Memory Overflow Protection for Embedded Systems using Runtime Checks, Reuse and Compression

By:
Surupa Biswas
Matthew Simpson
Rajeev Barua

Department of Electrical & Computer Engineering
University of Maryland

1. INTRODUCTION

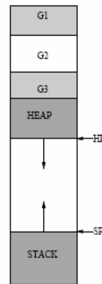
- Out-of-memory errors can be a serious problem in computing, but to different extents in desktop and embedded systems.
- In desktop systems, virtual memory reduces the ill-effects of running out of memory in two ways.
  - First, when a workload does run out of physical main memory (DRAM), virtual memory makes available additional space on the hard disk called swap space, allowing the workload to continue making progress.
  - Second, when either the stack or heap segment of a single application exceeds the space available to it, hardware-assisted segment-level protection provided by virtual memory prevents the overflowing segment from overwriting useful data in other applications.

1. INTRODUCTION

- Embedded systems typically do not have hard disks, and often have no virtual memory support.
  - Requires an accurate compile-time estimation of the maximum memory requirement of each task across all input data sets.
  - For a concurrent task set, the physical memory must be larger than the sum of the memory requirements of all tasks that can be simultaneously live.

1. INTRODUCTION

- Accurately estimating the maximum memory requirement of an application at compile-time is difficult,
- Consider that data in applications is typically sub-divided into three segments
  – global, stack and heap data.
  - The global segment has fixed size at compile time
    - Easy to estimate
  - The stack and heap grow and shrink at run-time
    - Hard to estimate



1. INTRODUCTION

- Estimating the **stack** size at compile-time
  - Stack grows with each procedure and library call, and shrinks upon returning from them.
  - the maximum memory requirement of the stack can be accurately estimated by the compiler as the longest path in the call graph of the program from *main()* to any leaf procedure.
  - Fails for at least the following
    - (i) Recursive functions
    - (ii) Virtual functions
    - (iii) First-order functions in imperative languages like C
      - First-order functions are those that are assigned to function variables, and called indirectly through those variables, so that the compiler may not know which function is actually called when a function variable is called.
    - (iv) Languages, such as GNU C, which allow stack arrays to be of runtime dependent size

1. INTRODUCTION

- the stack may run out of memory even when its size is predictable.
  - The size of the heap is unpredictable, since the stack and the heap typically grow towards each other.
  - Even when both its stack and heap requirements are predictable. This can happen in pre-emptive multi-tasking workloads, common in many embedded systems.

1. INTRODUCTION

- Estimating the **Heap** size at compile-time
  - **more difficult**. The heap is typically used for dynamic data structures such as linked lists, trees and graphs.

    -unknowable at compile-time.

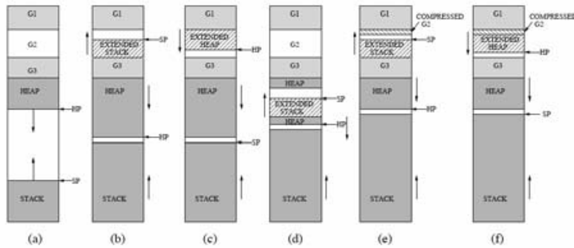
1. INTRODUCTION

- The usual industrial approach:
  - run the program on different input data sets and observe the maximum sizes of stack and heap.
  - Memory requirement estimate is multiplied by a safety factor to reduce the chance of memory errors,

1. INTRODUCTION

- This paper proposes a scheme for software-only memory protection and memory reuse in embedded systems that takes a three-fold approach to improving system reliability.
  - **Safety run-time checks**
    - checking for stack or heap overflow requires a run-time check for overflow at each procedure call and each *malloc()* call
  - **Reusing dead space**
    - (i) when the overflowing stack and heap are allowed to grow into dead global variables, especially arrays.
    - (ii) when the stack is allowed to grow into free holes in the heap segment.
  - **Compressing live data**
    - compresses live data.The compressed data is later de-compressed before it is accessed.

1. INTRODUCTION



2. RUNTIME CHECKS

- **Heap** checks:
  - If the *malloc()* finds that no free chunks of adequate size are available then an out-of-memory error is reported. It exists by default in most versions of *malloc().*
- **Stack** checks:
  - Inserted at each procedure call. These are new and add run-time overhead.
  - The compiler inserts code at the entry into each function, which compares the values of the new, updated stack pointer and the current allowable boundary for the stack.

2. RUNTIME CHECKS

- The boundary for the **stack**:
  - (i) the heap pointer, if the heap adjoins the growing direction of the stack.
  - (ii) the base of the adjoining stack, if another task's stack adjoins the growing direction of stack. –Multiple Tasking
  - (iii) the end of memory, if the stack ends at the end of memory.

