# Memory Overflow Protection for Embedded Systems using Run-time Checks, Reuse and Compression

Surupa Biswas
ECE Department
Univ. of MD, College Park
surupa@Glue.umd.edu

Matthew Simpson
Computer Engg. Department
Clemson University
simpsom@clemson.edu

Rajeev Barua
ECE Department
Univ. of MD, College Park
barua@Glue.umd.edu

## ABSTRACT

Out-of-memory errors are a serious source of unreliability in most embedded systems. Applications run out of main memory because of the frequent difficulty of estimating the memory requirement before deployment, either because it depends on input data, or because certain language features prevent estimation. The typical lack of disks and virtual memory in embedded systems has two serious consequences when an out-of-memory error occurs. First, there is no swap space for the application to grow into, and the system crashes. Second, since protection from virtual memory is usually absent, the fact that a segment has exceeded its bounds is not even detected and hence no pre-crash remedial action is possible.

This work improves system reliability in two ways. First it proposes a low-overhead system of run-time checks by which the out-of-memory errors are detected just before they will happen, by using carefully optimized compiler-inserted run-time check code. Such error detection enables the designer to incorporate system-specific remedial action, such as transfer to manual control, shutting down of non-critical tasks, or other actions. Second, this work proposes five related techniques that can grow the stack or heap segment after it is out of memory, into previously un-utilized space such as dead variables and space freed by compressed live variables. These techniques can avoid the out-of-memory error if the extra space recovered is enough to complete execution.

Results from our benchmarks show that the overheads from the system of run-time checks for detecting memory overflow are extremely low: the run-time and code-size overheads are 1.1% and 0.09% on average. When the reuse functionality is included, the run-time and code-size overheads increase to only 3.2% and 2.33%, but the method is able to grow the stack or heap beyond its overflow by an amount that ranges from 0.7% to 93.5% of the combined stack and heap size.

## Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors; D.4.5 [**Operating Systems**]: Reliability; D.4.2 [**Operating Systems**]: Storage Management; C.3 [**Special-Purpose And Application-Based Systems**]: Real-time and embedded systems


*CASES'04*, September 22-25, 2004, Washington DC, USA.


## General Terms

Reliability, Languages

## Keywords

Out-of-memory errors, runtime checks, reuse, data compression, stack overflow, heap overflow, reliability

## 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 [15] 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. Such protection ensures than an application with an excessive memory requirement, manifested by an unacceptable level of thrashing, can be terminated by the user without crashing the system.

Embedded systems, on the other hand, typically do not have hard disks, and often have no virtual memory support either. This means that out-of-memory errors leave the system in greater peril [25]. For correct execution, the designer must ensure a rather severe constraint – that the total memory footprint of all the applications running concurrently fits in the available physical memory at all times. *This requires an accurate compile-time estimation of the maximum memory requirement of each task across all input data sets.* Thereafter, choosing a physical memory size larger than the maximum memory requirement of the embedded application guarantees correct execution. 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, *i.e.,* running or pre-empted before completion, at a time.

Unfortunately accurately estimating the maximum memory requirement of an application at compile-time is difficult, increasing the chance of out-of-memory errors. To see why estimation is difficult, consider that data in applications is typically sub-divided into three segments – global, stack and heap data. The global segment is the only one whose size is easy to estimate, as it is of a fixed size that is known at compile-time. The stack and heap grow and shrink at run-time. Let us consider each in turn.

Estimating the stack size at compile-time is difficult for the following reasons. Consider that the stack grows with each procedure and library call, and shrinks upon returning from them. Given this

behavior, 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. However stack size estimation from the call-graph fails for at least the following four cases: (i) recursive functions, which cause the longest call-graph path to be of unbounded length; (ii) virtual functions in object-oriented languages, which result in a partially unknown call-graph; (iii) first-order functions in imperative languages like C, which also result in a partially unknown call-graph; and (iv) languages, such as GNU C, which allow stack arrays to be of run-time-dependent size, causing the procedure stack frame to be of unknown size at compile-time. In all these cases, estimating the stack size at compile-time may be impossible.

Paradoxically, the stack may run out of memory even when its size is predictable. This can happen if the size of the heap is unpredictable, since the stack and the heap typically grow towards each other. Further, the stack may run out of space, even when both its stack and heap requirements are predictable. This can happen in pre-emptive multi-tasking workloads, common in many embedded systems. In such environments, the stacks of the different tasks are given fixed amounts of space each, while the heap is allocated from a free-list shared across tasks. When a task is pre-empted (interrupted) before completion, its stack and heap remain in memory. Hence, if the stack sizes of all the tasks are predictable, but the heap size of *even one* of them is not, the task whose stack abuts the heap, may run out of space.

Estimating the heap size at compile-time is even more difficult for the following reason. The heap is typically used for dynamic data structures such as linked lists, trees and graphs. The sizes of these data structures are highly data-dependent and thus unknowable at compile-time.

