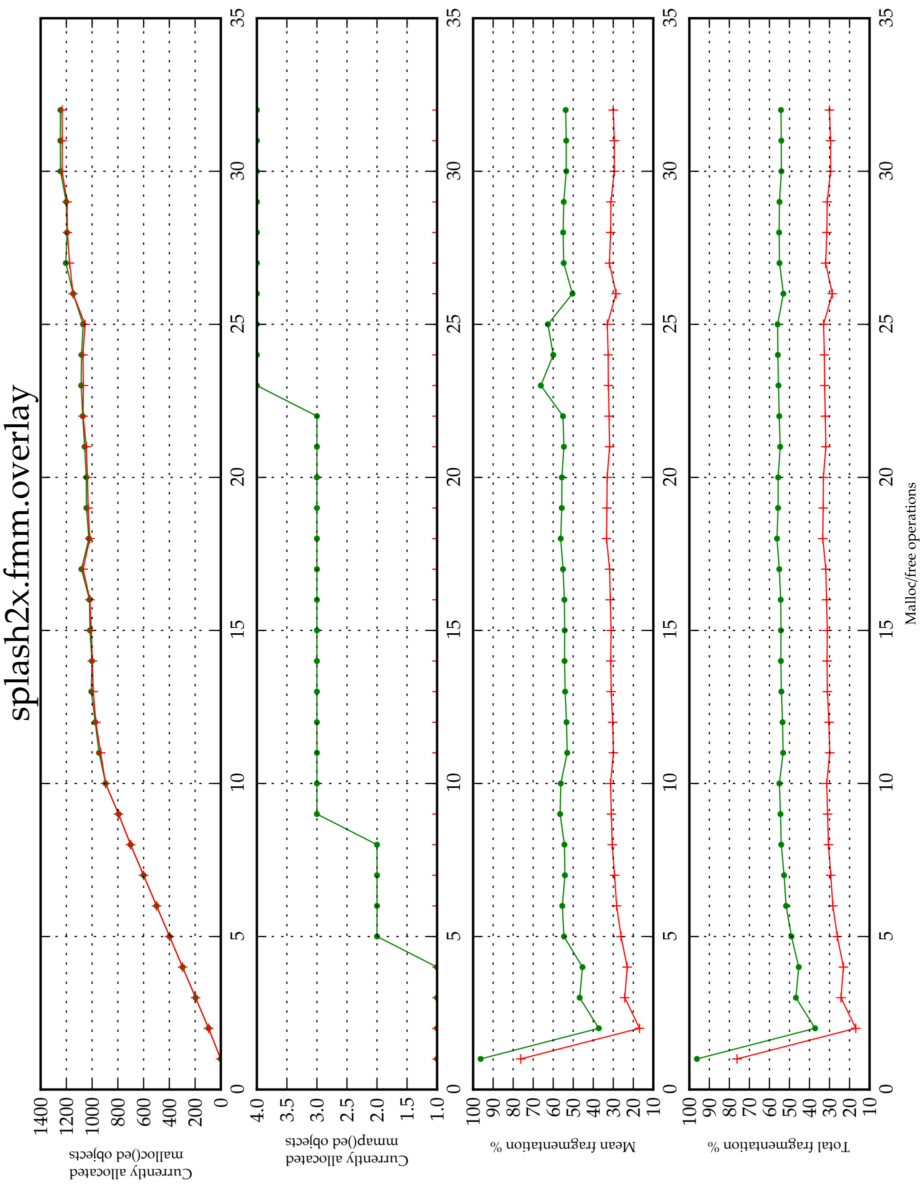


Figure 5.12: splash2x.fmm plot *b*

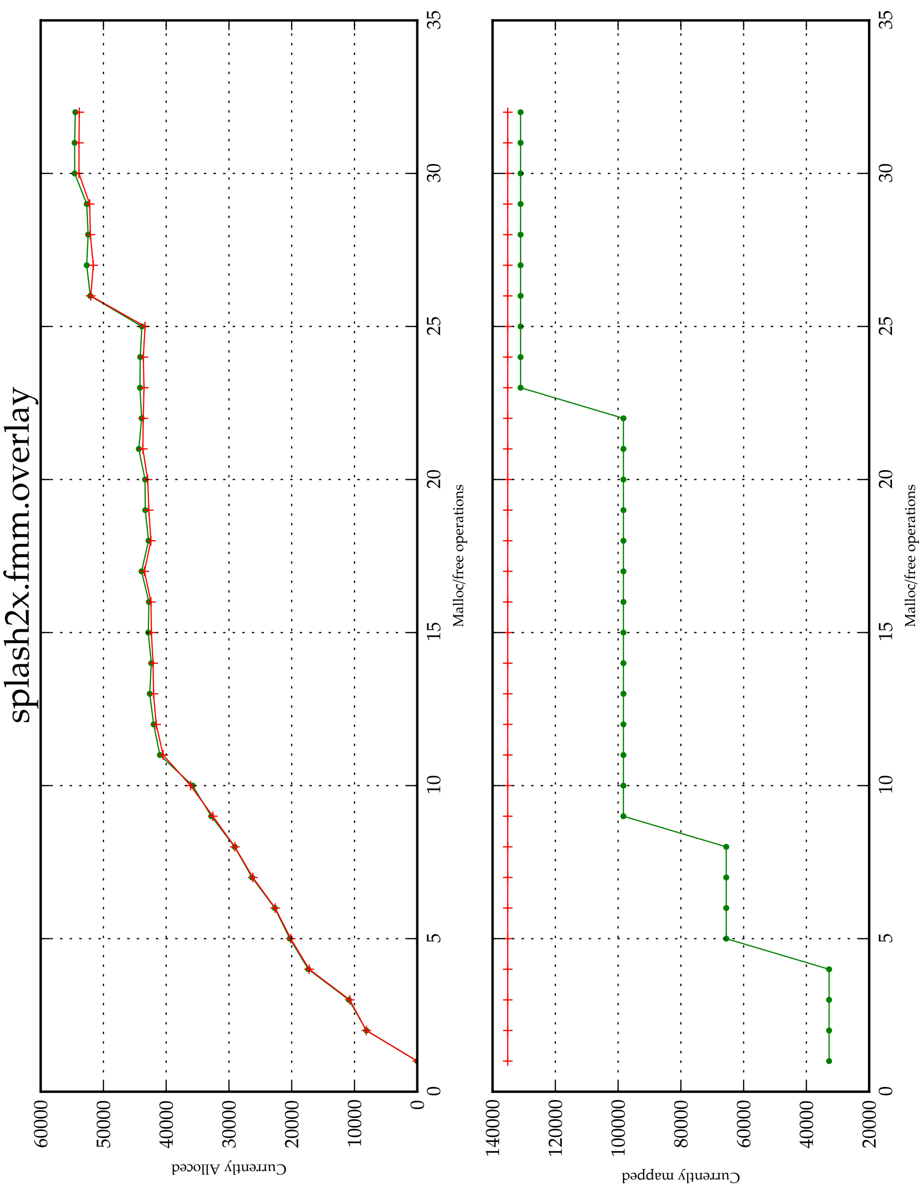


Table 5.21: splash2x.fmm.dmmlib statistics

function	Total events	Mean requested size	Total cost	Mean cost	Std deviation
malloc	2210	39.193213	134983027	61383.823101	1835930.59561
free	2210	0.0	59492500	58154.936461	1003061.53926
calloc	1	0.0	4495808	4495808.0	0.0
realloc	11	0.0	13460239	1223658.09091	3464418.39205
mmap	4	32768.0	1542750	192843.75	268787.663272
Instrumentation slowdown	1.020513				
Base running time(s)	45.482636				
Data downsampling factor	1				