PER-PROCEDURE SAFETY CHECK CODE

```
1.  if (Stack-Ptr < ORIGINAL_BOUND) {        /* Stack Overflow */
2.       call routine to handle out-of-memory condition
3.  }
```

2. RUNTIME CHECKS

- **Reduce the overheads**
  -rolling checks optimization.
  - enough to check *once at the start of the parent that there is enough space for the stack frames of both parent and child procedures together.*
  - The check for the child is 'rolled' into the check for the parent, eliminating the overhead for the child.

2. RUNTIME CHECKS

- **Issues for rolling checks optimization**
- A child procedure's check cannot be rolled into its parent if heap data is allocated inside the parent before the child procedure is called.
- In object-oriented languages if the call to the child from the parent is an unresolved virtual function call, then the child's check cannot be rolled to the parent
- Since a call-graph represents potential calls and not actual calls, it is possible that for a certain data set a parent may not call a child procedure at all.
  - Limit the rolling checks optimization such that the rolled stack frame size does not exceed 10% of the maximum observed stack + heap size in the profile data.
- Rolling checks can be permitted inside of recursive cycles in the application program, but not out of recursive cycles.

3. REUSING GLOBALS FOR **STACK**

- First, the compiler performs liveness analysis to detect dead global arrays.
- Second, selects one of the global arrays that is dead, and grows the stack into it.
- **Identifying dead globals:**
  - First, the compiler divides the program up into several regions, and for each region, builds a list (called Reuse Candidate List) of global arrays that are dead throughout that region and also dead in all functions that are called directly or indirectly from that region.
  - Second, the Reuse Candidate List is sorted at compile-time in decreasing order of size to give preference to large arrays for reuse.
  - Third, at run-time, when the program is out of memory it looks up the Reuse Candidate List for that region and selects the global variable at the head of the list to extend the stack into.

3. REUSING GLOBALS FOR **STACK**

- *Data-Program Relationship Graph (DPRG)*

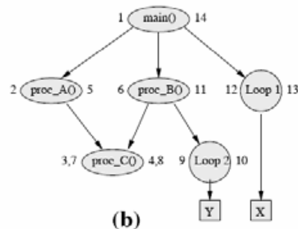
```
main () {
   proc-A()
   proc-B()
   while (...){X = ... } /* Loop 1 */
}

proc-A () {
   proc-C()
}

proc-B () {
   proc-C()
   for ( ... ){Y = ... }      /* Loop 2 */
}

proc-C () { ... }
```

(a)



3. REUSING GLOBALS FOR **STACK**

- **Region-merging optimization:**
  - merging regions whenever possible to reduce the overhead. In particular, if two regions that are executed <u>consecutively</u> at run-time are such that they have the exact same Reuse Candidate Lists, they are merged into a single region.
- **Growing stack into globals**

4. REUSING GLOBALS FOR **HEAP**

- **Implementation**:
  - First, the Reuse Candidate Lists are sorted at compile-time by next time- of-access and size.
  - Second, the *malloc()* library function is modified to make a call to a special Out-of-Heap Function when there is no available free chunk to satisfy the allocation request.
    - *Malloc()* is modified such that, instead of returning -1, when it is unable to find any chunk on the free-list capable of satisfying the current allocation request, it makes a call to the Out-of-Heap Function,
  - Third, the compiler inserts the Out-of-Heap Function in the code;

5. REUSING **HEAP** FOR **STACK**

- **Steps**:
  - When the stack is out-of-memory
  - First tries to grow the stack into dead globals.
  - Second grown into free holes in the heap.
  - To grow into the heap, a special *malloc()* call is made to allocate a chunk in the heap among its free holes, and thereafter the **stack** is grown into the returned chunk.
  - This method of growing into free holes in the heap is unnecessary when these holes are periodically eliminated using heap compaction. Heap compaction is usually possible only in systems that do garbage collection.

6. COMPRESSING GLOBALS FOR STACK

- scheme differs from the scheme for growing the stack into dead globals in the following three ways.
  - The reuse candidates are extended to include **live** global arrays.
  - At run-time, when the stack is about to grow into a particular candidate in the global segment, if the candidate chosen is live at that point, it is compressed and saved so that it can be restored when the array is accessed later.
  - The code inserted by the compiler at the start of every region is augmented to ensure that if reuse has started, then all compressed global arrays accessed in the following region are de-compressed in their original locations.