Lacking an effective way to estimate the size of the stack and heap at compile-time, the usual industrial approach is to run the program on different input data sets and observe the maximum sizes of stack and heap [7]. Unfortunately, this approach of choosing the size of physical memory never guarantees an upper bound on memory usage for all data sets, and thus out-of-memory errors are still possible. Sometimes the memory requirement estimate is multiplied by a safety factor to reduce the chance of memory errors, but there is still no guarantee of error-free execution. Indeed, the safety factor used for determining memory size is often limited since many embedded systems have a low per-unit cost budget.

The possibility of out-of-memory faults takes a toll on the reliability of embedded systems. Unlike in desktops where a system crash is often no more than an annoyance, in an embedded system, a crash can lead to loss of functionality of the controlled system, loss of revenue, industrial accidents, and even loss of life, depending on the type of embedded system. Moreover, the lack of virtual-memory-based protection implies that an out-of-memory error may not even be detected by the embedded system. Without protection, the system does not check if the stack, for example, has exceeded the space for it – the only observable effect is incorrect functionality. Lacking such a check, the embedded system cannot take corrective action before the crash occurs, such as shutting down the system safely, sending a message to the operator to take over manual control of the system to ensure safe operation, or shutting down low-priority processes to free up memory.

The problem of embedded systems lacking hardware protection and their consequent unreliability has been widely recognized and lamented by industry practitioners. In [17] the authors argue for some form of memory protection, and write, "It's truly a wonder that non-memory protected operating systems are still used in complex embedded systems where reliability, safety, or security are important." In [1], the authors write about the desirability of memory protection in future systems. They write "Overrun protection would, for example, allow . . . stack overflows to be trapped to prevent corruption of other tasks' memory areas. An even more fault-tolerant system can be envisioned by incorporating . . . (resource limit reached) thresholds that trigger appropriate recovery actions." In [25], in an article appropriately titled "Programming Without A Net", the authors point out that even if an embedded system does have a sophisticated OS, it still does not have a good solution to the memory protection problem without hardware support, which is often unavailable.

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. Each component is described in turn below.

**Safety run-time checks** The first technique proposed to improve system reliability is to modify the application code in the compiler to insert software checks for all out-of-memory conditions. Lacking virtual memory, most embedded systems do not check for out-of-memory conditions; examples include [23, 16, 11, 32]. With such checks, the embedded system can take corrective action when it runs out of memory. One can imagine industrial and transportation scenarios where warning the operator to assume manual control can prevent deadly and expensive accidents. In industrial control systems, shutting down the system can also prevent accidents.

In a naive implementation, checking for stack or heap overflow requires a run-time check for overflow at each procedure call and each *malloc()* call in the program. We describe the *rolling-checks optimization* that is able to selectively eliminate many of these checks while retaining the guarantee of always detecting overflow.

This system of safety run-time checks is a *stand-alone method that can be implemented by itself*. The remaining techniques below for reusing dead space and compressing live data are *optional*, and can be implemented if the designer wants to use previously unusable memory, at the cost of some implementation complexity. The reuse and compression methods below augment the safety run-time scheme to reuse memory when a segment overflows.

**Reusing dead space** Our second technique aims to reduce the application's memory footprint by allowing segments (stack or heap) that run out of memory to grow into non-contiguous free space in the system, when available. Two cases are explored: (i) when the overflowing stack and heap are allowed to grow into dead global variables, especially arrays; and (ii) when the stack is allowed to grow into free holes in the heap segment. By using previously unutilized space, the out-of-memory error is postponed and may be avoided if this extra space is enough to complete execution.

Figure 1 illustrates how the overflowing stack or heap grows into various sources of free space in the system. Figure 1(a) shows the memory layout during normal operation, when no segment is out of memory. Figure 1(b) shows the overflowing stack growing into the space for the dead global variable *G2*. Figure 1(c) shows the overflowing heap growing into the space for the same dead global. Figure 1(d) depicts the overflowing stack growing into free holes in the heap. Figures 1(e) and (f) are discussed later.

Growing the stack and heap into non-contiguous space is implemented in three steps. First, at compile-time, liveness analysis detects dead global variables at each point in the code as possible candidates for growing into. This liveness information is then stored in run-time data structures. To reduce the size of the data structures, liveness information is stored per region, instead of per instruction, where regions are defined in section 4. Dead variables that may become live in a later region may also be used for grow-