Table 5.22: splash2x.fmm.dmmlib debug statistics

function	Entries	Exits
malloc	2199	2199
free	1023	1023
mmap	8	8
realloc	11	11
calloc	1	1
munmap	1	1
mremap	0	0
sbrk	0	0

Table 5.23: splash2x.fmm.libc statistics

function	Total events	Mean requested size	Total cost	Mean cost	Std deviation
malloc	2196	39.113388	54681302	25037.22619	613345.753832
free	2196	0.0	505786	24085.047619	106296.389349
calloc	1	0.0	6214174	6214174.0	0.0
realloc	11	0.0	13289612	1208146.54545	2560429.31778
mmap	0	nan	935065	233766.25	335361.76393
Instrumentation slowdown	1.000422				
Base running time(s)	46.140648				
Data downsampling factor	1				

Table 5.24: splash2x.fmm.libc debug statistics

function	Entries	Exits
malloc	2185	2184
free	1023	21
mmap	4	4
realloc	12	11
calloc	1	1
munmap	1	1
mremap	0	0
sbrk	3	3

5.3.7 tonto

Figure 5.13: tonto plot *a*

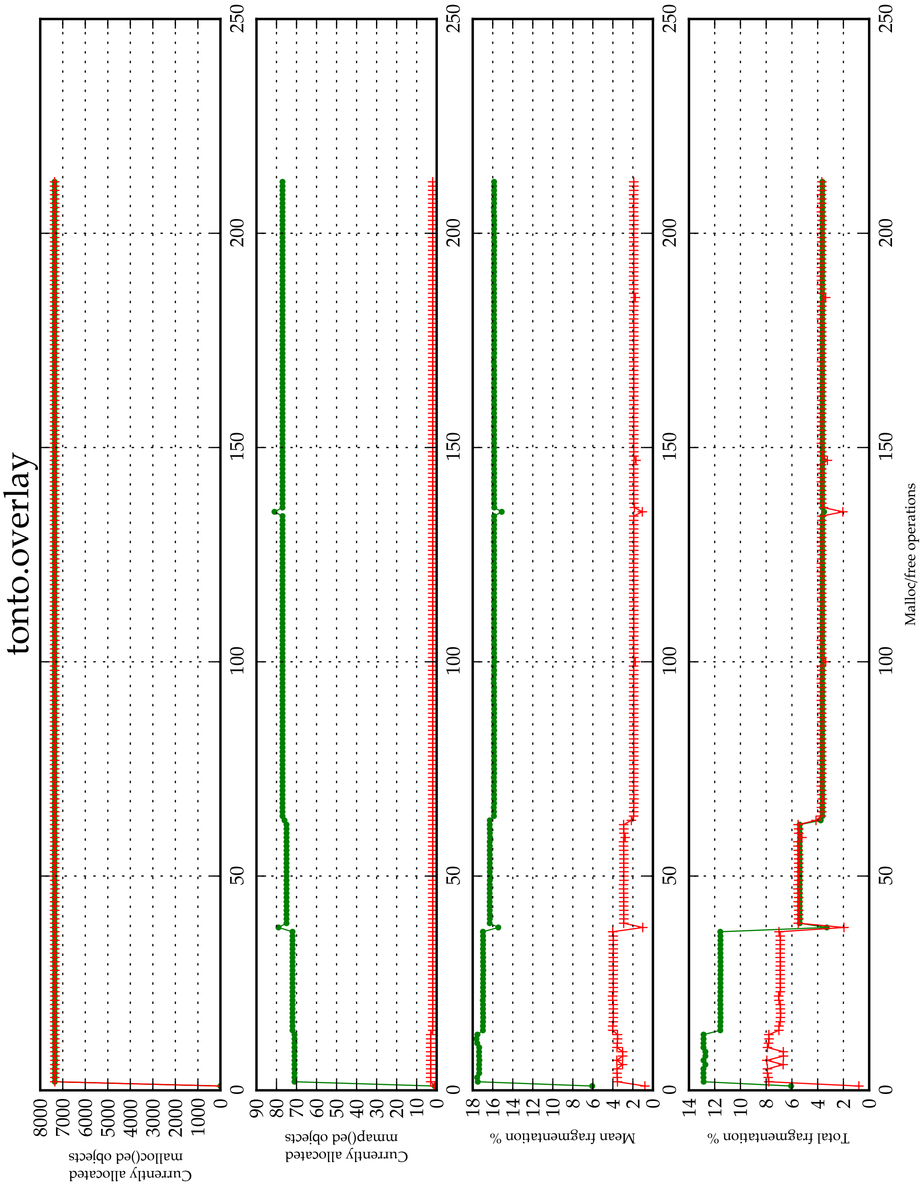


Figure 5.14: tonto plot *b*

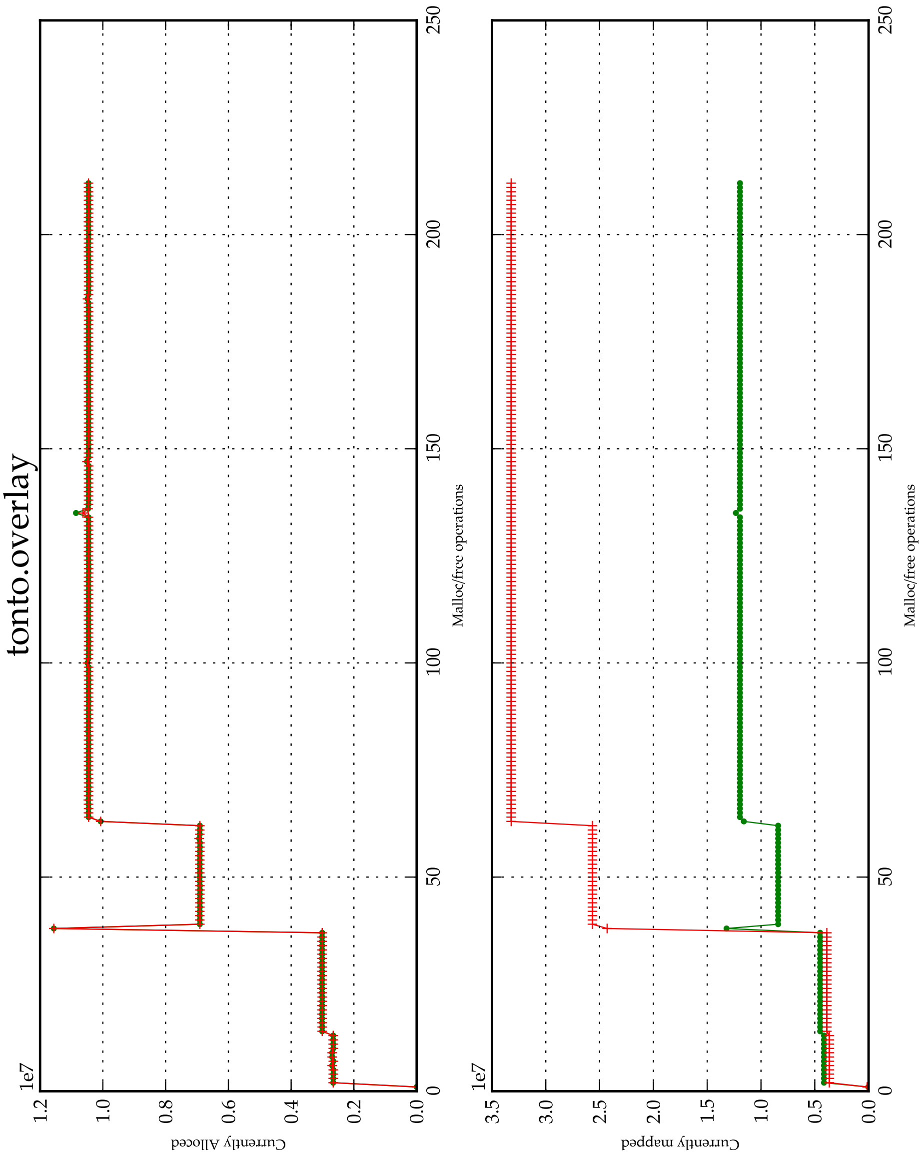


Table 5.25: tonto.dmmlib statistics

function	Total events	Mean requested size	Total cost	Mean cost	Std deviation
malloc	1056868683	374.389887	1431146743323	1354.138669	3096.617367
free	1056868683	0.0	1338048192968	1265.957387	4732.009073
calloc	3	0.0	80502114	26834038.0	25720546.9276
realloc	6	0.0	11595816	1932636.0	3748940.63388
mmap	1675509	149606.905488	47353806691	28262.324123	10171.691534
Instrumentation slowdown	2.217757				
Base running time(s)	1179.066572				
Data downsampling factor	5				

Table 5.26: tonto.dmmlib debug statistics

function	Entries	Exits
malloc	1056868677	1056868677
free	1056945681	1056945681
mmap	1675510	1675510
realloc	6	6
calloc	3	3
munmap	1675445	1675445
mremap	0	0
sbrk	0	0

Table 5.27: tonto.libc statistics

function	Total events	Mean requested size	Total cost	Mean cost	Std deviation
malloc	1056868683	374.389887	181388007657	171.627764	975.028532
free	1056868683	0.0	7176891	93.170077	1936.526992
calloc	2	0.0	9143080	4571540.0	3532113.0
realloc	6	0.0	10008953	1668158.83333	3290399.31144
mmap	8	4135936.0	1226753	136305.888889	239265.244157
Instrumentation slowdown	1.959927				
Base running time(s)	866.312864				
Data downsampling factor	5				

Table 5.28: tonto.libc debug statistics

function	Entries	Exits
malloc	1056868675	1056868675
free	1056945681	77030
mmap	9	9
realloc	7	6
calloc	3	2
munmap	9	9
mremap	0	0
sbrk	219	219

We observe the expected positive correlation between memory objects and mapped objects , either total size or population. We also observe the expected variance of heap activity between different benchmarks. Instrumentation speed ranges between 1x and 2.5x which we consider acceptable.
