Memory Management Optimizations by Static Detection of Thread Local Storage

Yair Sade – Tel-Aviv University

Mooly Sagiv– Tel-Aviv University

Ran Shaham – IBM Haifa Labs

What is Memory Management?

- Responsible for program heap allocations
- C Memory Management routines
  - malloc()
  - free()
  - realloc()
  - …

How is it implemented?

- Global heap
- malloc() allocates memory from the heap
- free() returns the memory to the heap

Motivation

- Memory management can become a bottleneck on multithreaded environments
- The global heap is shared among the program threads
- To avoid data races, the global heap must be synchronized

Main Results

- Improved runtime routines for specialized memory management in multithreaded environments
- A static analysis algorithm for automatic detection of specialized allocations
- Implementation for full C
- Experimental results
  - 10-50% memory management improvements for standard memory management benchmarks
  - On large benchmarks static analysis is still imprecise

Motivating Example

```
void *g;
int f(void *);
int main() {
   void *p;
   …
   p = malloc(..);
   pthread_create(f, p);
}
int f(void *q) {
    void *l;
    free(q);
    l = malloc(..);
    g = malloc();
    …
    free(l);
}
```

Outline

- Multithreaded MM Performance Problem
- Static Analysis Algorithm Description
- Correctness proof
- Experimental results
- Related work
- Conclusions

Performance Problem

- Synchronization leads to contention
  - Threads can be blocked
  - Parallelism reduced
  - Context switches occur
- Synchronization primitives have overhead even on single-processor machines due to system calls

Performance Problem (cont.)

- On SMP machines the problem can become acute
  - Threads running on different processors call memory management routines
  - Processors are blocked waiting for heap lock
  - System utilization drops

Runtime Solutions

- High-scaled specialized memory management implementations
  - E. Berger’s Hoard
  - H. Boehm’s GC based
- Reduce the problem

Programmable Solutions

- Application specific allocators
- Use of thread local arenas
- Drawbacks
  - Requires special application design
  - Very hard to implement on large projects
  - Existing applications

Our Solution

- “Thread local storage”
  - memory only accessible by a single thread
- Can be allocated on a separate heap
  - the thread heap
- Almost no synchronizations on thread heap

Our Solution (cont.)

- Runtime allocator (Hoard based) that implements thread local storage
  - tls_malloc()
  - tls_free()

Our Solution (cont.)

- Static analysis tool
  - Detects thread local memory allocations
  - malloc() can be replaced to tls_malloc()
  - free() can be replaced to tls_free()

Our Solution (cont.)

```
void *g;
int f(void *);
int main() {
   void *p;
   …
   p = malloc(..);
   pthread_create(f,
p);
}
int f(void *q) {
    void *l;
    free(q);
    l = malloc(..);
   g = malloc();
    …
    free(l);
}
```

Replace with tls_malloc()

Replace with tls_free()

Our Solution (cont.)

- Static analysis must be sound
  - Transformation preserves behavior
  - Tool's output must be thread local storage
- Static analysis gives an approximation results
  - False negatives can occur
  - Not all thread local storage are detected

Location Creator

- Associate a thread creator for each memory location
  - Stack location
  - Heap location
  - Global location

Escaped Location

- A memory location escapes
  - Accessed by a different thread than the creating thread
  - Once memory location escaped, considered as escaped ever since

Thread Local Location

- Non-escaping location is called “thread local location”
- Thread local heap location is accessed only by its creator thread
- Can be allocated safely on Thread Local Storage
  - We can use our tls_malloc function

Shared Locations

- Shared memory location
  - A location that pointed by locations of different creators
  - For example – global variables are shared locations

Shared Locations (cont.)

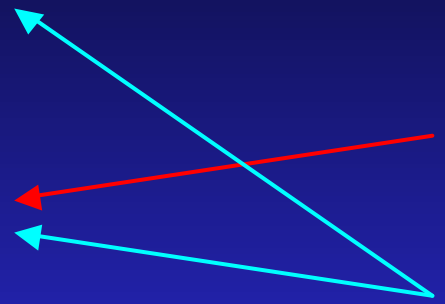
Escaped	Shared	Possible
No	No	Yes
No	Yes	Yes
Yes	No	No
Yes	Yes	Yes

Shared Locations (cont.)

