Optimizing C Multithreaded Memory Management Using Thread-Local Storage

Yair Sade[1], Mooly Sagiv[2], and Ran Shaham[3]

[1] Tel-Aviv University
{sadeyair, msagiv}@post.tau.ac.il *
[2] Tel-Aviv University
[3] IBM Haifa Laboratories
rans@il.ibm.com

Abstract. Dynamic memory management in C programs can be rather costly. Multithreading introduces additional synchronization overhead of C memory management functions (`malloc`, `free`). In order to reduce this overhead, we extended Hoard — a state of the art memory allocator with the ability to allocate thread-local storage. Experimental results using the tool show runtime saving of up to 44% for a set of memory management benchmarks.

To allow transparent usage of thread-local storage, we develop a compile-time algorithm, which conservatively detects allocation sites that can be replaced by thread-local allocations. Our static analysis is sound, i.e., every detected thread-local storage is indeed so, although we may fail to identify opportunities for allocating thread-local storage. Technically, we reduce the problem of estimating thread-local storage to the problem of escape analysis and provide an efficient escape analysis for C. We solve the problem of escape analysis for C using existing points-to analysis algorithms. Our solution is parameterized by the points-to information. We empirically evaluated the solution with two different methods for computing points-to information. The usage of scalable points-to analysis algorithms and the fact that our reduction is efficient, guarantees that our static analysis technique is scalable.

1 Introduction

This paper addresses the problem of reducing the overhead of memory management functions in multithreaded C applications by combining efficient allocation libraries with compile-time static pointer analysis techniques.

1.1 Multithreaded Memory Management Performance

Memory allocation in C programs can be costly in general; multithreading functions add additional complexity. Memory management implementations in C usually consist of a global heap. The `malloc` function acquires a memory block from the global

* Supported in part by a grant from the Israeli Academy of Science.

R. Bodik (Ed.): CC 2005, LNCS 3443, pp. 137–155, 2005.

heap and the `free` function returns the memory block into the heap. The global heap data-structure is shared among the process threads. In order to protect this shared data-structure from concurrent accesses and race conditions, accesses are synchronized by locking primitives such as mutexes or critical-sections. This synchronization may degrade performance due to the following reasons: (i) on multithreaded environments, threads that call memory management functions concurrently are blocked by the locks; (ii) once a thread is blocked, an expensive context-switch occurs; (iii) the lock primitives can have an overhead even if no block occurs.

On SMP machines the problem can become acute and cause an application performance bottleneck. It can happen when threads that are executed on different processors call memory management functions concurrently. Those threads are blocked by the lock primitives and the blocked processors become unutilized. This reduces the application parallelism and may reduce throughput.

1.2 Existing Solutions

There are two main approaches for improving the performance of memory management routines in multithreaded applications: (i) runtime solutions, and (ii) programmable solutions. Runtime solutions usually provide an alternative multithreaded-efficient memory management implementation [8, 11, 21]. In programmable solutions, the programmer develops or exploits application-specific custom allocators, for example memory-pools or thread-local arenas as in [1].

Runtime approaches only mitigate performance degradation — even the most efficient memory management implementations have a synchronization overhead. In programmable approaches, the programmer has to design the application to work with a custom allocator, which is not an easy task on large-scale applications, and almost impossible on existing systems. Moreover, programmable solutions are error prone and might cause new bugs. Finally, in [9] it is shown that in most cases custom allocators do not improve the performance of the applications at all.

1.3 Our Solution

Thread-Local Storage Allocator. We extended Hoard — a state of the art memory allocator [8] by adding the ability to allocate *thread-local storage*. Thread-local storage is a memory location, which is allocated and freed by a single thread. Therefore, there is no need to synchronize allocation and deallocation of thread-local storage. Specifically, we enhance the memory management functions with two new functions, `tls_malloc` and `tls_free`. The `tls_malloc` function acquires storage from a *thread-local heap*, and the `tls_free` function deallocates storage acquired in a thread-local heap. Both functions manipulate the thread-local heap with no synchronization.

Additional benefit of thread-local storage is better utilization of the processor's cache. Modern processors maintain a cache of the recently used memory. The processor's cache saves accesses to the memory which are relatively expensive operations. When a thread is mainly executed on the same processor, the locality of the thread-local storage allocations improves the processor's cache utilization.

Statically Estimating Thread-Local Storage. Employing thread-local storage by programmers in a language like C is far from trivial. The main difficulty is deciding whether an allocation statement can be replaced by `tls_malloc`. Pointers into shared data can be accessed by multiple threads, thus complicating the task of correctly identifying thread-local storage. Therefore, in this paper, we develop automatic techniques for conservatively estimating thread-local storage. This means that our algorithm may fail to identify certain opportunities for using `tls_malloc`. However, storage detected as `tls_malloc` is guaranteed to be allocated and freed by the same thread. Thus, our solution is fully automatic.

The analysis conservatively detects whether each allocation site can be replaced by `tls_malloc`. This is actually checked by requiring that every location allocated by this statement cannot be accessed by other threads. In particular, it guarantees that all deallocations are performed in the code of this thread. Therefore, our algorithms may be seen as a special case of escape analysis, since thread-local storage identified by our algorithm may not escape its allocating thread. We are unaware of any other escape analysis for C.

Our analysis scales to large code bases. Scalability is achieved by developing a flow- and context-insensitive algorithm. Furthermore, our algorithm performs simple queries on a points-to graph in order to determine which allocation site may be accessed by other threads. Thus, existing flow-insensitive points-to analysis algorithms [7, 27, 30, 14, 15, 18] can be exploited by our analysis. This also simplifies the implementation of our method.

Empirical Evaluation. We have fully implemented our algorithm to handle arbitrary ANSI C programs. Our algorithm is parameterized by the points-to information. Thus the precision of the analysis is directly affected by the precision of the points-to graph. In particular, the way structure fields are handled can affect precision. We therefore integrated our algorithm with two points-to analysis algorithms: Heinze's algorithm [18], which handles fields very conservatively, and GrammaTech's CodeSurfer points-to algorithm [30]. CodeSurfer handles fields in a more precise manner.