However we notice that the standard deviation of mean cost in cycles is very high. We don't know if this high variance is inherent in the memory management algorithm or it is due to cycles noise, either idle processes or kernel mode cycles or even instrumentation overhead.We also cant exclude the possibility of error in our timing implementation.
*In the splash family of benchmarks we observe that the total mapped memory is slightly higher for the glibc allocator. This is because the glibc allocator pads sbrk requests with additional bytes if they are smaller than M_TOP_PAD which has a default value of 128*1024, in aggreeance with the line of around 140000 bytes we see in the graphs. In bodytrack and swaptions benchmarks the mapped memory appears to be significantly higher for the glibc allocator. Actually it's an artifact that appears because glibc allocates thread specific arenas by using MAP_NORESERVE , speculating that it may never need the mapped space wholly. One may look up such details in glibc source code [6].*
Notice how in the tonto benchmark the currently mapped space of glibc stays significantly higher.This is because for a small moment the program requires that much memory (edge in currently allocated graph[5]).The glibc implementation, being heap based, cannot release this memory back to the system.This behavior is examined in detail in the next section.

5.4 Workload specific results

The above results suggested that our code produces sane output across a wide selection of workloads. However comparing two different allocators only gives answers regarding memory footprint and time cost. In order to gain more insight we present three cases where , while keeping the other conditions constant, we change a single allocator parameter iterating over a predefined range of values.

5.4.1 M_TOP_PAD

While keeping the other conditions constant we tweak M_TOP_PAD. M_TOP_PAD is a glibc allocator parameter that controls the minimum value of the size argument passed to sbrk. Whenever glibc runs out of space, assuming the request size isn't eligible for allocation via mmap ,it enlarges the heap by calling sbrk. Since sbrk is a system call, hence costly, it makes sense to preallocate space on the heap,

[5] note that the edge actually reaches $35 * 10^6$ but its not shown due to naive subsampling

in anticipation for more requests, minimizing the total number of sbrk calls. If the sbrk request is smaller than M_TOP_PAD it is padded to that size. However this may lead to a waste of space if the preallocated area is never actually used.