- A location can become shared on the following conditions
  - The location is global location
  - The location has been passed as a thread parameter (by pthread_create)
  - The location is pointed by shared location

Shard Locations (cont.)

```
void *g;
int f(void *);
int main() {
   void *p;
   …
   p = malloc(..);
   pthread_create(f, p);
}
int f(void *q) {
    void *l;
    free(q);
    l = malloc(..);
    g = malloc();
    …
    free(l);
}
```



p is escaped location

p,g are shared locations

Static Analysis Tool

- Input for the algorithm
  - C Program
  - Pre-generatged pointer information
- Output
  - Thread local locations

Limitations

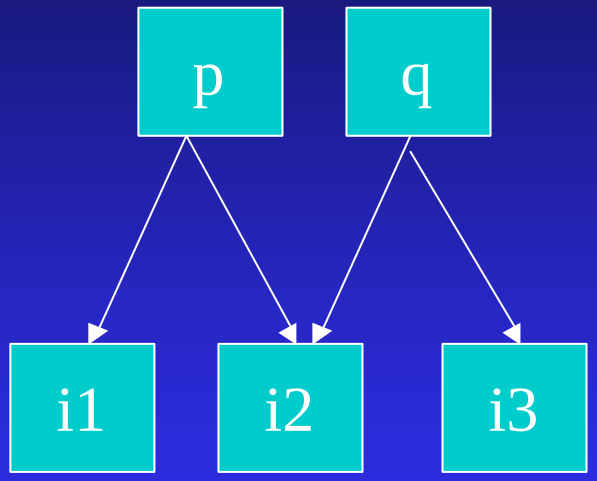
- ANSI C programs only
- pthreads threading model

Points-to Graph

- Describes the pointer relations between abstract memory locations
- Abstract location represents a set of real memory locations
  - Heap
  - Stack

Points-to Graph (cont.)

```
…
int i1,i2,i3;
int *p, *q;
p = &i1;
p = &i2;
q = &i3;
q = p;
…
```

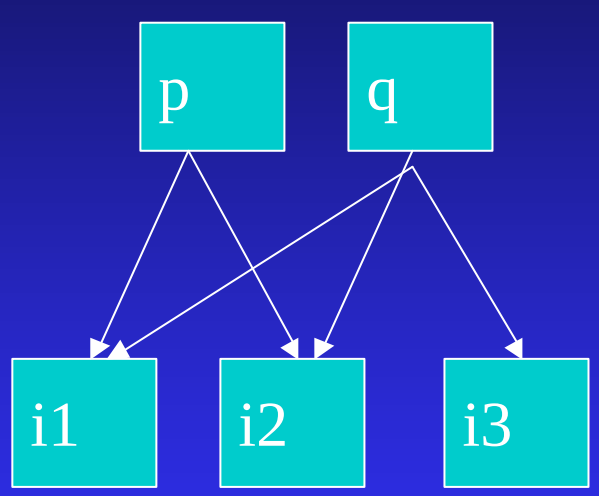


Points-to Algorithms

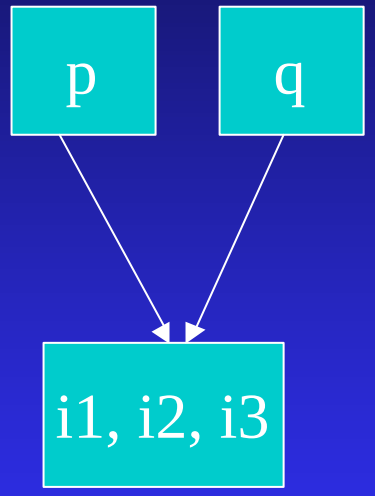
- Gives an approximation for the points-to graph
- Two general algorithms
  - Unification based (Steensgaard *et al.*)
  - Constraint based (Andersen *et al.)*
- Both algorithms are
  - Flow insensitive
  - Context insensitive

Points-to Algorithms (cont.)

```
…
int i1,i2,i3;
int *p, *q;
p = &i1;
p = &i2;
q = &i3;
q = p;
…
```