We have tested our implementation on a set of 7 memory management benchmarks used by Hoard and other high performance allocators. We verified that on the memory management benchmarks our static analysis precisely determines all opportunities for use of `tls_malloc` instead of `malloc`.

The standard memory management benchmarks are somewhat artificial. Thus, we also applied our static algorithm to work with a multithreaded application that uses `Zlib` [5], the popular compression library. Finally, we applied our algorithm on `OpenSSL-mttest` which is a multithreaded test of the `OpenSSL` cryptographic library [4].

For 3 of the memory management benchmarks, there are no opportunities for replacing `malloc` with `tls_malloc`. On the other 4 memory management benchmarks, we achieve up to 44% speedup due to use of thread-local storage. This is encouraging, given that Hoard is highly optimized for speed.

For the `Zlib` library, our static algorithm detected the opportunities for using `tls_malloc` instead of `malloc`. We achieved a speedup of up to 20% over Hoard

by using the thread-local storage allocator. This result shows the potential use of our methods on more realistic applications.

On `OpenSSL-mttest` our static algorithm fails to detect opportunities for thread-local storage. However, inspecting the runtime behavior of this benchmark, we find that only a negligible amount of the allocated memory during the run is actually thread-local. Therefore, even an identification of some thread-local storage for this benchmark is not expected to yield any performance benefits. Nevertheless, the application of our algorithm on `OpenSSL` demonstrates the scalability of our tool for handling large code bases.

1.4 Related Work

Static Analysis. We reduce the problem of thread-local storage detection to the problem of escape analysis for C. In this paper we developed an escape analysis algorithm that uses an existing points-to algorithm. Our analysis uses points-to information generated by any flow-insensitive points-to analysis. Flow-sensitive points-to algorithms are more precise but less suitable for multithreaded programs, due to the thread interaction that needs to be considered. Our analysis can use either context-sensitive or context-insensitive points-to analysis algorithms.

There are two commonly used techniques for performing flow-insensitive points-to analysis: (i) unification-based points-to analysis suggested by Steensgaard [27], (ii) inclusion-based points-to analysis suggested by Andersen [7]. Generally, the unification method is more scalable but less precise. The GOLF algorithm [14, 15] is a unification-based implementation with additional precision improvements.

In our prototype, we used the following points-to analysis algorithms: (i) GrammaTech's CodeSurfer [30], (ii) Heintze's points-to analysis [17, 18]. Both algorithms are context-insensitive algorithms based on Andersen's analysis. A specialized flow- and context-sensitive points-to algorithm that is specialized for multithreaded programs, is that of Rugina and Rinard [24]. The algorithm is used for multithreaded programs written in Cilk, an extension of C. Another efficient context-sensitive points-to analysis is Wilson and Lam analysis [29].

In this paper, we study the problem of thread-local storage identification through escape analysis for C programs and the performance benefits obtained through the use of thread-local storage. Escape analysis for Java has been studied extensively [13, 10, 12, 6, 23, 26] and was employed for thread-local storage in [28]. We note that there are several differences between C and Java, which make our task non-trivial. First, in contrast to Java, C programs may include unsafe casting, pointers into the stack, multilevel pointers, and pointer arithmetic. These features complicate the task of developing sound and useful static analysis algorithms for C programs. Second, explicit memory management is supported in C, whereas Java employs automatic memory management, usually through a garbage collection mechanism. In [11] Boehm observes that a garbage collector may incur less synchronization overhead than in explicit memory management. This is due to the fact that many objects can be deallocated in the same GC cycle, while explicit memory management requires synchronization for every `free`. Our thread-local storage allocator reduces the above synchronization overhead by providing synchronization-free memory management constructs.

Of course it should be noted that our analysis for C is made simpler since it does not need to consider Java aspects such as inheritance, virtual method calls and dynamic thread allocation. The difficulties that rise in escape analysis for C programs are handled by the underlying points-to algorithms. The points-to algorithms for C are rather conservative, but interestingly they provide good empirical results when analyzing the memory management benchmark programs.

In [28] Steensgard describes an algorithm for allocating thread-local storage in Java using the unification-based points-to analysis described in [27]. Our simple static algorithm can use an arbitrary points-to algorithm. Our prototype implementation uses inclusion-based points-to analysis algorithms which are potentially more precise. Indeed, one of the interesting preliminary conclusions from our initial experiments is that in many C programs thread-local storage can be automatically identified despite the fact that C allows more expressive pointer manipulations.

Multithreaded Memory Allocation for C. In [19] Larson studies multithreading support and SMP scalability in memory allocators. Berger's Hoard allocator [8] is an efficient multithreaded allocator. In the paper we extend Hoard to support an efficient thread-local storage allocation. In [21], Maged shows an extension of Hoard with an efficient lock handling based on hardware atomic operations. In [11], Boehm suggests a scalable multithreaded automatic memory management for C programs.

1.5 Contributions

The contributions of this paper can be summarized as follows:

- A new generic and scalable escape analysis algorithm targeted for C. The input to our static algorithm is points-to information obtained by *any* flow-insensitive points-to algorithm.
- Static estimation of thread-local storage allocations.
- Extending an existing allocator with high performance treatment of thread-local storage.
- Empirical evaluation which shows rather precise static analysis algorithms resulting in significant runtime performance improvements.

1.6 Outline of the Rest of This Paper

The remainder of the paper is organized as follows: Section 2 provides an overview of our work. Section 3 describes our thread-local storage allocator. In Section 4 the static analysis algorithm is described. Empirical results are reported in Section 5. Preliminary conclusions and further work are sketched in Section 6.

2 Overview

This section provides an overview of the capabilities of our technique by showing its application to artificial program fragments. These fragments are intended to give a feel of the potential and the limitations of our algorithms.