Figure 5.15: gcc plot *a*

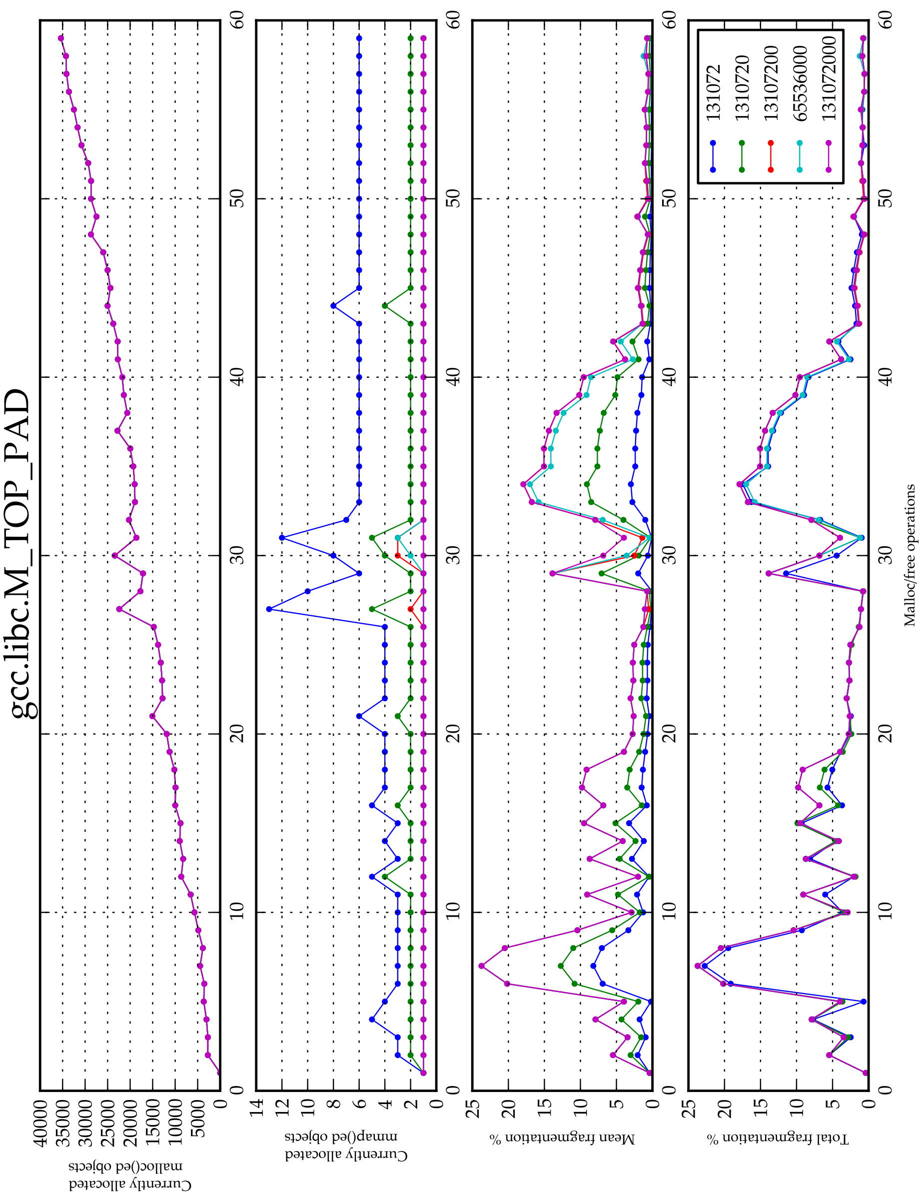


Figure 5.16: gcc plot *b*

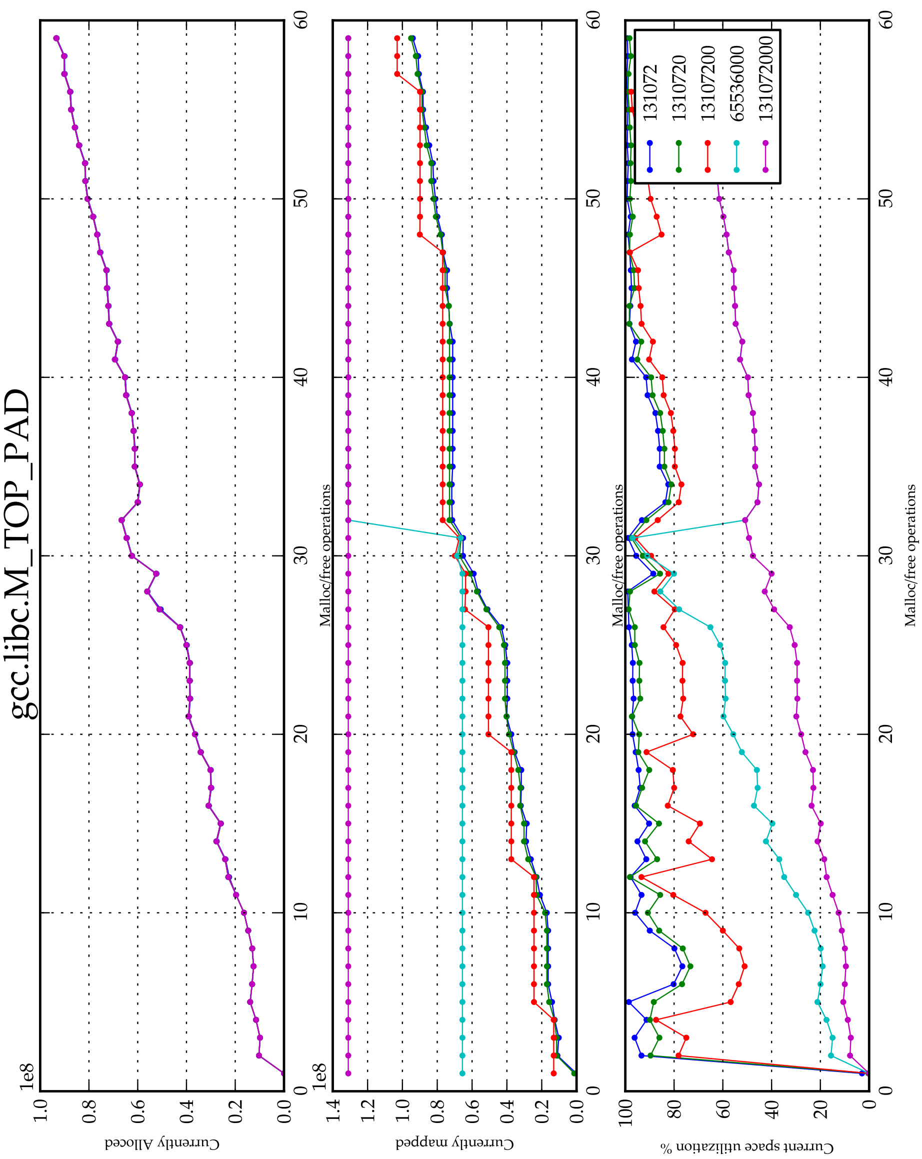
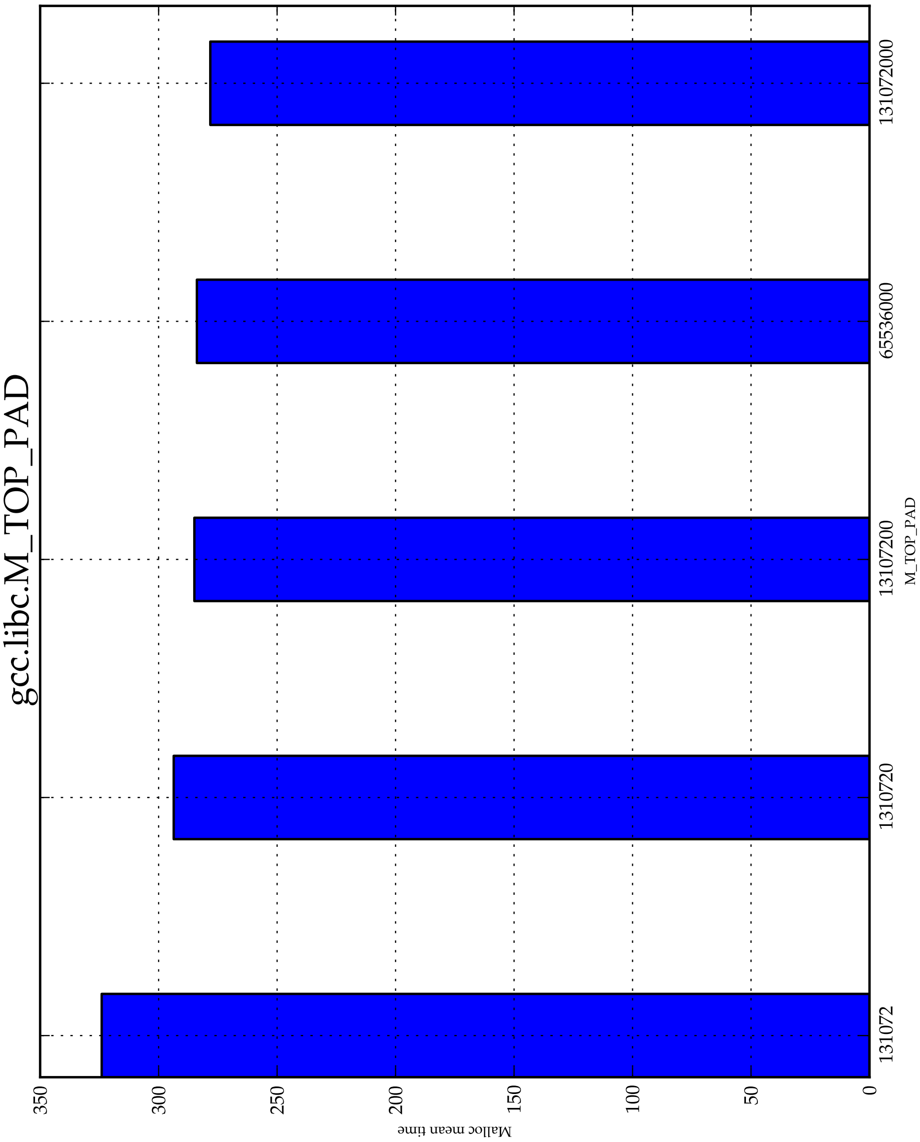