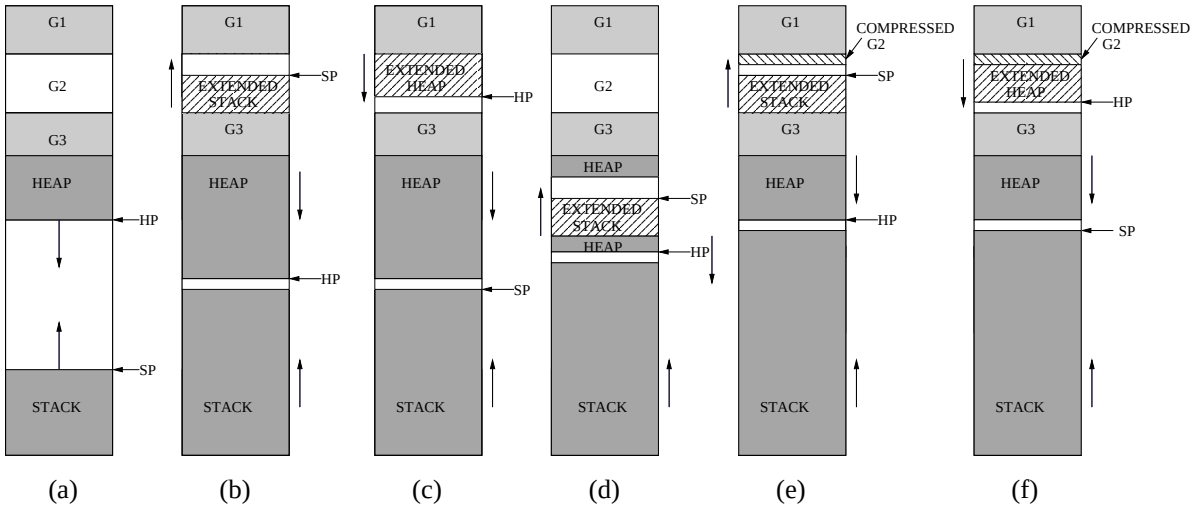


**Figure 1: Memory layouts for our schemes. (a) Normal operation; (b) Overflow stack in dead global *G2*; (c) Overflow heap in dead global *G2*; (d) Overflow stack in free hole in heap; (e) Overflow stack in compressed live global *G2*; (f) Overflow heap in compressed live global *G2*.**

ing overflowing segments, provided the compiler can guarantee that the overflow space will be freed before the dead variable becomes live. Second, if the run-time checks described earlier reveal that the stack or heap is out of memory, then special code is executed to grow the overflowing segment non-contiguously into unused space. The unused space can be dead globals for growing the stack or heap, or free holes in the heap for growing the stack.

The common case overheads of the reuse and compression schemes are reduced by the new *region-merging optimization* described later, and by the rolling-checks optimization inherited from the safety checks. Results shows that the overheads with optimization are low.

**Compressing live data** Our third and final technique for improving reliability compresses live data and uses the resulting freed space to grow the stack or heap when it overflows. The compressed data is later de-compressed before it is accessed. Live data is compressed only after all available dead space is used up for overflow by our reuse technique described above. Currently we investigate compressing globals for growing the stack and the heap.

Figures 1(e) and (f) illustrate how the overflowing stack or heap grows into space freed by compressing live global variables. Figure 1(e) shows the overflowing stack growing into the space freed by compressing the dead global variable *G2*. Figure 1(f) shows the heap growing into the same space.

Let us consider the correctness requirements and performance of our compression scheme. For correctness, the data placed in space freed up by compression must itself be provably dead before the compressed global is accessed again, so that the global can be de-compressed in-place. In-place de-compression ensures that the global is never moved – moving data can complicate its addressing, and can cause incoming pointers to it to become invalid, and so is avoided. For good performance, the global chosen for compression should be one that will not be used for a long time, so that compression and de-compression are infrequent. To be sure, compression and de-compression can be expensive at run-time, but the overheads are incurred only if the system runs out of memory. At that point, any overhead is often acceptable if the alternative is a system crash!

**Discussion** Let us examine whether our schemes can be used in real-time systems. Since the overhead of the safety run-time checks is compile-time-predictable and small, they are easily adapted to any real-time environment. The same is true for our reuse and compression schemes in the common case (*i.e.*, before the system would have run out of memory on a conventional system). The only problematic case is when the reuse or compression schemes are used, and the system has run out of memory, *i.e.*, is using reclaimed space. Here the overheads are less predictable, so hard real-time guarantees are difficult to provide. Soft real-time guarantees are still possible though. In the vast majority of systems, a slow response is better than no response.

At first glance, it appears that a counter argument to our scheme is that simply increasing the amount of physical memory in the system can improve reliability by the same amount as our method does. Although it is true that increasing the amount of memory improves reliability, there are three justifications for our method. First, reliability at any given system cost is improved. Because of the earlier-described difficulties in estimating the memory requirement of an application set, a 100% guarantee of adequate memory is still not possible. By delaying or avoiding the out-of-memory condition, the reliability for any given memory size is significantly improved. A second justification for our method is that sometimes when reusing dead space, our method can provably reduce the memory requirement of the system, which can reduce the size of physical memory needed, and thus its cost. Third, the presence of run-time checks for out-of-memory conditions is a new feature that cannot be substituted by increasing the physical memory size.

Our method has been implemented in a GCC-based compiler for the Motorola MCore [24] processor. Results are collected on a cycle-accurate MCore simulator. For our benchmarks, the run-time and code-size overheads of our scheme of safety run-time checks are measured at 1.1% and 0.09% respectively. Quantitative results, of course, cannot evaluate the benefit of remedial action that our out-of-memory checks allow, which can be invaluable.