2.1 Escaped Locations and C Multithreading

C programs consist of three types of memory locations:

Stack Locations. Stack locations are allocated to automatic program variables and by the `alloca` function.
Global Locations. Static and global variables are allocated in global locations.
Heap Locations. Heap locations are the dynamically allocated locations.

Multithreading is not an integral part of the C programming language. In this paper, we follow the `POSIX thread` standard. Inter-thread communication in `pthreads` is performed by `pthread_create` function, which creates a thread and passes an argument to the thread function. The argument may point to memory locations that are accessible by the creator thread. After invoking the `pthread_create` function, these memory locations are also accessible by the new thread. However, our method can also support different thread implementations with other inter-thread communication methods such as *message-passing* or *signals*. Finally, we assume that each thread owns its own stack.

We say that a heap-location *escapes* in a given execution trace when it is accessed by different threads than the one in which it was allocated. A heap-location that does not escape on any execution trace, is accessible only by a single thread. Therefore, it is allocated and freed by the same thread. Hence, the allocation statement can be replaced by `tls_malloc`.

Our static analysis algorithm conservatively estimates whether a location may escape. The estimation is performed by checking the following criteria: (i) global locations as well as locations which may be pointed by global pointers may escape; (ii) locations passed between threads by the operating system's inter-thread-communication functions and locations reachable from these locations may escape. Allocation of locations that do not meet the above criteria are guaranteed to be accessible by a single thread [25], thus are allocated using `tls_malloc`.

Clearly, our algorithm is conservative and may therefore detect a certain location as "may-escape" while there is no program execution in which this location escapes. This may result in missing some opportunities for using thread-local storage.

2.2 Motivating Example

Figure 1 shows a program fragment that uses the `pthread` implementation of threads. This program creates a thread by using the `pthread_create` function, and waits for its termination by using the `pthread_join` function.

Our static algorithm detects the allocation in line 1 as a thread-local storage allocation and replaces it with `tls_malloc`. The location that is pointed by the assigned variable `l` is accessible by a single thread. Specifically, it is allocated and freed by the `foo` thread. In this case, static analysis can trivially detect the latter, since `l` is assigned once. Therefore, this allocation statement can be allocated on the thread-local heap of `foo`. In principle, the free statement in line 2 could be replaced by `tls_free`. However, as explained in Section 3.2 we extend the `free` statement implementation to support the deallocation of thread-local storage with negligible overhead, thus it is also possible to avoid replacements of the `free` statement with `tls_free`.

```
#include <stdio.h>
#include <pthread.h>
char *g;
void foo(void *p){
  char *l;
  1: l = malloc(...); // Is tls_malloc?
  2: free(l);
  3: free(p);
  4: g = malloc(...); // Is tls_malloc?
  5: free(g);
}
int main(int argc, char **argv) {
  char *x, *q;
  pthread_t t;
  6: x = malloc(...); // Is tls_malloc?
  7: q = x;
  8: if (get_input()){
       9: pthread_create(&t, NULL, foo, q);
       10: pthread_join(t, NULL);
  }
  11: else {
        12: free(x);
  }
  13: return 0;
}
```

Fig. 1. A sample C program

One can mistakenly conclude that the `malloc` in line 6 can be replaced by `tls_malloc`. At first glance, it seems that the location allocated in line 6 and freed in line 12 is allocated and freed by the same thread and can therefore be allocated on the thread-local storage. However, if we observe more closely, we can see that in line 7, that location is assigned to the pointer `q`, if the condition in line 8 holds, we execute line 9 on which `q` is passed as a parameter to the thread function `foo`, and then it is finally freed in line 3. Thus, on some executions, the location allocated in line 6 may be freed by a different thread and therefore it *cannot* be allocated on the thread-local storage. Our static algorithm correctly identifies that by observing that `q` is passed as a parameter to another thread, and therefore marks the memory locations that `q` may points-to as accessible by multiple threads. The flow-insensitive points-to analysis tracks the fact that `x` and `p` are aliases to the location that is allocated in line 6. Therefore, that location violates the conditions for thread-local storage allocation. Of course, manually tracking pointer values for complex applications is not a trivial task and it is error prone.

The allocation in line 4 pointed by `g` is allocated and freed by the same thread — the `foo` thread, and can therefore be safely allocated on the thread-local storage and replaced by `tls_malloc`. However, our static algorithm will fail to identify the memory allocated in line 4 as thread-local. This is due to the fact that memory allocated in line 4

is pointed by the global variable `g`, making it accessible by both the `main` and the `foo` threads.

3 Thread-Local Storage Allocator

Our allocator is based on the Hoard [8] allocator which is briefly described in Section 3.1. In Section 3.2 we describe our extensions to allow thread-local storage support in Hoard.

3.1 The Hoard Allocator

Hoard is a scalable memory allocator for multithreaded applications running on multiprocessor machines. It addresses performance issues such as contentions, memory fragmentation, and cache-locality. In particular, it reduces contentions by improving lock implementation and by avoiding global locks. Hoard manages a dedicated heap for each processor. The use of dedicated processor heaps reduces the contention and also improves the processor cache locality.

Hoard maintains two kinds of heaps: (i) a *processor heap* which belongs to a processor, and (ii) a *global heap* which is one heap for the entire process. Each heap is synchronized using locks. The global heap is backed by the operating system memory management routines[4]. The fact that a thread is mostly executed on the same processor helps in synchronization reduction since its processor's heap should be unlocked when it calls the allocator. Contention may occur if the thread is accessing the processor heap from a different processor.

The processor heap and the global heap contain *super-blocks*, where a super-block is a pool of memory blocks of the same size. When a thread attempts to allocate memory, Hoard first tries to acquire it from its thread heap super-blocks, then (if there is no memory available in these blocks), it attempts to allocate a super-block from the global heap and assigns the block to the current processor. As a last resort Hoard attempts to allocate memory from the operating system.

