

David Gay, Rob Ennals and Eric Brewer

Safe Manual Memory Management

Presented by Andreas Unterweger

Concurrency and Memory Management Seminar, winter term 2010

University of Salzburg, Department of Computer Science

Overview

- Use and issues of manual MM
- HeapSafe description
- HeapSafe implementation
- HeapSafe performance
- Open issues
- Conclusion

Issues of manual MM

[1]

- Memory leaks
 - Allocated memory is not freed anymore
 - Increases memory consumption
- Double frees
 - Free is called more than once for the same pointer
 - May lead to unexpected behaviour of MM
- Dangling pointers
 - Pointer to an object which has already been freed
 - May cause access violations

Why manual MM?

- Garbage collection may have side effects
 - Changes of runtime behaviour of the program
 - Once modified, the program will not work without a garbage collector anymore
 - Memory leaks may occur
- Proposed tool HeapSafe helps to make manual memory management safe(r)

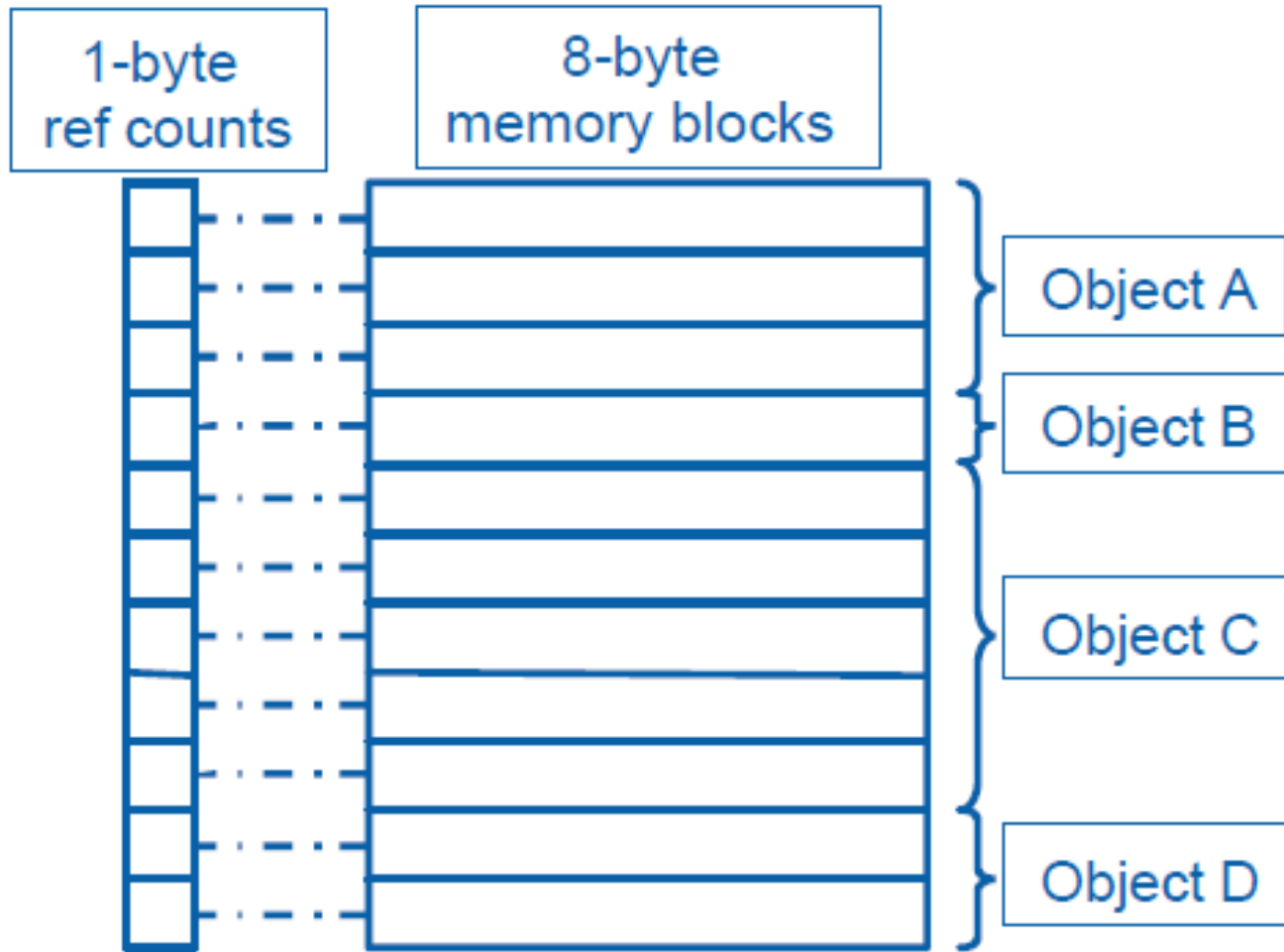
What is HeapSafe?