We also measure the benefits of the reuse and compression schemes. Results show that 0.7% to 93.5% of the combined stack and heap size can be grown non-contiguously into previously unutilized space, such as space for dead globals, or space freed by compressing live globals. The results do not measure the reduction in the total memory footprint since the primary goal of the method is not to reduce the amount of physical memory. Instead the results measure the reduction in the footprint of the growing segments, which is a more direct measure of reliability. The overhead is higher when reuse or compression is used, but is still low in

the common case when the system is not out of memory – 3.2% in run-time and 2.33% in code-size.

The rest of the paper is organized as follows. Section 2 presents related work. Section 3 describes our scheme for run-time checks for memory overflow protection. Sections 4 and 5 describe our schemes for growing the stack and heap, respectively, into dead global variables. Section 6 describes how to grow the stack into free holes in the heap. Sections 7 and 8 describes how to grow the stack and heap, respectively, into space freed by compressing live global variables. Section 9 explores the choice of data compression algorithm. Section 10 discusses the space requirement of running our overhead routines. Section 11 describes issues in liveness analysis. Section 12 describes experimental results. Section 13 concludes.

## 2. RELATED WORK

In a few high-end embedded systems, a limited form of virtual memory is available [27, 22] that provides memory protection but not swap-space. Unlike virtual memory for desktop systems that gives programmers the illusion of an unlimited amount of available memory, all embedded systems, with or without virtual memory, are inherently constrained by the size of the physical memory [25] because of the typical lack of hard-disks and hence of swap space. As a consequence, even programs running on embedded systems that have memory management hardware and virtual memory, can run out of space. Hence, our techniques for recovering space from the limited amount of memory available, are also valuable for such programs. The added benefit of run-time checks discussed earlier, however, is not applicable to these systems.

On the other hand, most commercial embedded processors, such as [23, 16, 11, 32, 3], do not have virtual memory of any kind. This is because the cost of the hardware memory management units (MMUs) that provide virtual memory has been considered by processor vendors to be excessive in an embedded environment [12]. It is easy to see why: MMUs must contain segment or page tables and their associated logic, which are expensive in area, run-time and power. In such processors, *all* our techniques proposed are valuable since they provide memory protection in software at low cost and also some ability to reclaim dead space in the case of an overflow.

We are not aware of any method that uses software run-time checks for out-of-memory errors, or that reuses space in another segment when one segment is full.

Similar run-time checks to ours have been proposed in [5], though in a completely different context, and with a different goal. The run-time checks in [5] are used to implement a stack management scheme that allows high-concurrency desktop servers to support threads without allocating a large contiguous portion of the virtual memory for their stacks. Instead a thread's stack is allocated in a small fixed-size heap chunk, and is grown discontinuously into other heap chunks when one is full. Our system differs from theirs in the following seven ways. First, our system is applied, optimized and evaluated for embedded systems and for a different goal of detecting out-of-memory errors. Second, our method works with any existing stack layout, while their method requires a change in the stack layout to treat it more like the heap, in that it consists of un-ordered fixed-size chunks that are dynamically allocated. Third, our scheme does not incur the extra overhead of discontinuous stack growth unless the system is out of memory, which is rare, while their scheme would incur that overhead whenever the small fixed-size chunks run out, which is more common. Fourth, our run-time checks consider heap growth in deciding if the stack is running out of memory. This is not needed when using fixed-size heap chunks for stack, but is needed when the stack and heap can grow into each other, as is possible in the general case. Fifth, our scheme can handle virtual function calls, essential to handle object-oriented languages, while their scheme does not apply to such languages. Sixth, our reuse scheme can reclaim the space in dead global variables, which is not their goal. Seventh, our evaluation measures the impact on code-size which is important for embedded systems, while they do not, given their focus on desktop servers.

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

**Figure 2: Pseudo-code for safety run-time checks.**

A different approach to increasing the amount of space available to a program is garbage collection [4, 6], whose primary goal is to reclaim unreachable heap objects. Recently, traditional garbage collection techniques have been adapted to embedded environments [18, 8, 21]. Of our five techniques, however, four attempt to recover space from the global segment, which is not addressed by garbage collection. A further distinguishing feature of our work is that we provide run-time checks for reliability which is not a feature of garbage collection. In essence, garbage collection also reduces the memory footprint of a program, but by using a different approach, and hence is complementary to our scheme.

Compression of program data [35] has been discussed in the context of heap structures to reduce the memory footprint and hence the cost of embedded systems. The overall goal of our technique, on the other hand, is increasing the reliability of the system by smoothly transitioning to a reuse mode in case of a memory shortage. Other techniques such as compression and compaction of embedded code [31, 19, 14] (not data), reduce the amount of ROM required and these schemes, as such, are orthogonal to ours.

## 3. SAFETY RUN-TIME CHECKS FOR OVERFLOW PROTECTION

This section describes our light-weight, software-only scheme for detecting out-of-memory errors. To see how out-of-memory errors can be detected, consider that the stack grows only at procedure calls, and the heap grows only in dynamic memory allocation routines such as *malloc()*. It follows that a baseline un-optimized scheme involves simply inserting a run-time check for overflow at each procedure call and each *malloc()* call. In the rest of the paper, *malloc()* is used as shorthand for any dynamic memory allocation routine.