Hoard improves the performance significantly, however, a synchronization contention may still frequently occur for the global heap (and less frequently for the processor heap). Our extensions to Hoard reduce these kinds of synchronization contention.

3.2 Hoard Extensions

We extend Hoard to allow support for thread-local heaps. In particular, we enhance the memory management functions with two new functions, `tls_malloc` and `tls_free`. The `tls_malloc` function acquires storage from the thread-local heap, and the `tls_free` function deallocates storage acquired in a thread-local heap. Both functions manipulate the thread-local heap with no synchronization. In addition, we extend the `free` statement implementation to deallocate thread-local storage. This extension is made to allow a `free` statement to deallocate memory allocated both by a `malloc` statement and a `tls_malloc` statement.

[4] Actually, Hoard uses dlmalloc [2] implementation instead of the standard operating system memory management routines.

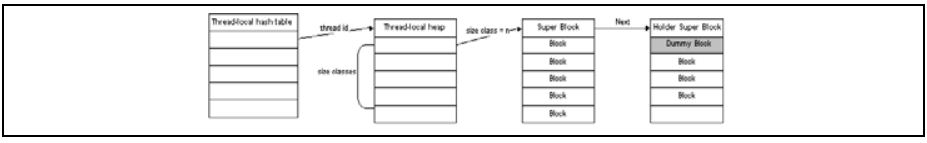

Fig. 2. Thread-local heaps layout

Thread-Local Heaps Implementation. In principle, we could have used the POSIX thread specific functions (`pthread_setspecifc, pthread_getspecific`) to allow a thread to access its corresponding thread-local heap. These functions, however, have performance cost as Boehm shows [11]. Thus, Boehm provides a more efficient implementation to allow a thread to access its corresponding thread-specific information. However, Boehm's implementation assumes a garbage-collector environment. We therefore develop a similar implementation for the case of an explicit allocator environment.

We maintain the thread-local heaps in a hash table (denoted further by *thread-local hash table*) as shown in Figure 2. We normalize the unique thread id and use it as a key for that table. Our implementation uses the value for a key as a pointer to a thread-local heap ; thus, a thread accesses its thread-local heap by fetching the value in the hash table entry corresponding to its thread id.

Our implementation assumes the following simplifying assumptions: (i) the number of thread-local heaps is fixed. A thread may thus fail to obtain a thread-local heap. In this case, memory is allocated using `malloc`. Our implementation sets the number of thread-local heaps to 2048. We expect most programs to have a smaller number of threads. (ii) we assume a 1-1 mapping between a thread id and an entry index in the thread-local hash table. This assumption holds for the Linux `pthreads` implementation.

Creation and maintenance of thread-local heaps require some overhead. We therefore create such heaps only upon the first `tls_malloc` request. Thread-local heaps are not backed by the Hoard global heap, but directly by the operating-system heap. We do not use the global heap for simplicity reasons, and because it does not affect performance on the benchmarks we tried. Synchronization is required only when the thread-local heap aqcuires/frees memory from the operating system.

In order to avoid trashing of allocations and deallocations of blocks from the operating-system, we guarantee that a thread-local heap always maintains one super-block for each size class. We call this super-block a *holder super-block*. The latter is enabled by allocation of a *dummy block*, which prevents the deallocation of this holder super-block. The dummy block is freed only when the holder super-block becomes full. Using this method we can help applications that frequently allocate and deallocate small blocks. Upon thread termination, we clean up the thread-local heap, as well as its holder super-blocks.

tls_malloc. Figure 3 shows a pseudo-code of the `tls_malloc` implementation. In order to allow fast access to a thread-local heap, we maintain thread-local heaps in a hash table. Thus, `tls_malloc` first searches the hash table for the thread-local heap corresponding to the allocating thread. We make an optimization, and allocate a thread-

```
tls_malloc(size)
 if no thread-local heap in the hash-table for the current thread
 exists then
   create thread-local heap and store in the hash-table
 if no available super-block exists in thread-local heap then
   if no holder super-block exists then
     allocate a super-block from OS and mark as thread-local
     set super-block as a holder super-block
     allocate dummy block from holder super-block
     insert to thread-local heap super-block list
   else
     free the dummy block back to the super-block
     set holder super-block as regular super-block
 return block from super-block
```

Fig. 3. Pseudo-Code for tls_malloc

local heap only upon the first `tls_malloc` request occurring in a thread; thus, threads that do not make `tls_malloc` requests are not affected.

Next, the `tls_malloc` routine looks at the thread-local heap super-blocks list in order to find a suitable super-block for the allocation. As in Hoard, each heap has super-blocks of various allocation sizes. In case a super-block is not found, we check whether a holder super-block exists, and allocate one if necessary. As mentioned earlier, the holder super-blocks are used to reduce the number of the operating system's memory management functions calls.

Once we mark the super-block as thread-local, we save this super-block in our thread-local hash table. The next step is acquiring a dummy block from the holder super-block, that will prevent deallocation of the latter, even when it becomes empty. The last step is adding this holder super-block as a part of our super-blocks list. In case we have an allocated holder super-block that is out of free blocks, we free our dummy allocated block and transform the holder super-block to a regular super block. Once we have a super-block we return a block from it to the caller.

tls_free. Freeing memory is performed by the `tls_free` function. The function returns the block to its super-block and frees the super-block in case it becomes empty. As already mentioned all the thread-local heap manipulations are performed without synchronization since only a single thread accesses the heap data.

The `tls_free` complements the `tls_malloc` operation, and the programmer invokes it to free thread-local storage objects. In addition, we extend the `free` statement implementation to deallocate thread-local storage. This extension is made to allow a `free` statement to deallocate memory allocated both by a `malloc` statement and a `tls_malloc` statement. In particular, when a block is freed using the `free` function, our allocator first checks whether the allocated block is from the thread-local heap. This information was stored in the super-block of the block. Once we determine that the allocated block is thread-local block we will free it appropriately.