Andersen



Steensgaard

Static Algorithm

- Requires pre-generated points-to graph
- Context and flow insensitive
- Interprocedural algorithm

Subset of C

- Simple C statements
  - pthread_create(t,p)
  - a = b
  - a = &b
  - a = *b
  - *a = b
  - a = malloc()

Static Algorithm Description

- Initialization
  1. Mark all abstract location as not shared
  2. Mark global locations as shared
  3. Mark each abstract location which is pointed by global as shared

Initialization Phase

```
void *g;
int f(void *);
int main() {
   void *p;
   …
   p = malloc(..);
   pthread_create(f, p);
}
int f(void *q) {
    void *l;
    free(q);
    l = malloc(..);
    g = malloc();
    …
    free(l);
}
```

Shared Locations
g

Mark and Replace Phases

- Mark shared locations phase
- For each malloc/free call
  - If the location not marked as shared then malloc/free can be replaced to TLS

Mark Phase

- Handling pthread_create(f, b)
  - Mark each b as shared
  - Mark each abstract location pointed by b as shared

Mark Phase (cont.)

```
void *g;
int f(void *);
int main() {
   void *p;
   …
   p = malloc(..);
   pthread_create(f, p);
}
int f(void *q) {
    void *l;
    free(q);
    l = malloc(..);
    g = malloc();
    …
    free(l);
}
```

Shared Locations
g
p,q

Mark Phase (cont.)

- Handling a=b
  - If a is pointed by shared abstract location then
    - Mark each abstract location pointed by b as shared

Mark Phase (cont.)

- Handling a=&b
  - If a is pointed by shared abstract location then
    - Mark b as shared
    - Mark each abstract location pointed by b as shared

Mark Phase (cont.)

- Handling a=*b
  - If a is pointed by shared abstract location then
    - Mark each abstract location pointed by *b as shared

Mark Phase (cont.)

- Handling *a=b
  - If *a is pointed by shared abstract location then
    - Mark each abstract location pointed by b as shared

Replacement Phase

```
void *g;
int f(void *);
int main() {
   void *p;
   …
   p = malloc(..);
   pthread_create(f, p);
}
int f(void *q) {
    void *l;
    free(q);
    l = malloc(..);
    g = malloc();
    …
    free(l);
}
```

Shared Locations
g
p,q

Replace with tls_malloc()

Replace with tls_free()

Correctness Proof Sketch

- Every location that the algorithm marks as local never escapes
  - We show that ESC $\subseteq$ SHARED
  - Separation to concrete and abstract semantics

Correctness Proof Sketch (cont.)

- For each statement show local correctness of our abstraction
  - ?(ABS_SHARED) $\supseteq$ SHARED
- Global correctness theorem (CC77)

Complexity

- Points-to analysis
  - Between $O(n^3)$ to linear time
  - The more complex the more precise
  - There are many implementations
- Our algorithm
  - Linear time complexity

Static Algorithm Implementation

- Points-to analysis
  - N. Heintze scalable implementation to Andersen Points-to analysis
- C parser is based on CKIT of N.Heintze

Memory Management Implementation

- Based on Hoard allocator
- A special heap for each thread – thread local heap
- Actions on thread local heap are synchronization free

Preliminary Results

- We tested malloc bencharmks
- On dual processor Linux
  - ThreadTest, CacheStresss, LinuxScalability – 20%
- On 16 processors SGI Irix
  - ThreadTest, Cache-Trash – 50%

Related work

- Escape Analysis (Bogda, Ruf)
- Steensgaard GC optimizations
- Analysis of multithreaded programs (Rinard)
- Points-to Analysis (Andersen, Steensgaard, Manuvir, Heintze, Rinard)
- Multithreaded Memory Management (Zorn, Boehm, Berger)

Conclusions

- Contributions
  - New escape analysis algorithm for C programs
  - Static detection of thread local storage
  - High performance allocator for thread local storage

Conclusions (cont.)

- Our research status
  - Improve precision of the analysis
  - Find practical benchmarks

The End

yairs@cyber-ark.com