The safety run-time checks are implemented as follows. First, for heap checks, if the *malloc()* finds that no free chunks of adequate size are available then an out-of-memory error is reported. Such a check is nothing new since it exists by default in most versions of *malloc()*, and thus it adds no overhead. Second, consider the stack checks which are inserted at each procedure call. These are new and add run-time overhead. They work as follows. 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. Without loss of generality, if the stack grows into lower addresses, then if the new stack pointer is less than the stack's allowable boundary, an out-of-memory error is flagged and handled safely. The boundary for the stack could be either (i) the heap pointer, if the heap adjoins the growing direction of the stack; or (ii) the base of the adjoining stack, if another task's stack adjoins the growing direction of stack; or (iii) the end of memory, if the stack ends at the end of memory. Which of these three

```
void          do_rolling_optimization() {
1.    Sort all procedures in decreasing order of number
          of calls to each procedure in the profile data
2.    for (each procedure Curr_Proc in sorted list)
3.        can_roll_to_all_parents ← true
4.        for (each parent P of Curr_Proc)
5.            if (!can_roll(Curr_Proc, P, Curr_Proc))
6.                {can_roll_to_all_parents ← false; break}
7.        if (can_roll_to_all_parents)
8.            for (each parent P of Curr_Proc)
9.                roll_check(Curr_Proc, P)
10.   return


boolean       can_roll(Curr_Proc, Ancestor, Ancestor_Child) {
11.   if (call to Curr_Proc is virtual function call)
12.       return (false)
13.   if (there is any heap allocation in Ancestor before calling
              Ancestor_Child for the LAST time in Ancestor)
14.       return (false)
15.   if (either Curr_Proc or Ancestor recursive but not both in same cycle))
16.       return (false)
17.   if (Curr_Proc == Ancestor)
18.       return (false)    /* Termination for recursive cycles */
19.   Longest_path ← Path in call graph from Ancestor to
          Child, not including Child, with largest sum of
          stack frame sizes among all such paths
20.   Sum_stack_size ← Sum of stack sizes along Longest_path
21.   if (Sum_stack_size > 10% of max. stack + heap size in profile)
22.       return (false)
23.   if (Rolled_size [Ancestor] == 0)        /* No check on Ancestor */
24.       for (each parent P of Ancestor in the call-graph)
25.           if (!can_roll (Curr_Proc, P, Ancestor))
26.               return (false)
27.   return (true)      /* Can roll check from Curr_Proc to Ancestor */


void          roll_check (Curr_Proc, Ancestor) {
28.   if (Curr_Proc == Ancestor)
29.       return (false)    /* Termination for recursive cycles */
30.   if (Rolled_size [Ancestor] == 0)
31.       for (each parent P of Ancestor in the call-graph)
32.           roll_check (Curr_Proc, P)
33.   else {            /* Can roll check from Curr_Proc into Ancestor */
34.       Longest_path ← Path in call graph from Ancestor to
              Child, not including Child, with largest sum of
              stack frame sizes among all such paths
35.       Sum_stack_size ← Sum of stack sizes along Longest_path
36.       Rolled_size [Ancestor] ← max (Rolled_size [Ancestor],
                  Sum_stack_size + Rolled_size [Curr_Proc])
37.       Rolled_size [Curr_Proc] ← 0
38.   }
39.   return
```

**Figure 3: Pseudo-code for rolling checks optimization**

cases to use is known at compile-time and thus the compiler uses the correct boundary in the compiled code. Figure 2 shows what the safety run-time check code looks like for the stack checks; the heap checks are not shown.

The above scheme is un-optimized, but we can reduce the overheads of the added stack checks by the *rolling checks optimization*. The intuition behind the optimization can be understood by the following example. If a parent procedure calls a child procedure, then instead of checking for stack space at the start of both procedures, it might be, in certain cases, 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*. In this way, the check for the child is 'rolled' into the check for the parent, eliminating the overhead for the child. If the child is called more frequently than the parent, the reduction in overhead can be more than half. Thus, given a choice, it is more important to roll checks out of frequently called child procedures than out of less frequent procedures.

There are several issues that complicate the above simple picture of the rolling checks optimization, which must be taken into account. First, 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. This is because when the parent is called, it is impossible to guarantee enough space for the child since the heap could have grown in the meantime cutting into the space available for the child. Thus the rolling optimization is not done in this case. Second, 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 the exact identity of the child is unknown at compile-time. Third, 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. In that case, rolling the child's check to the parent may declare the program to be out of memory when in reality it would not have been. To avoid this effect from becoming too pronounced, we 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. This guarantees that a premature out-of-memory declaration can happen only when the space remaining is less than 10% of the maximum stack + heap requirement. Fourth, rolling checks can be permitted inside of recursive cycles in the application program, but not out of recursive cycles since every time a parent procedure is called, its child procedure can be called multiple times if it is recursive.