Figure 5.17: gcc plot *c*



The above graph proves our point. Increasing M_TOP_PAD leads to:

- *Less objects allocated via mmap,since the heap is less often short of free space*

- *No significant changes in total fragmentation, showing that glibc packs chunks efficiently even when there is surplus of free space.*

- *Less calls to sbrk since the line showing currently mapped space has less steps.*

- *Lowering of space utilization owing to more preallocated space.*

- *A downward trend of cycles spend in malloc, due to less system calls.*

The best value for this workload seems to be 13107200 after which space utilization dramatically worsens.

5.4.2 M_MAP_THRESHOLD

In our next case we tweak M_MAP_THRESHOLD. When the glibc allocator is out of free space, if the request's size is above M_MAP_THRESHOLD it uses mmap instead of the heap. Mmaps are more expensive than sbrk and the fact that they must be page aligned can lead to waste of space, but they offer the advantage that they can be released back to the system upon freeing of the respective chunk without leaving empty areas.

Figure 5.18: gcc plot *a*

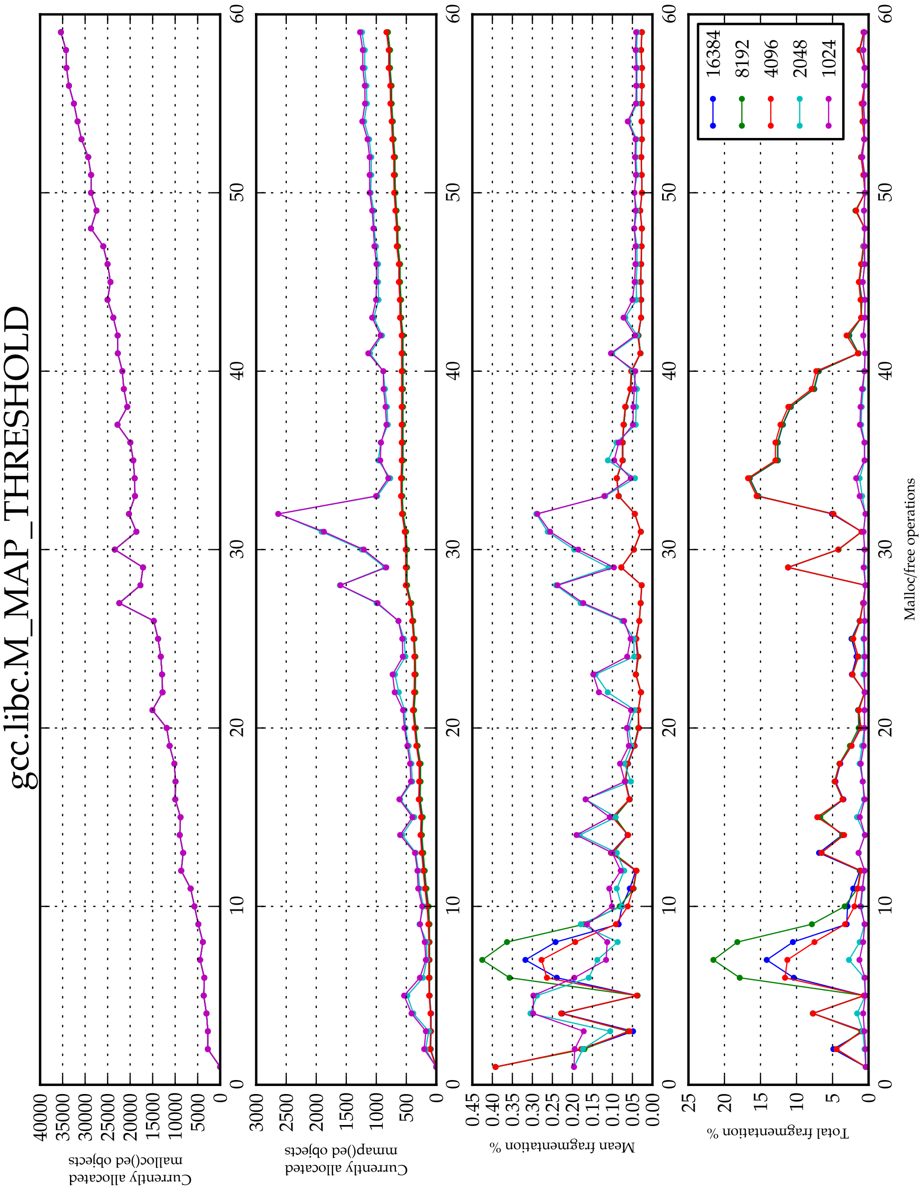


Figure 5.19: gcc plot b

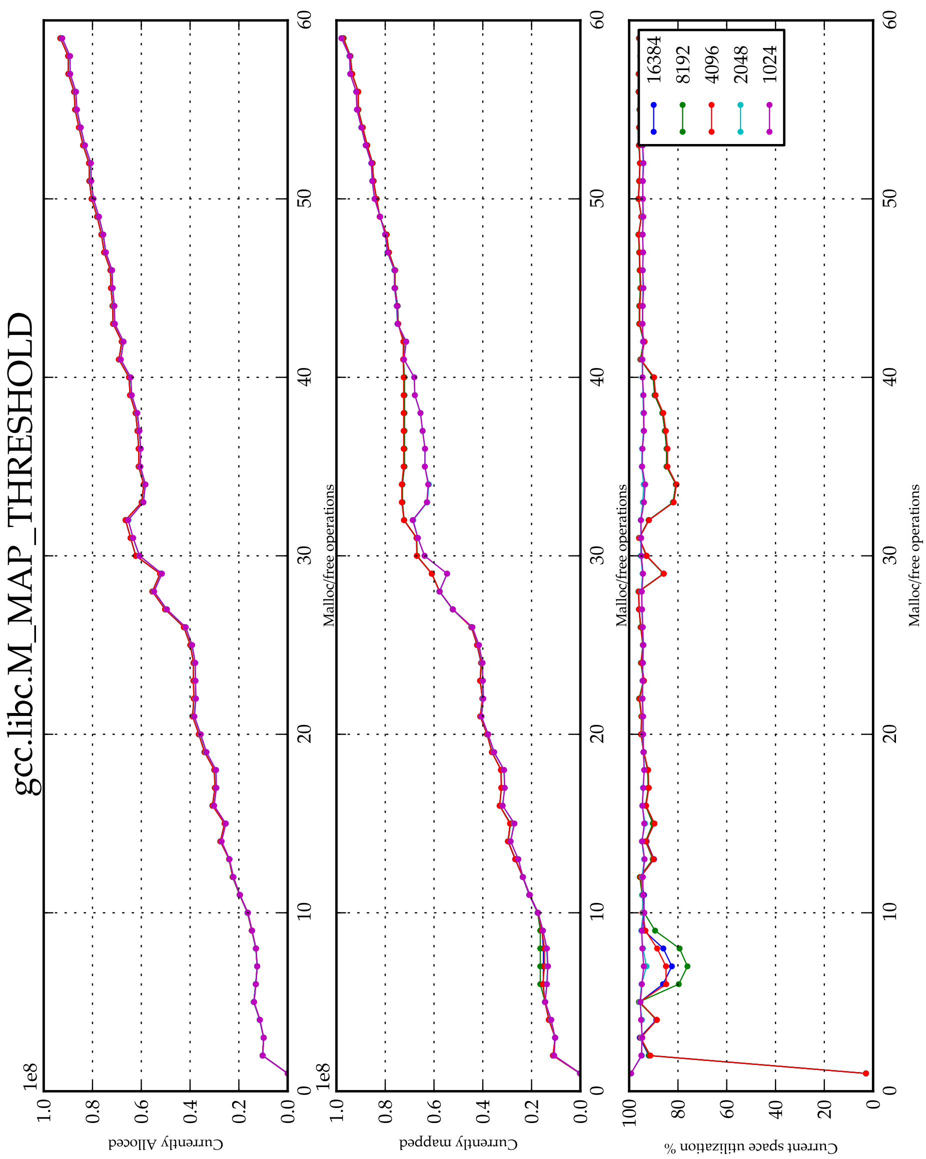
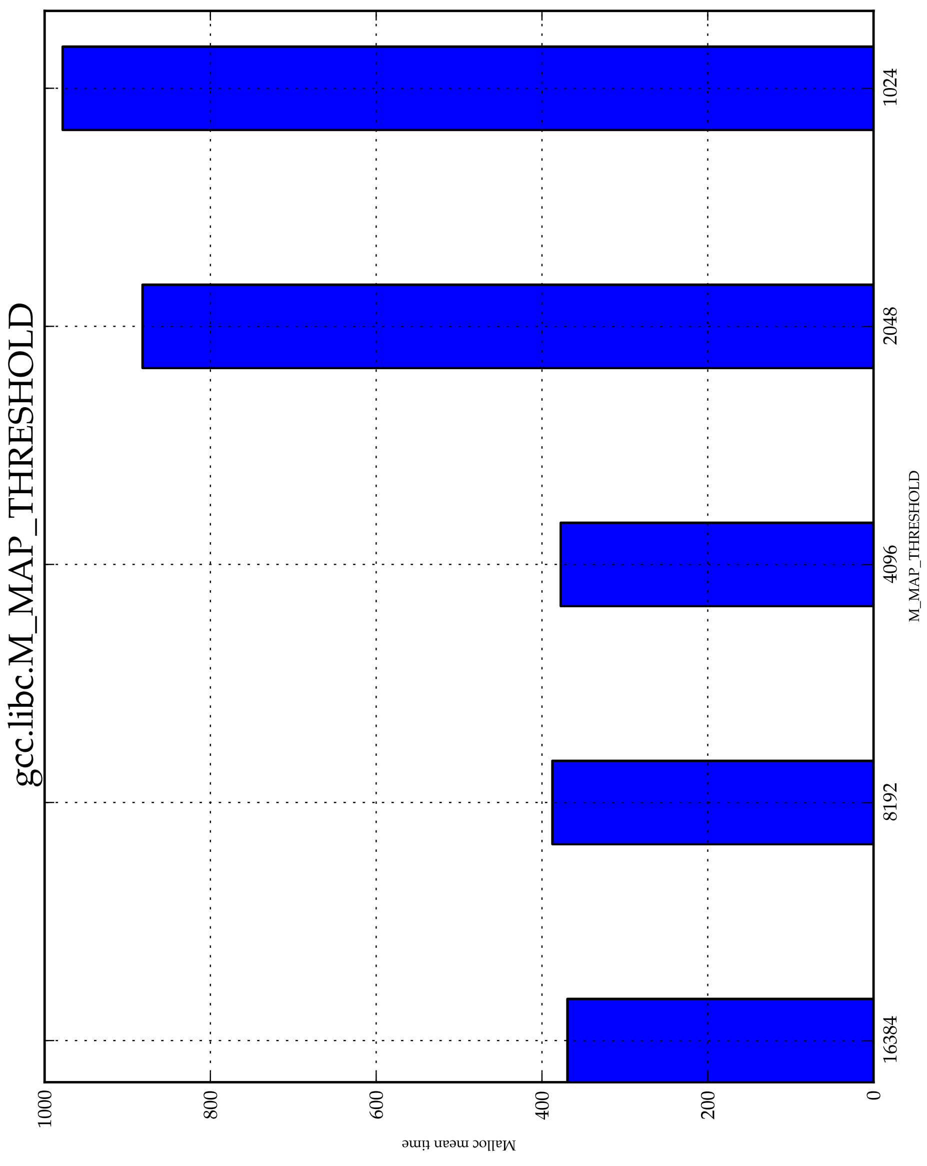