7.COMPRESSING GLOBALS FOR **HEAP**

- First, it uses the same Reuse Candidate Lists that are sorted according to the next-time-of-access and size of the global array.
- Second, once the system has run out of heap space, it makes a call to the Out-of-Heap Function, which is now slightly modified to support compression.
  - It first compresses the global array.
  - Including maintaining book-keeping information in the Compression Table.
  - Finally, makes a call to the free library function with a pointer to the space freed up by compression.
- Third, before every region a check is made to see if reuse has started. If it has, all compressed globals are de-compressed as in that section.

8. COMPRESSION ALGORITHM

- It should compress program data to a high degree.
- It should have a very low or zero persistent memory overhead.
- Since compression is done at run-time, the sum of the compression and de-compression times should be small.
  - (i) LZO
    <David Solomon. Data Compression: The Complete Reference. Springer-Verlag Inc., New York, 2000.>
  - (ii) WKdm
    <Paul R. Wilson, Scott F. Kaplan, and Yannis Smaragdakis. The case for compressed caching in virtual memory systems. In *Proceedings of the USENIX Annual Technical Conference*, Monterey, CA, June 1999.>
  - (iii) WKS
    <Paul R. Wilson, Scott F. Kaplan, and Yannis Smaragdakis. The case for compressed caching in virtual memory systems. In *Proceedings of the USENIX Annual Technical Conference*, Monterey, CA, June 1999.>

9. SPACE OVERHEADS OF ROUTINES

- **First** source is calls are made to certain functions such as the Out-of-Heap Function, the compression and de-compression.
- **Second** source of memory overhead is to store the Reuse Candidate Lists for every region in the same memory device where program code is stored, which is usually readonly memory (ROM) in embedded systems.
  - Do not change at run-time.

10. LIVENESS ANALYSIS

- First, in object-oriented languages when a virtual function is called, the compiler does not usually know which real function is actually called at run-time.
- Second, in imperative languages such as C, first-order functions may prevent knowledge of the call-graph at compile-time.
- Liveness analysis in such situations may not be precise, but is always conservative in that it never declares a live variable to be dead.
- For object-oriented languages, liveness analysis at compile-time has been investigated in the paper listed below. Restricting the set of functions a virtual function may call.

< Patrik Persson. Live memory analysis for garbage collection in embedded systems. In *Proceedings of the ACM SIGPLAN 1999 workshop on Languages, compilers, and tools for embedded systems*, pages 45–54. ACM Press, 1999.>

11. RESULTS

- The proposed techniques have been implemented in the public-domain GCC cross-compiler targeting the Motorola MCore embedded processor.

Benchmark	Source	Description	Total Data Size (in bytes)	Lines of Code
SUSAN	MIBench	Digital Image Processing	383000	5733
HISTOGRAM	UTDSP	Image Enhancing Application	17850	634
KS	PTRDist	Graph Partitioning Tool	31400	2231
JPEG	UTDSP	Image Encoding and Decoding	169000	18758
SPECTRAL	UTDSP	Power Spectral Estimation of Speech	3200	1218
LPC	UTDSP	Linear Predictive Coding Encoder	8000	4377

Table 1: Benchmark programs and characteristics.

Benchmark	Run-time Increase (%)		Code size Increase(%) (with optim.)
	Without Optimization	With Optimization	
SUSAN	0.8	0.1	0.1
HISTOGRAM	3.5	2.2	0.06
KS	3.8	1.5	0.01
JPEG	2.0	0.2	0.2
SPECTRAL	1.1	0.6	0.1
LPC	3.7	2.2	0.1
Average	*2.5*	*1.1*	*0.09*

Table 2: Overheads for Safety Checks

11. RESULTS

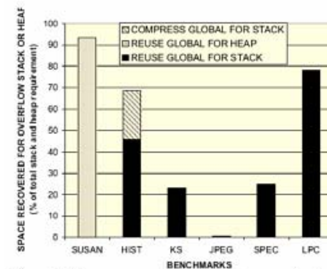


Figure 7: Extra space recovered for stack and heap as a fraction of total stack and heap requirement for each benchmark

11. RESULTS

Benchmark	Increase in Run-time (%)		Increase in Code-size (with optimization)	
	Without Optimization	With Optimization	Due to checks (%)	Due to added routines (KB, %)
SUSAN	1.8	0.3	0.2	6.7, 1.5
HISTOGRAM	10.6	6.5	0.2	14.3, 4.0
KS	8.8	3.6	0.06	6.7, 1.7
JPEG	4.6	0.4	0.4	6.7, 2.1
SPECTRAL	3.3	1.9	0.4	6.7, 1.6
LPC	11.1	6.5	0.3	6.7, 1.5
Average	*6.7*	*3.2*	*0.26*	*2.07%*