Figure 3 shows the complete pseudo-code for the rolling checks optimization, taking into account the issues mentioned above. Too involved to describe in detail, we briefly outline the pseudo-code here. Routine **do_rolling_optimization**() is the highest-level routine for the optimization. It considers rolling checks in the order of their frequency. In order to roll a check, it first ensures that the check can be legally rolled to all its parents (lines 3-6), before it actually rolls the checks to its parents (lines 7-9). Routine **can_roll**(), shown next, is a recursive routine that checks if the current procedure can be rolled in to the Ancestor (both arguments to **can_roll**()). It handles the exceptions mentioned earlier that prevent rolling for virtual functions (line 11-12), heap allocations (line 13-14) and pre-mature declarations (lines 19-22). It also handles recursive functions in the application as outlined earlier (lines 15-18). Finally lines 23-26 check if the parent already had its check rolled; if so, the child recursively checks (line 25) whether it can roll its check to the parent's parents (*i.e.*, its grandparents).

Routine **roll_check**() takes a similar recursive approach of rolling the checks from the current procedure up to its ancestors (lines 30-32). It does not need to check rolling-preventing exceptions, as those have been checked already in **can_roll**(). The primary termination condition of the recursion is when the parent has a check on it, and hence the rolling can be done to it (line 33-38). The Rolled_size variable for each procedure initially stores the size of the frame for that procedure. When a check is rolled, the Rolled_size is set to zero for the child, and to the sum of the parent and child frame sizes for the parent. Care is taken that if a parent has multiple children, then the Rolled_size is set to be the maximum needed across all its children (line 36).

The rolling checks optimization is effective in eliminating much of the overhead of the safety run-time checks. More details are in the results section.

# 4. REUSING GLOBALS FOR STACK

Our scheme of reusing globals for stack allows the program's stack to grow into the global segment when it is detected that the system is running out of stack space. This is implemented by the following two tasks. First, the compiler performs liveness analysis to detect dead global arrays, if any, at each point in the program.


**Figure 4: Example showing (a) a program outline; and (b) is its DPRG showing nodes, edges & timestamps.**

Second, in case the safety run-time checks described in section 3 find that the stack is out of memory, our scheme selects one of the global arrays that is dead at that point, and grows the stack into it.

**Identifying dead globals** Depending on where in the program's execution the stack ran out of space, a different global array is chosen to grow the stack into. The method of choosing the global to grow into has the following three steps. 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[1]. This deadness constraint ensures that none of the functions pushed on to the global variable portion of the stack access the global array, and thus the global array remains dead during the life of those stack functions, allowing reuse. 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. Since the list is sorted at compile-time in decreasing order of size, this chooses the largest dead global to grow into. An implementation detail is that for the program to look up the list for the current region, it must know what the current region is. Thus the compiler inserts a current- region variable into the program which is assigned a new value each time a new region is entered. This new **per-region reuse code** is shown in figure 5(i).

A good choice of regions should satisfy the following three criteria. First, the regions should be short enough to be able to closely track the Reuse Candidate List preferences of different program points. Second, the regions should be long enough that the run-time overhead due to code inserted at the start of every region remains a small fraction of the total run-time. Third, it is desirable if the regions can be numbered at compile-time in the order of their run-time execution. Such a static run-time ordering does not help in this section, but will help later in section 5 while growing the heap into dead global variables.

The following heuristic choice of regions satisfies all the above criteria: *every static loop beginning and end, and function beginning and end, marks the entry into a new region.* Each region continues until the start of the next region in run-time order. Figure 4(b) illustrates the choice of regions for the code in figure 4(a). The figure shows the start of the regions numbered with *timestamps* 1 to 14. The timestamp to the left of a node depicts its beginning, and the timestamp to its right depicts its end. Timestamps depict the run-time order of those points in a compile-time data structure.

[1]Such liveness analysis is possible even for situations where the call-graph is not fully known. See section 11 for details.

More formally, figure 4(b) is the *Data-Program Relationship Graph (DPRG)* for the code in figure 4(a). The DPRG is a compiler data structure that was first proposed in [33], which we adopt in our work. It consists of the call-graph of the program appended with nodes for loops and variables connected in the obvious manner depicted in the figure. The timestamps (1-14) are obtained by a depth-first search (DFS) of the DPRG, which numbers each region in the order in which they are visited during traversal. Interestingly, the timestamp order is the run-time order of the regions. Recursion is handled by collapsing recursive cycles in the DPRG into a single node before DFS; such a node is therefore assigned a single timestamp during DFS. The collapsed node is thus a single region, and is handled as any other. More details on the properties of the DPRG are available to the interested reader in [33], but are not essential to the understanding of this paper.

**Region-merging optimization** One optimization we perform to reduce the overhead of regions is to merge regions whenever possible. In particular, if two regions that are executed consecutively at run-time are such that they have the exact same Reuse Candidate Lists, they are merged into a single region. This process is repeated until the minimal set of regions, each with a distinct Reuse Candidate List, is obtained. This ensures that the overhead from code inserted at the entry into regions is minimized, without sacrificing the best choice of the Reuse Candidate List per region.