Figure 5.20: gcc plot *c*



Let us observe the two apexes around point 30. These apexes represent malloc "storms", that is, brief moments were more memory is required and then freed.

Lowering M_MAP_THRESHOLD leads to these requests rising above the threshold and thus an increase and subsequent decrease of mapped areas in parallel with the storms. If these requests fall bellow the threshold , after the respective objects have been freed they will leave a hole in the heap(where they had been allocated), an event which is represented in the graphs by two spikes in the fragmentation graph lagging behind the original storms. In contrast when the threshold is sufficiently low these objects would leave the heap fragmentation unaffected and the respective mapped regions would be fully released back to the system. Note that ,in the case of heap allocation ,the heap doesn't shrink back to its original size, owing to the fact that the heap can only shrink from the top plus the conservative trimming policy. This is represented in both currently mapped space and space utilization subplots. We can also clearly see how mmap call counts affect mean malloc cycles, so whatever spatial gains we had are obviously negated.

5.4.3 SYSALLOC_SZ

By the same process of thought we attempted to experiment with dmmlib by changing SYSALLOC_SZ which is the equivalent of glibc's M_TOP_PAD but for mmapped regions instead.

Figure 5.21: gcc plot *a*

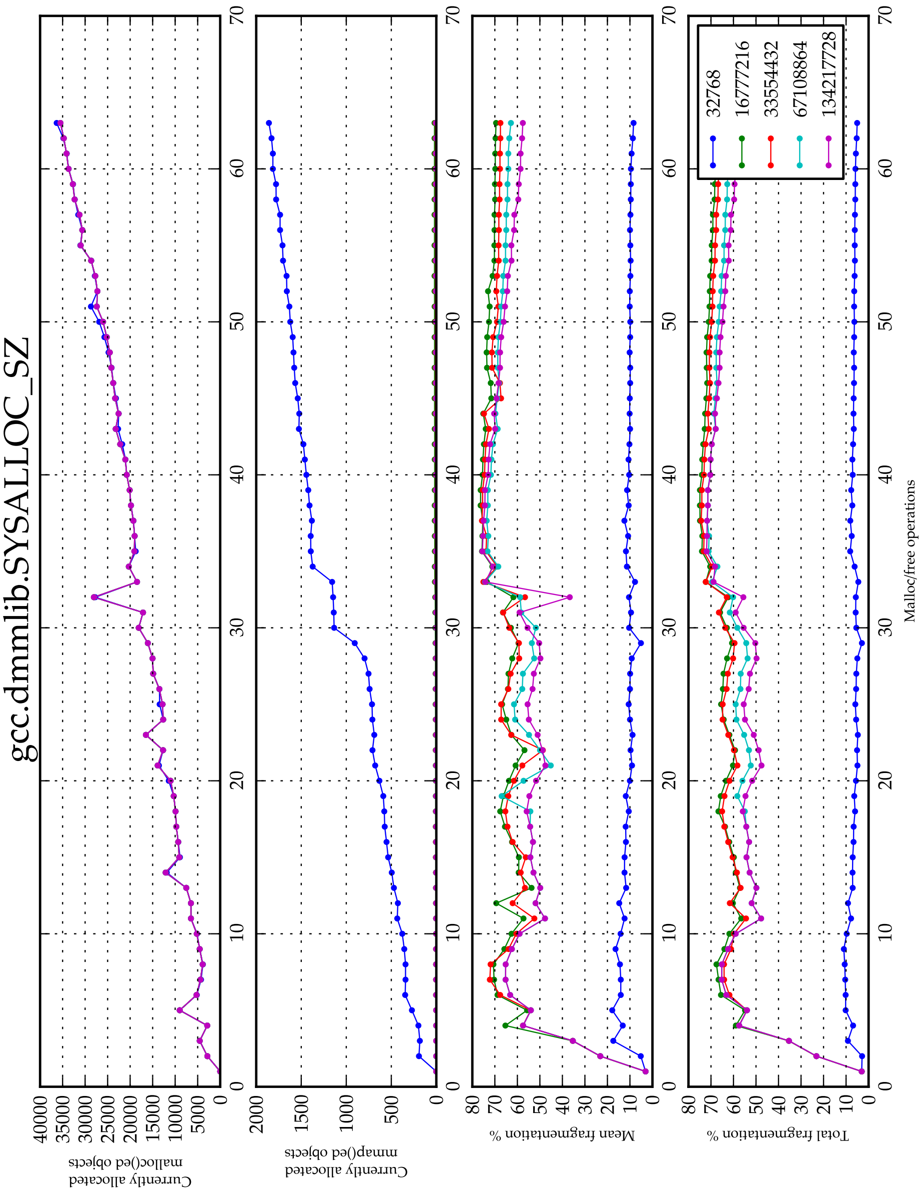
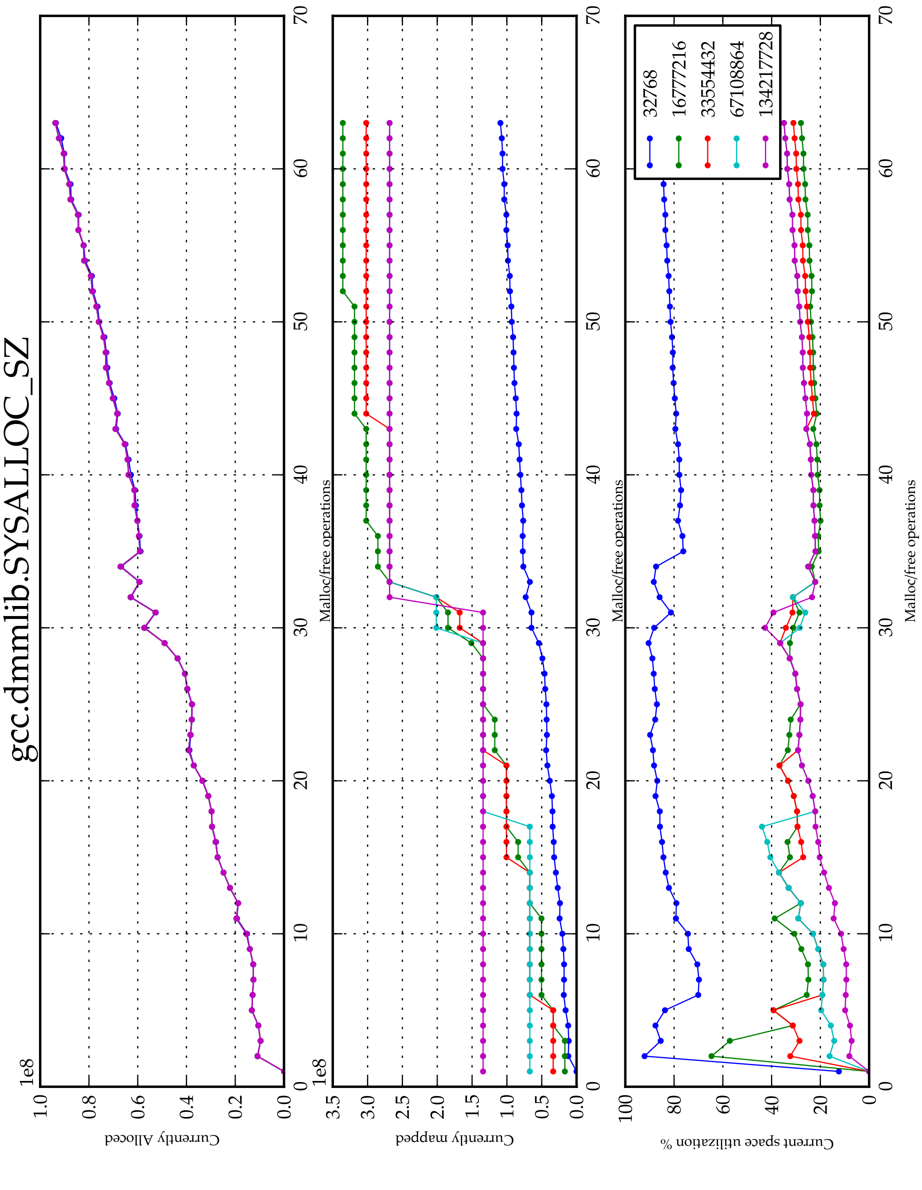