- A C-to-C translator allowing for safe(r) MM
- Helps to check correctness of malloc/free calls
- Implements reference counting to check for dangling pointers on free calls (logs if there are dangling references after the free call)
- Extends the malloc/free API by extra tools to avoid reporting false positives
- Requires some changes in the program in order to work properly (requires type safety!)

Reference counting I

- Reference count table for every 8 byte block of available memory
- Updated on every pointer write operation
- Additional byte for counter necessary (false negatives on counts which are 0 mod. 256)
- Relies on lazy page allocation of Linux (reference count table in 32 bit environments would yield a 512 MB table if fully used)

Reference count table example



Reference counting II

- Changes necessary
 - Memory has to be zeroed after allocation
 - Local variables which are pointers have to be zeroed (global pointers are zeroed by default)
 - Update pointers within structures and unions using additional functions provided by the programmer
 - Type-aware versions of memset, memcpy etc.
- Local variables (pointers) are tracked using deferred reference counting (shadow stack)

Reference counting example

Code/Description	RC for 0x1000	RC for 0x1008
Initial state	-	-
<code>char *p = malloc(16);</code>	1 (assuming p = 0x1000)	0
<code>char *q = p;</code>	2	0
<code>p += 8;</code>	1	1
<code>p = NULL;</code>	1	0
<code>free(q);</code>	0	0

- Optimization: assume pointer operations leave pointer within the same object → check sum of reference counts of object (all bytes) for zero

Delayed free scopes

- Deal with temporary dangling pointers
- Delayed free scope: actual reference count checking is done at end of scope, not at free call within the scope → may hide double frees
- Nested scopes: scope absorption (delay reference check until the outermost scope)
- Free calls outside of any scope are checked immediately (and reported if necessary)


Dangling pointer example

```
void free_cyclic_list(struct cyclist *start)
{
    struct cyclist *next, *cur = start;
    do
    {
        next = cur->next;
        free(cur);
        cur = next;
    } while (cur != start);
}
```

- Pointer "start" dangles until the end of the loop

Delayed free scopes example

```
void free_cyclic_list(struct cyclist *start)
{
    struct cyclist *next, *cur = start;
    delayed_free_start();
    do
    {
        next = cur->next;
        free(cur); Delayed!
        cur = next;
    } while (cur != start);
    delayed_free_end();
}
```