**Growing stack into globals** Once the out-of-stack condition is detected by the safety run-time checks, growing the stack discontinuously into the dead global array is done by changing the stack pointer to the end address of the array. Further calls occur as usual, and procedure returns need no modification since the return address is recovered from the current procedure's stack frame. The return address, when recovered from the stack frame, is correct since the stack pointer is updated to reflect the address of the global array only after the original stack pointer value has been saved in the current procedure's frame.

Growing the stack into globals is implemented by augmenting the safety check code, which detects the overflow, with code that performs the reuse for that region. Figure 5(ii) shows the augmented code. To understand the code, consider that a new global boolean variable called Reuse-Started, initialized to false, is inserted in the code by the compiler. The first time the stack overflows (first part of line 1), Reuse-Started is set to true(line 3), and the stack pointer is changed to the end address of the first element on that region's Reuse-Candidate-List (lines 4-5), which achieves the discontinuous growth. Otherwise, if Reuse-Started is true, *i.e.*, the stack is currently in overflow mode, (lines 8-17), the stack overflow check is repeated with the new boundary of the global array (line 8), since the original check on line 1 is no longer correct. If the

**PER-REGION REUSE CODE** **(i)**

```
1.  Current-Region ← CURRENT_REGION_CONSTANT_ID
```

**SAFETY CODE AUGMENTED WITH REUSE CODE FOR THAT REGION** **(ii)**

```
1.  if ((Stack-Ptr < ORIGINAL_BOUND + Space needed by reuse routines) OR (Reuse-Started)) {
2.      if (!Reuse-Started) {
3.          Reuse-Started ← 1
4.          Current-candidate ← Head of Reuse-Candidate-List[Current-Region]
5.          Stack-Ptr ← Current-candidate→base-address + Current-candidate→size
6.      }
7.      else {
8.          if (Stack-Ptr < Current-candidate→base-address + Space needed by reuse routines) {      /* Stack Overflow */
9.              Current-candidate ← Next element of Reuse-Candidate-List[Current Region]
10.             Stack-Ptr ← Current-candidate→base-address + Current-candidate→size
11.         }
12.         if (Stack-Ptr > (Current-candidate→base-address + Current-candidate→size)) {      /* Stack Underflow */
13.             if (Current-candidate == Head of Reuse-Candidate-List[Current Region]) {
14.                 Reuse-Started ← 0
15.             else
16.                 Current-candidate ← Previous element of Reuse-Candidate-List[Current Region]
17.         }
18.     }
19. }
```

**Figure 5: Pseudo-code for inserted safety run-time checks augmented for reuse.**

stack has overflowed this global array, it is discontinuously moved to grow into the next global array in the Reuse-Candidate-List of that region (lines 9-10). If there is no next element on line 9, (code not shown), we are out of memory.

Lines 12-17 handle the case when the array had overflown, but has now retreated to the original space. If the retreat is from the first global array in the Reuse-Candidate-List (line 13), then we go back to the original stack space and reset Reuse-started to false (line 14), otherwise we go back to the previous global array.

The overheads for reuse are larger than those for safety checks alone in three ways. First, figure 5(i) shows that the run-time overhead for the start of regions without a safety check is one scalar assignment. Second, figure 5(ii) shows that the safety check is augmented so that in the common case when the system is not out of memory, the additional run-time overhead is that the **if** condition on line 1 has an extra OR with a boolean variable Reuse-Started. The body of the **if** (line 2-18) is not executed in the common case. Third, the code-size overhead from figure 5(ii) is modest since the entire body of the **if** statement (line 2-18) is moved to a procedure that is called repeatedly from each modified safety check instance in the program. The results section shows that the overheads with reuse remain small.

# 5. REUSING GLOBALS FOR HEAP

This section describes how our scheme of reusing globals for stack, described in section 4, is extended to allow reuse of global variables for heap data as well. This is achieved by adding the dead global arrays to the heap free-list when the heap is full. The extended scheme leverages the framework built earlier, which includes dividing the program into regions, adding a variable to keep track of the current region, performing liveness analysis to detect dead global arrays and building Reuse Candidate Lists per region.

Growing the heap into dead globals entails implementing the following three additional tasks beyond the ones for growing the stack. First, the Reuse Candidate Lists are sorted at compile-time by next-time-of-access and size, rather than by size alone, such that the dead global array that comes alive farthest into the future is placed at the head of the list. The size is used as a tie-breaker: if there are two arrays that come alive at the same time, the larger is placed earlier in the list. 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. Third, the compiler inserts the Out-of-Heap Function in the code; it selects the candidate at the head of the current region's Reuse Candidate List, and adds it to the heap free-list. The code for these three tasks is not shown, but each is elaborated upon below.