Figure 5.22: gcc plot *b*



Similarly to the former case mmap operations are decreased when increasing SYSALLOC_SZ. However there is a dramatic decrease of space utilization that can't *be explained by the waste inherent in pre-allocating. We conclude that possibly we exposed a big blocks specific bug in that allocator.*

5.5 Summary

Unremarkable spatial data across many workloads. Questionable accuracy of temporal data. Utilizable insights in workload specific glibc allocator tuning .

Chapter 6

Conclusion

6.1 Summary

We used dynamic binary instrumentation to measure the performance of both libc and dmmlib heap managers using pin from Intel. We successfully derived spatial metrics as well as temporal metrics (the later of questionable accuracy). We tested our methodology on select benchmarks from the parsec and spec CPU 2006 suites. We extracted the data using numpy and plotted them using matplotlib.

6.2 Future Additions

Most importantly we must validate the accuracy of our spatial data by comparing it to the output of built in statistics methods that memory allocators expose (glibc provides such an interface). We should explore the use of kernel performance counters instead of the naive timestamp counter. The kernel performance counters would allow us to exclude kernel mode time from our calculations as well as noise from other idle processes, at the cost however of additional overhead and thus noise caused by calling kernel facilities.Interfacing with an existing cache simulator to obtain simulated cache statistics would be an interesting direction and mostly trivial,however at a huge performance cost. We could derive more specific spatial metrics to properly distinguish between space overhead and empty space (fragmentation) which are both computed currently as a single fragmentation metric. The current code base contains checks for the currently instrumented heap manager and should preferably be refactored to be manager agnostic. Lastly we should investigate in improving the currently unused memory access patterning to overcome the threading related locking overhead by using lockless data structures.

Bibliography

[1] Chi-Keung Luk et al. "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation". In: *ACM SIGPLAN conference on Programming language design and implementation*. ACM, 2005, pp. 120–190.

[2] David Ascher et al. *Numerical Python*. UCRL-MA-128569. Lawrence Livermore National Laboratory. Livermore, CA, 1999.

[3] Christian Bienia. "Benchmarking Modern Multiprocessors". PhD thesis. Princeton University, Jan. 2011.

[4] Paul F. Dubois. "Extending Python with Fortran". In: *Computing Science and Engineering* 1.5 (Sept. 1999), pp. 66–73.

[5] Paul F. Dubois, Konrad Hinsen, and James Hugunin. "Numerical Python". In: *Computers in Physics* 10.3 (May 1996).

[6] *Glibc source code*. 2013. URL: http://sourceware.org/git/?p=glibc.git;a=summary.

[7] J. D. Hunter. "Matplotlib: A 2D graphics environment". In: *Computing In Science & Engineering* 9.3 (2007), pp. 90–95.

[8] *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3*. Intel. Santa Clara, CA, USA, Aug. 2012. Chap. 17.13 Time Stamp Counter.

[9] Ioannis Koutras. *dmmlib*. 2013. URL: http://dmmlib.microlab.ntua.gr/.

[10] *Linux man pages online*. 2013. URL: http://man7.org/linux/man-pages/index.html.

[11] Travis E. Oliphant. *Guide to NumPy*. Provo, UT, Mar. 2006. URL: http://www.tramy.us/.

[12] *SPEC CPU2006*. 2013. URL: http://www.spec.org.

[13] G. Van Rossum. *The Python Language Reference Manual*. Network Theory Ltd., Sept. 1, 2003. ISBN: 0954161785. URL: http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20%5C&path=ASIN/0954161785.

[14] Steven Cameron Woo et al. "The SPLASH-2 programs: Characterization and methodological considerations". In: *INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE*. ACM, 1995, pp. 24–36.