4 Statically Identifying Thread-Local Storage

In this section, we describe our static algorithm for estimating allocation sites that can be replaced by thread-local storage allocation. Our analysis conservatively detects whether each allocation site can be replaced by `tls_malloc`. In particular, it guarantees that all deallocations are performed in the code of this thread.

We reduce the problem of finding thread-local storage to the problem of escape-analysis. In order to determine that an allocation site can be replaced by `tls_malloc`, a static algorithm must ensure that all locations allocated at the allocation site are thread-local storage, i.e., deallocated by the code of the allocating thread. Our algorithm does that by checking stronger property for locations. Our algorithm makes sure that memory locations allocated at an allocation site never *escape* their allocating thread, i.e., all locations allocated at that site are accessed only by the allocating thread in all execution traces. Clearly, locations that do not escape their allocating thread cannot be deallocated by other threads, therefore we conclude that our algorithm indeed yields safe thread-local storage information.

Our algorithm enjoys two characteristics that make it attractive for the "real-world". First, it scales for large code bases. Second, our algorithm is very simple to implement. Scalability is achieved by using flow- and context-insensitive algorithms, based on simple queries on points-to graphs in order to determine allocations sites that do not allocate escaped memory locations. Furthermore, the points-to graph we use may be obtained by applying *as is* any existing flow-insensitive points-to analysis(e.g., [7, 27, 30, 14, 15, 18, 29]). This last fact greatly simplifies the implementation of our algorithm. In fact, we integrated our algorithm with two existing points-to analysis algorithms, as discussed in Section 4.2.

4.1 The Algorithm

Our algorithm partitions the memory locations into two sets, *may-escape* locations and the *non-escaped* locations. A may-escaped location may be accessed by other threads, while a non-escaped location cannot be accessed by threads, other than its allocating thread, on all execution paths. Our algorithm concludes that an allocation site that does not allocate may-escape locations may be replaced by `tls_malloc`.

Our algorithm performs simple queries on a points-to graph generated by a flow-insensitive point-to analysis. This points-to graph is an abstract representation of all memory locations and pointer relations that exist for all program points and for all execution paths. A node in the graph represents an abstract memory location and an edge in that graph represents a points-to relation.

Static analysis of C programs is not trivial. There are difficulties such as casting and pointers arithmetic. These difficulties are tackled during the generation of the points-to graph which is a preceding step to our analysis. Our analysis can simply traverse the graph and bypass the problems of static analysis of C programs.

A pseudo-code of the algorithm for detecting may-escape locations is shown in Figure 4. The algorithm traverses abstract heap locations that represent allocation sites. For each abstract location it performs a query on the points-to graph. The query checks whether the location is pointed by a global abstract location, or whether it is being passed

```
Input: Program points-to flow-insensitive graph
Output: Partition of the locations to may-escape/thread-local
for each abstract heap location l {
  if l is reachable from a global location or
    l is reachable from a thread function argument or
    l is reachable from a location that passed as thread
    function argument
  then
    mark l as may-escape
  else
    mark l as thread-local
}
```

Fig. 4. Thread-local storage detection by an escape analysis for C using points-to information

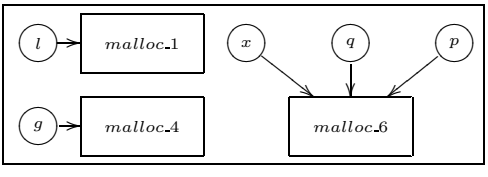

Fig. 5. Flow-insensitive points-to graph for the program shown in Figure 1

as an argument to inter-thread communication functions. Otherwise all runtime locations represented by the abstract heap location, cannot be pointed by any global location or by inter-thread communication function arguments, and thus, the location can be safely allocated using thread-local storage.

We can also detect deallocations of thread-local storage as follows: for each statement of the form `free(x)`, if all the abstract locations which may be pointed by `x` are not may-escape, we can safely replace this statement by `tls_free`. Otherwise we conservatively assume that it may represent a location which is not allocated using thread-local storage. In this case, the runtime implementation checks the status of this location and deallocates it appropriately. Our experience shows that this runtime overhead is marginal. Therefore, we decided not to implement this static optimization and leave `free` statements unchanged.

Let us demonstrate the application of our algorithm by running it on the sample C program shown in Figure 1. In Figure 5 a flow-insensitive points-to graph is shown. The heap abstract location, representing locations allocated by the `malloc` in line 1, is not pointed by any global abstract location nor by inter-thread communication function arguments. Therefore it can be safely allocated on the thread-local heap. The heap abstract location, representing locations allocated by the `malloc` in line 4, is pointed by a global location, and therefore cannot be allocated on the thread-local heap. The abstract heap location in line 6, may be pointed by `q` location which is an argument of the inter-thread communication function, thus the location may-escape and cannot be allocated on the thread-local heap.

The precision of our algorithm is affected directly by the precision of the underlying points-to algorithm. One of the issues that mostly affects the precision of our algorithm is the way the points-to algorithm handles structure fields. There exist three kinds of points-to algorithms that respect: (i) field-insensitive points-to analysis, (ii) field-based points-to analysis, and (iii) field-sensitive points-to analysis.

Field-insensitive points-to analysis [7, 17] ignores structure fields, thus all structure members are abstracted to a single abstract location. Field-based points-to analysis [7, 17] abstracts all instances of the same structure field to a single global abstract location. For our algorithm, this means that all structure fields are considered may-escape, and cannot be considered as thread-local storage; thus it makes little sense to use these kind of algorithms for our purposes. Field-sensitive points-to analysis [30] is more precise than field-insensitive point-to analysis and field-based points-to analysis. It abstracts the fields of an allocated structure to different abstract locations.

4.2 Implementation of the Flow-Insensitive Algorithm

We have implemented our thread-local storage detection algorithm and integrated it with two points-to graphs with varying degrees of precision. The first points-to graph was produced by the CLA (compile-link-analyze) pointer-analysis of [18]. CLA provides field-based or field-insensitive points-to analysis. The second points-to graph was generated by GrammaTech CodeSurfer, which provides field-sensitive points-to analysis.