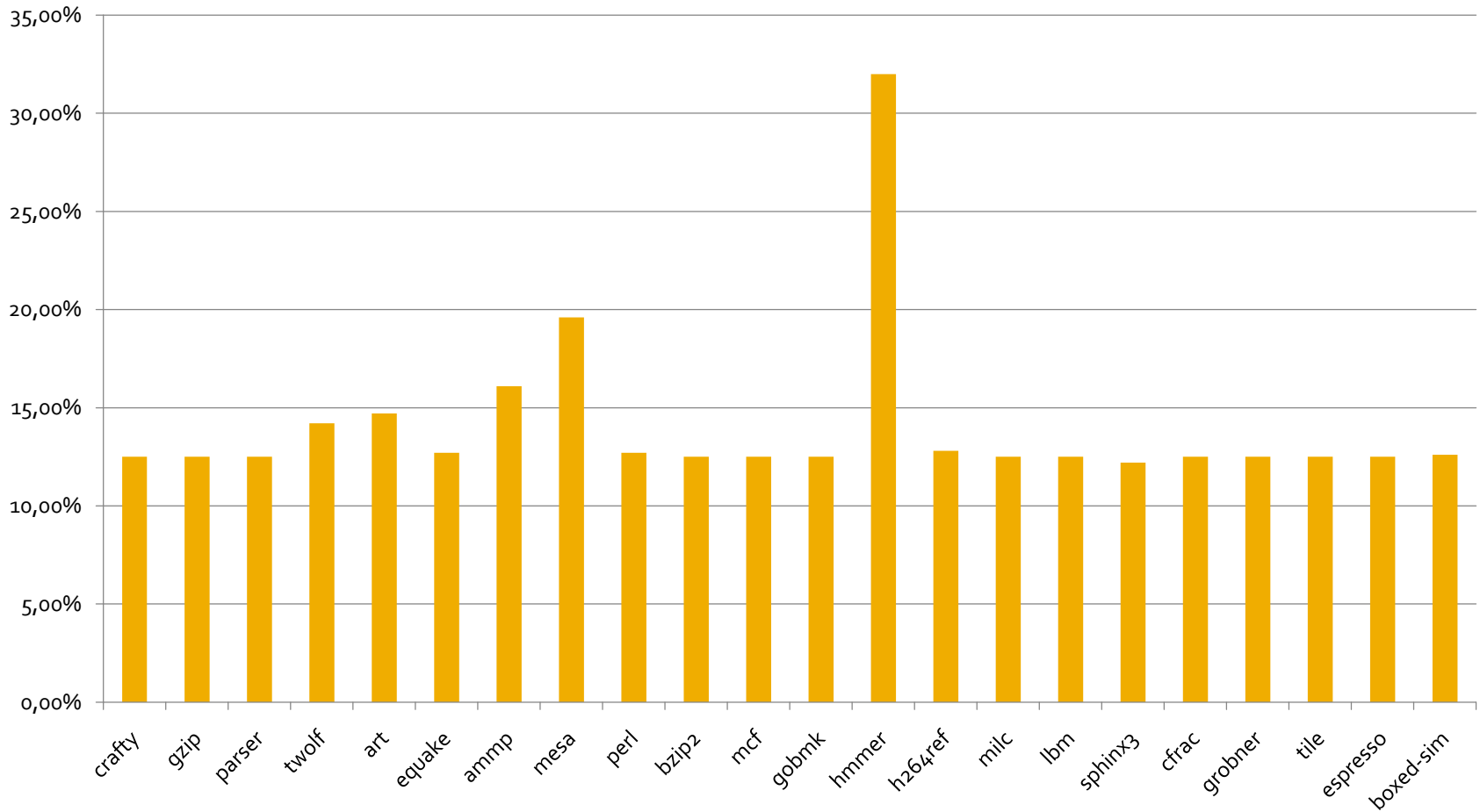


Delayed free scope

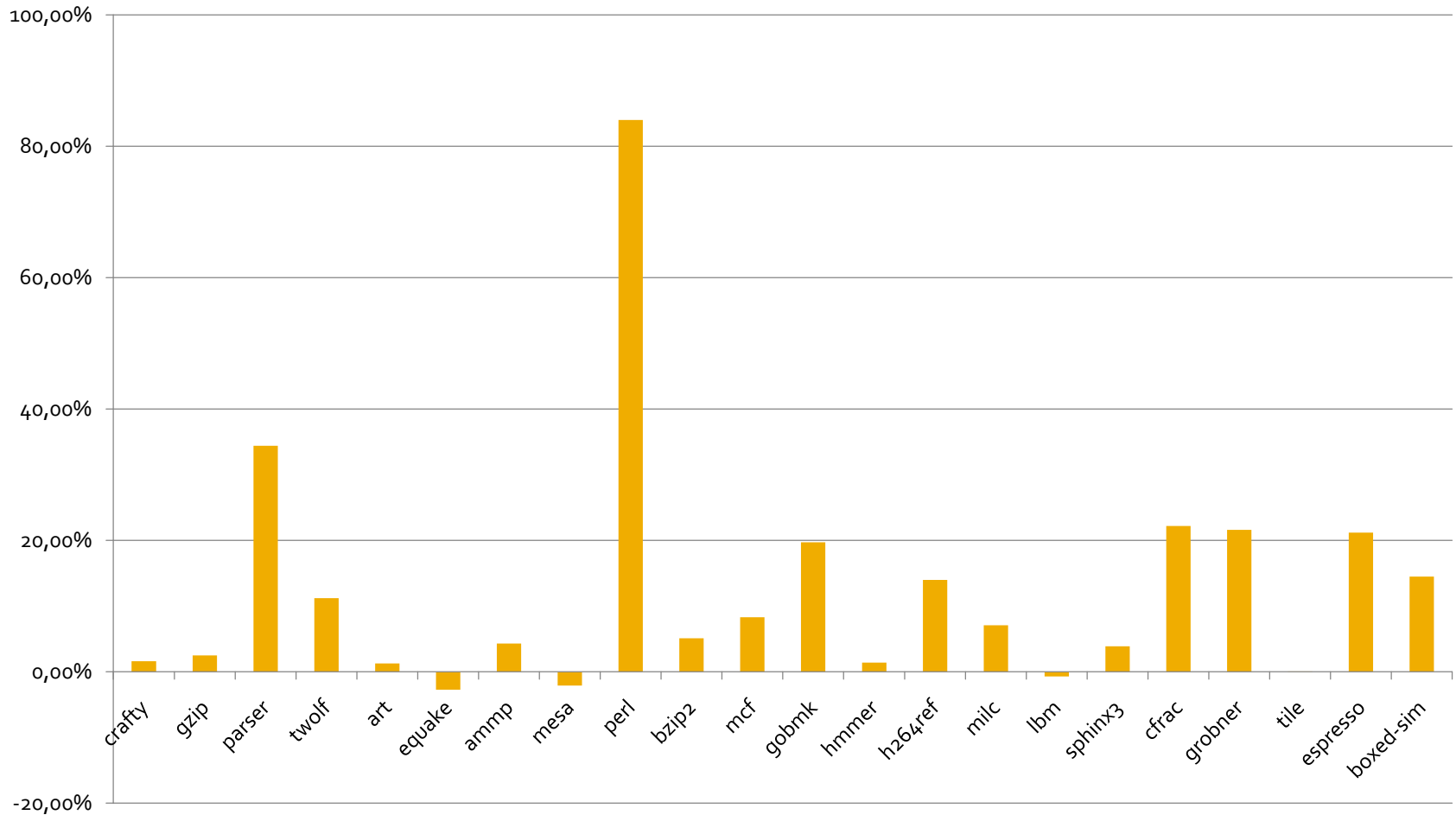
Practical use of HeapSafe

- Modify Makefile so that it calls HeapSafe instead of gcc or any other C compiler
- Build the project and correct code according to HeapSafe's warnings (type safety etc.)
- Remove any type casts and add type functions
- Run the program with test input and check the output for logged dangling references
- Correct the code so that there are no dangling references anymore (add scopes if necessary)

Benchmarks I (space overhead)



Benchmarks II (runtime overhead)



Sources of overhead

- Time overhead
 - Zeroing allocated memory
 - Reference count updating and checking
 - Deferred reference check (shadow stack)
 - Delayed freeing using delayed free scopes
- Space overhead
 - Reference counts (table) and delayed free scopes
 - Shadow stack
- Compile-time overhead and code changes

Open issues

- Implementation not usable in multi-threaded environments
 - Reference counts are not updated atomically
 - Pointer writes are not atomic
 - Deferred references from other threads are not taken into account
- Increased memory consumption
- Increased runtime overhead
- More checks to deal with memory leaks etc.

Conclusion

- HeapSafe can help to find most cases of dangling pointers and double frees in single-threaded applications
- HeapSafe cannot detect memory leaks in its current version, but could be extended
- Other tools (valgrind etc.) may help to deal with memory leaks and undetected errors
- HeapSafe can be used in production code instead of being a debugging-only tool

References

- [0] Gay, D., Ennals, R. and Brewer, E., *Safe Manual Memory Management*. In Proceedings of the 6th international symposium on Memory management (ISMM '07), pp. 2-14, 2007.
- [1] Scott Michael L., *Programming Language Pragmatics (Third Edition)*. Morgan Kaufmann Publishers, San Francisco, 2009.

Thank you for your attention!

Questions?