Table 3: Overheads for Memory Reuse and Compression Schemes

MTSS: Multi Task Stack Sharing for Embedded Systems

By:
Surupa Biswas
Matthew Simpson
Rajeev Barua

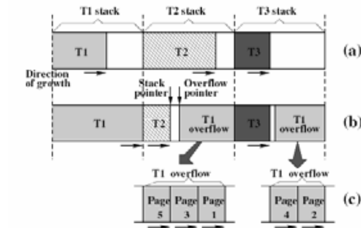
Department of Electrical & Computer Engineering
University of Maryland

12. MTSS

- MTSS: a multi-task stack sharing technique, that grows the stack of a particular task into other tasks in the system after it has overflow its bounds.
- Cactus stack memory layout
- The stacks for all tasks that can be simultaneously active (running or pre-empted) are non-overlapping in memory. The heap is allocated from a free list shared across tasks.

12. MTSS

- Cactus stack memory layout

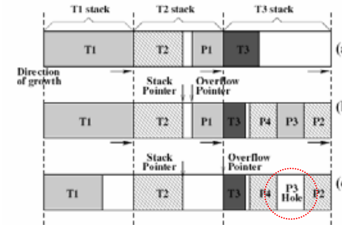


12. MTSS

- **First**, run-time checks are inserted by the compiler to detect stack overflow in each task. And The set of stack pointers for inactive (swapped out) tasks is stored as an array in memory.
- **Second**, each overflow pointer is assigned to the base of a particular task's stack.
- **Third**, overflow pointers are always grown in the direction opposite to that of the growth of the stack pointer, that is from lower memory addresses to higher memory addresses.
- **Fourth**, each time a page is allocated for an overflowing stack in a task, the overflow pointer for that task is incremented by the size of the page.

12. MTSS

- Holes:
  - classifying every page in a task stack as either free or filled. This information is maintained in an array data structure for each task.



12. MTSS

- **Re-using heap for stack** on Cactus stack memory layout
- Previous method can be easily extended to allow for reuse of the heap when a stack frame overflows, and there is no space available across all the tasks in the system. Since in a multi-tasking system the heap is shared by all the tasks.

12. MTSS

- Using ARM GCC v3.4.3 cross compiler targeting the ARM7TDMI embedded processor.

Workload	Benchmark	Description	Number of lines of code	Stack size Allocated(Bytes)
Automotive	Basicmath	Basic Math	132	1024
	Qsort	Quick Sort Algorithm	78	65536
	Bitcnt	Bit Manipulation	383	1024
	Susan	Digital Image Processing	2183	13824
Security	Blowfish	Block Cipher Encryption/Decryption	2362	6144
	PGP	Public Key Encryption	34973	65536
	Rijndael	Block Cipher Encryption/Decryption	1812	1536
	SHA	Secure Hash Algorithm	286	10240
Telecomm	ADPCM	Pulse Code Modulation	759	768
	FFT	Fast Fourier Transform	505	1280
	CRC32	Cyclic Redundancy Check	307	1024
	GSM	Voice Encoding/Decoding	6062	2176
Network	Dijkstra	Shortest Path Algorithm	371	1216
	Patricia	Tries for Network Routing Tables	620	1280
	Treeadd	Recursive sum in balanced B-tree	287	1280
	TSP	Traveling Salesman Problem	603	1856

Table 1: Multi-tasking benchmark programs and characteristics

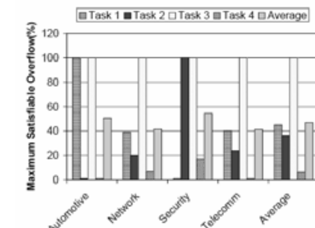
12. MTSS

Workload	Code Size Increase(%)	Run-time Increase(%)	Energy Increase(%)
Automotive	2.15	5.43	2.02
Security	1.25	0.84	1.07
Telecomm	3.52	0.62	0.03
Network	3.41	0.47	1.68
(Average)	2.59	1.84	1.20

Table 2: Run-time, code size and energy overheads of MTSS

12. MTSS

- Maximum Satisfiable Overflow (MSO)
  - defined as the maximum amount of stack space that can be recovered for each task expressed as a percentage of the total stack allocated to the task.



12. MTSS

- Proportional Reduction Satisfiability (PRS)
  - An alternate use of MTSS is to decrease the physical memory required by an embedded system while maintaining the same reliability.

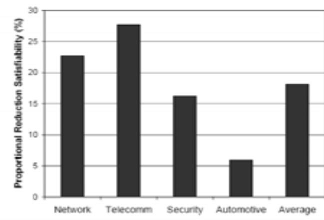


Figure 7: Proportional Reduction Satisfiability for different workloads