Our implementation supports the analysis of programs that follow the `POSIX thread` standard. In particular, we model the `pthread_create` function (which creates a thread and passes an argument to it) as an assignment of the thread parameter to a global variable. Thus, memory pointed by the thread parameter is conservatively assumed to be escaping.

The CLA based analysis scales better than the CodeSurfer based analysis, however it provides less precise results. On the small benchmarks we used, both implementations have been able to detect thread-local storage correctly. In general, for larger programs, the precision of field dependent analysis (as in CodeSurfer implementation) is expected to be better. However, we did not observe differences in the benchmarks we performed.

5 Experimental Results

In this section we describe the experimental results of our static analysis tool and our thread-local storage allocator. Our static analysis experimental performance results were produced on 2X2.8GHZ pentium IV processor with 1GB of memory running RedHat enterprise Linux with a kernel version of 2.4.21. Our runtime experimental performance results were produced on 8X700MHZ Pentium III processor with 8GB of memory running a RedHat enterprise Linux with a kernel version of 2.4.9-e3. We compare our allocator with the default Linux *glibc malloc*, and with the Hoard version 2.1.2d [8].

5.1 Benchmarks

Measuring the performance of multithreaded dynamic memory allocation in real life applications is almost impossible. The multithreaded servers are mostly I/O bound and the effect of memory management improvements is hard to measure. Since there are no real

Table 1. Static Analysis results. The points-to time column presents the time that the underlying points-to analysis took. The algorithm time column presents the time that our algorithm ran on the points-to graph. The total mallocs column presents the number of malloc statements in the benchmark. The identified tls_mallocs column presents the number of mallocs that our algorithm actually identified as thread-local storage. The tls opportunities column presents the number of allocation sites that are actually thread-local storage

Benchmark	Description	LOC	points-to time	algorithm time	total mallocs	identified tls_mallocs	tls opportunities
cache-thrash [8]	Cache locality test	144	$< 1s$	$< 1s$	3	2	2
cache-scratch[8]	Cache locality test	144	$< 1s$	$< 1s$	5	3	3
threadtest [8]	Scalability test	155	$< 1s$	$< 1s$	3	3	3
linux-scalability [20]	Scalability test	137	$< 1s$	$< 1s$	1	1	1
sh6bench [3]	Scalability test	557	$< 1s$	$< 1s$	3	3	3
larson [19]	Inter-threadallocations	672	$< 1s$	$< 1s$	5	0	0
consume[8]	Inter-thread allocations	141	$< 1s$	$< 1s$	5	0	0
zlib[5] (field-sensitive)	Use of zlib compression library	$12K$	$9s$	$62s$	12	11	11
zlib[5] (field-insensitive)	Use of zlib compression library	$12K$	$8s$	$1s$	12	11	11
openssl mttest[4] (field-sensitive)	multithreaded sll connections	$140K$	$1565s$	$22449s$	N/A	N/A	N/A
openssl mttest[4] (field-insensitive)	multithreaded sll connections	$140K$	$542s$	$39s$	N/A	N/A	N/A

benchmarks for dynamic memory allocators, we applied the benchmarks used to evaluate the performance of Hoard [8]. These benchmarks have become the standard defacto benchmarks for dynamic memory allocations. They have been used by [8, 11, 19, 21]. We tested the following benchmarks: *cache-trash, linux-scalability, shbench, threadtest, cache-scratch, larson, consume* [5]. The first 5 benchmarks contain allocations that have been detected as thread-local by our static analysis tool, and have been optimized to use `tls_malloc` instead of `malloc`. The last two benchmarks contain no thread-local storage, and as expected, the static algorithm correctly determines it, and these benchmarks have therefore not been optimized.

We have also added benchmarks of more realistic applications. The `Zlib` benchmark tests multithreaded usage of the `Zlib` compression library [5]. Our static analysis algorithm successfully detected allocations as thread-local. Those allocations have been optimized to use `tls_malloc` instead of `malloc`. We also tested `OpenSSL-mttest` a multithreaded test of the `OpenSSL` cryptographic library [4]. Our static algorithm did not find opportunities for optimizing the program using thread-local storage. When we

[5] We took all the open-source multithreaded benchmarks from [8]. There are two additional multithreaded benchmarks (BEMengine, and Barnet-Hut) which we did not take since we did not have their source code.

manually examined the `OpenSSL-mttest` code we verified that there were no thread-local storage opportunities.

5.2 Static Analysis Results

Static analysis results are summarized in Table 1. For the first 7 benchmarks we used Heintze's field-insensitive pointer-analysis [18] as the underlying points-to algorithm. All of these benchmarks are small and artificial memory management benchmarks. The pointer-analysis time was less than a second for all of these and so was the application of our own static algorithm. Some of the benchmarks were originally written in C++. We ported these benchmarks to C, so we can apply our static analysis tool. For the larger programs of `OpenSSL-mttest` and `Zlib` we used CodeSurfer's pointer-analysis as a back-end for our algorithm. From the experimental results we can see that applying field-sensitive pointer-analysis yields to a much longer execution time. The reason for this is that the points-to graph can be exponentially larger in that case. We can also see that the field-sensitivite analysis did not improve the analysis precision for the benchmarks we selected, even though it is theoretically more precise.

5.3 Runtime Speedup

We executed each benchmark with a different number of threads. Each benchmark performs some work that consumes a certain period of time on a single-threaded execution. When we add threads, this work is performed concurrently and we expect the execution time to be shorter. On an optimal allocator, there should be a linear relation between the number of threads and the execution time. For each benchmark we performed the following tests. (i) an execution with *glibc* — the default allocator of the Linux operating system. (ii) an execution with the `Hoard` allocator. (iii) an execution with our allocator, after we have optimized the benchmark to use thread-local storage allocations. Runtime speedups for the benchmarks are shown in Figure 6. The circle line represents `glibc` allocator, the triangle line represents `Hoard` allocator and the box line represents our `tls Hoard` allocator.

The speedup on `threadtest` benchmark (see Figure 6(a)) is between 16% to 29% compared to the `Hoard` allocator. On `linux-scalability` benchmark (see Figure 6(c)) the speedup is between 18% to 44% and in most cases it is around 40%. On `shbench` benchmark (see Figure 6(d)), the speedup is between 2% to 14%. These benchmark programs test pure scalability, without other issues such as processor cache performance and memory fragmentation. As expected, we get a significant performance improvement, since the allocator reduces the global heap contention which directly leads to better scalability. On `cache-thrash` benchmark (see Figure 6(b)) our optimizations do not improve Hoard. This benchmark checks the cache behavior of the allocator and our allocator does not handle cache issues directly, even-though thread-local storage improves locality. However, we discovered that when the amount of computations between allocations is reduced, our optimized version outperforms Hoard, since the frequency of the allocations increases the contention, and our allocator handles it better. In `Zlib` benchmark (see Figure 6(e)) the speedup is between 1% to 20%. `Zlib` benchmark represents a more realistic application that also involves I/O processing and computations.

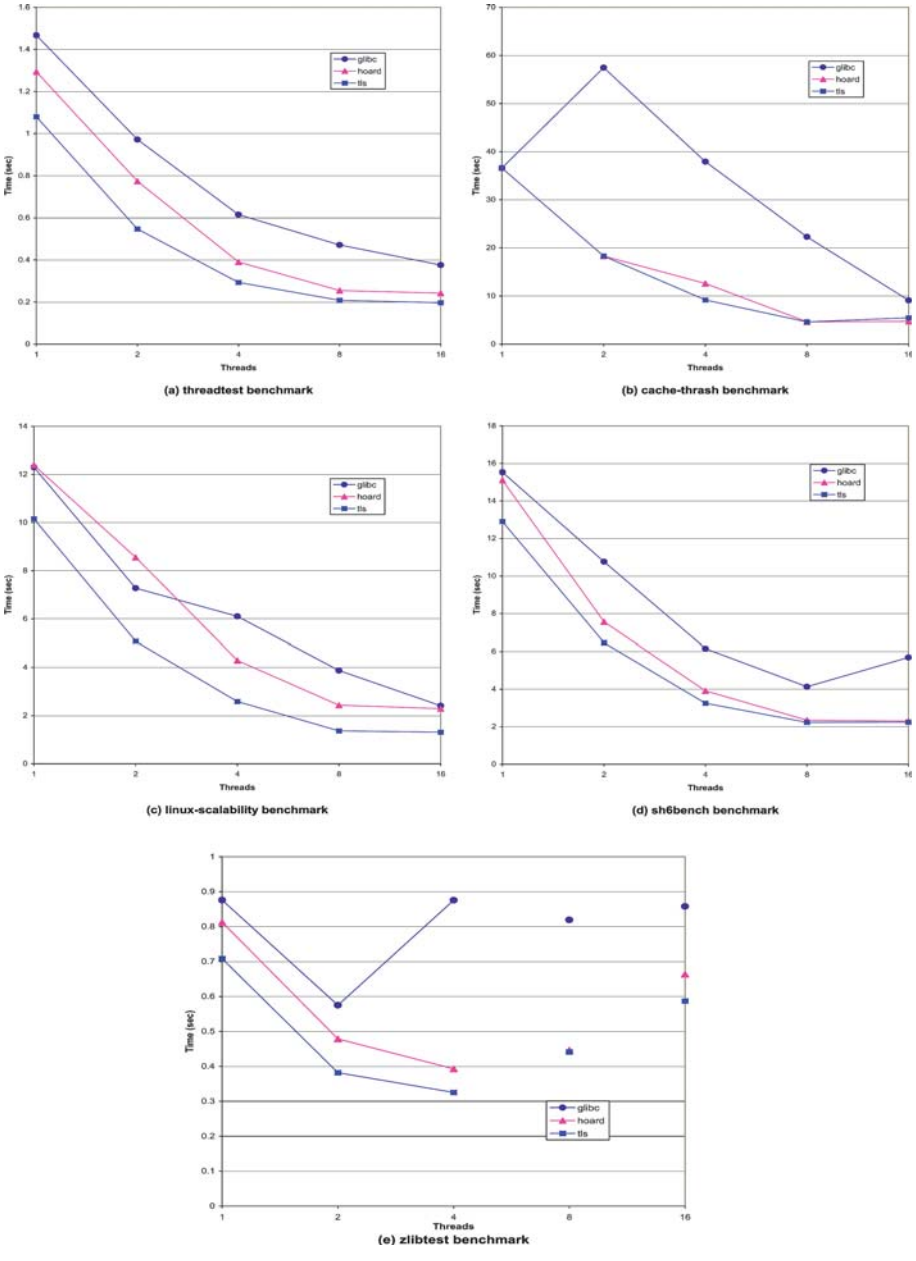

Fig. 6. Thread-local storage allocator benchmark results. The X axis is the number of threads and the Y axis is the runtime in seconds. The lines represent the *glibc* allocator, the *Hoard* allocator, and our allocator. We obtain up to 44% runtime speedup compared to the other allocators

The performance of the `Zlib` benchmark drops when the number of threads increases due to the cost of the I/O processing. However, our allocator still outperforms the others when the number of threads increases.

5.4 Summary

From the static analysis benchmark results shown in Table 1, we can deduce that the static algorithm successfully detects all the opportunities for thread-local storage for the standard memory management benchmarks. The analysis time is less than a second in these benchmarks, and the analysis is precise and identifies all opportunities for using thread-local storage. On the `Zlib` benchmark we also precisely detected all the possible opportunities for using thread-local storage. We proved that our analysis can handle large programs by running it on `OpenSSL-mttest` and `Zlib`. We could also see the significant performance overhead of using field-sensitive analysis.

The runtime benchmark results show that our allocator provides significant multithreaded scalability improvement for thread-local storage allocations. Moreover, our allocator performs better, compared to different allocators, even on a single-threaded environment. There are two potential reasons for this behavior. The first reason is that locks have some overhead even on a single-threaded environment. The second reason is the super-block holder, which we keep for each thread-local heap. These holders avoid trashing between the thread-local heap and the operating system heap and improve the locality and performance of allocation from the thread-local heap. The performance improvements for the `Zlib` benchmark result show the potential benefit of our method on more realistic programs.

6 Conclusions

Dynamic memory management in C for multithreaded applications can become a performance bottleneck. We could see the impact of the synchronization contentions by examining the memory allocation benchmarks suite. This paper shows that a thread-local storage allocator can significantly improve the performance of dynamic memory management. However, manual detection of thread-local storage is almost an infeasible task. Therefore, the paper shows that a simple sound static analysis can successfully detect heap allocation statements that can be replaced by allocating thread-local storage. We reduce the problem of finding thread-local storage to the escape analysis problem for C and solve it by using flow-insensitive points-to algorithms.

References

1. Apache http Server Project. Available at http://httpd.apache.org.
2. D. Lea A Memory Allocator. Available at http://g.oswego.edu/dl/html/malloc.html.
3. Microquill inc. Available at http://www.microquill.com.
4. openssl cryptographic library. Available at http://www.openssl.org.
5. zlib compression library. Available at http://www.zlib.org.
6. J. Aldrich, E. G. Sirer, C. Chambers, and S. J. Eggers. Comprehensive synchronization elimination for Java. Technical Report, University of Washington, Oct. 2000.
7. L. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU Univ. of Copenhagen., Copenhagen, Denmark, 1994.

8. E. Berger. Hoard: AScalable Memory Allocator for Multithreaded Applications. In *Architectural Support for Programming Languages and Operating Systems*, pages 117–128, Cambridge, Massachusetts, US, Nov. 2000.
9. E. D. Berger, B. G. Zorn, and K. S. McKinley. Reconsidering Custom Memory Allocation. In *Conf. on Object-Oriented Prog. Syst., Lang. and Appl.*, pages 1–12, Seattle, Washington, US, Nov. 2002.
10. B. Blanchet. Escape Analysis for Object Oriented Languages. Application to Java. In *Conf. on Object-Oriented Prog. Syst., Lang. and Appl.*, pages 20–34, Denver, Colorado, US, Nov. 1999.
11. H. Boehm. Fast Multiprocessor Memory Allocation and Garbage Collection. Tech Report, HP Labs, Dec. 2000.
12. J. Bogda and U. Hoelzle. Removing unnecessary synchronization in Java. In *Conf. on Object-Oriented Prog. Syst., Lang. and Appl.*, pages 35–46, Denver, Colorado, US, Nov. 1999.
13. J. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape Analysis for Java. In *Conf. on Object-Oriented Prog. Syst., Lang. and Appl.*, pages 1–19, Denver, Colorado, US, Nov. 1999.
14. M. Das. Unification-based Pointer Analysis with Directional Assignments. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, volume 35, pages 35–46, Vancouver, Canada, June 2000.
15. M. Das, B. Liblit, M. Fahndrich, and J. Rehof. Estimating the Impact of Scalable Pointer Analysis on Optimization. In *Static Analysis Symp.*, volume 2126, pages 260–278, Paris, France, July 2001.
16. T. Domani, G. Goldshtein, E. K. Kolodner, E. Lewis, E. Petrank, and D. Sheinwald. Thread-local heaps for java. In *Int. Symp. on Memory Management*, pages 76–87, Berlin, Germany, 2002.
17. N. Heintze. Analysis of Large Code Bases: The Compile-Link-Analyse Model. Unpublished Report, Nov. 1999.
18. N. Heintze and O. Tardieu. Ultra-fast Aliasing Analysis using cla: A Million Lines of C Code in a Second. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 254–263, Snowbird, Utah, US, May 2001.
19. P. Larson and M. Krishnan. Memory Allocation for Long-running Server Applications. In *Int. Symp. on Memory Management*, pages 176–185, Vancouver, Canada, Oct. 1998.
20. C. Lever and D. Boreham. malloc() performance in a multithreaded linux environment. In *USENIX, the Advanced Computing System Association*, San Diego, California, US, 2000.
21. M. M. Michael. Scalable Lock-Free Dynamic Memory Allocation. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 35–46, Washington, US, June 2004.
22. M. Rinard. Analysis of multithreaded programs. In *Static Analysis Symp.*, pages 1–19, Paris, France, July 2001.
23. E. Ruf. Effective Synchronization Removal for Java. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 208–218, Vancouver, Canada, June 2000.
24. R. Rugina and M. Rinard. Pointer Analysis for Multithreaded Programs. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 77–90, Atlanta, Georgia, US, May 1999.
25. Y. Sade. Optimizing C Multithreaded Memory Management Using Thread-Local Storage. Master's thesis, Tel-Aviv University, Tel-Aviv, Israel, 2004.
26. A. Salcianu and M. Rinard. Pointer and Escape Analysis for Multithreaded Programs. In *Principles Practice of Parallel Programming*, pages 12–23, Atlanta, Georgia, US, June 2001.
27. B. Steensgaard. Points-to Analysis in Almost Linear Time. In *Symp. on Princ. of Prog. Lang.*, pages 32–41, St. Petersburg Beach, Florida, US, Jan. 1996. ACM Press.

28. B. Steensgaard. Thread-Specific Heaps for Multi-Threaded Programs. In *Int. Symp. on Memory Management*, pages 18–24, Minneapolis, Minnesota, US, Oct. 2000.
29. R. P. Wilson and M. S. Lam. Efficient Context-Sensitive Pointer Analysis for C Programs. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 1–12, 1995.
30. S. Yang, S. Horwitz, and T. Reps. Pointer Analysis for Programs with Structures and Casting. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 91–103, Atlanta, Georgia, US, May 1999.