**Sorting Reuse Candidate Lists** To see why the individual Reuse Candidate Lists need to be resorted on the basis of next-time-of-access of the dead global arrays, consider the difference between growing the stack into dead globals versus growing the heap. The difference arises because stack frames have predictable lifetimes and are automatically popped off the stack once the corresponding functions exit. Thus it is easy to guarantee that the extended stack will be popped off by the time the dead global becomes live again. In contrast, liveness analysis for heaps is difficult. Even if heap objects are freed, it is difficult to prove that all objects allocated at a site, and not just some, have been freed. Consequently, there is no guarantee that the extended heap structure will be dead by the time the global array that it was growing into becomes live again.

Given the difficulty in liveness analysis for heaps, in case the dead global occupied by the extended heap becomes live, our scheme does a run-time check to see if the extended heap has been freed, immediately prior to the the global coming back to life. If the extended heap is empty then the program runs successfully. If the extended heap is not empty, then we declare that we are out-of-memory. In this last case, the out-of-memory condition is postponed but our method fails to prevent it. Finally, if there is a dead global that remains dead for the remaining lifespan of the program, then that variable is selected to grow the heap, and no further run-time check is needed to guarantee correctness.

We can now see why the dead globals in the Reuse Candidate Lists are sorted in decreasing order of next-time-of-access. The later the global variable comes back to life, the greater is the probability that the run-time check, discussed above, would succeed. Thus the chance of success increases, if globals that come alive later, are chosen first, to grow into.

The next-time-of-access of the dead global variable is estimated at compile-time using the DPRG timestamps described in section 4 as follows. Initially, for the current region, the set of later regions is computed as the union of two sets: (i) all regions with a greater timestamp than the current region, and (ii) all regions that are *descended* from the loop node L closest to *main()* on the DPRG path from *main()* to the current region. A node is descended from L if there exists a path from *main()* to the node through L. If there are no loop nodes on the path then this latter set is empty. Using these two sets, the next timestamp of access of the global variable is com-

puted as *the next timestamp in the common case ordering of the set of later regions*, keeping in mind that the common-case ordering of nodes descended from loops follows the loop's backward branch.

**Modifying malloc** The second task needed for growing the heap into dead globals is to modify the *malloc()* library function (or other dynamic memory allocation routines). *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, which is described in detail in the next paragraph. This task simply involves replacing the return statement in *malloc()* with a call to the Out-of-Heap Function, and since this call is executed only when the program has actually run out of heap space, there is no overhead in the common case of when the program is not out of memory.

**Out-of-Heap Function** The Out-of-Heap function is called from *malloc()* when it is out of heap space and does the following three tasks. First, it looks up the Reuse Candidate List corresponding to the current region and selects the dead global array at the head of the list. Second, it creates a *malloc()* chunk header at the start address of the selected global array so as to make it look like a usual heap chunk obtained by calling *malloc()*. The *malloc()* chunk header is standard in most language implementations – it includes information on the size of the chunk and whether the chunk is currently in use. Third, the Out-of-Heap function calls the free library function with a pointer to this global array, which places this chunk in the appropriate heap free-list bin, based on its size.

Two advantages of the extended scheme described above, are as follows. First, it is based on the same framework as the original scheme of reusing globals for stack, and requires no additional data-structures. Second, it has no extra run-time overhead in the common case, as explained earlier.

# 6. REUSING HEAP FOR STACK

When the program is out of stack space, another possibility is to grow the stack into free holes inside the heap, if available. Implementation is done by inserting additional code (not shown) in the existing check for whether the stack is out-of-memory in figure 5(ii). When the stack is out-of-memory, the code first tries to grow the stack into dead globals as described earlier; only after those are full is the stack 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. The special *malloc()* call returns the free hole of the largest available size, or of the compiler-estimated size of the remaining stack, if known, whichever is smaller. The free hole of the largest size is readily available in most widely used *malloc()* variants, which usually store the holes in lists of increasing power-of-two hole sizes [20].

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. Garbage collection is usually not done in imperative languages such as C and our technique of reusing heap for stack is useful in such environments. In systems that do heap compaction, however, the reusing heap for stack component of our technique is not useful and should be turned off.

# 7. COMPRESSING GLOBALS FOR STACK

When the program is out of stack or heap, it is possible to free up even more space by compressing live global variables, and growing the stack or heap into the resulting free space. Live data is compressed only after all available dead space is used up for overflow by our reuse technique described above. The compressed data is later de-compressed before it is accessed. For good performance, the global chosen for compression should be one that will not be used for a long time, so that compression and de-compression are infrequent. The choice of the actual data compression algorithm to use is explored later in section 9.

This section describes how the freed up space from compression can be used to grow the stack. The scheme is similar to the method described in section 4 for growing the stack into dead globals. In particular, it uses the same set of regions, the same method to detect the Out-of-Stack condition and the same mechanics for growing the stack discontinuously into the global segment.

The implementation of this scheme differs from the scheme for growing the stack into dead globals in the following three ways. First, the reuse candidates are extended to include live global arrays. Second, 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. Third, 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. The rest of the section describes these three modifications in detail.

**Extending the Reuse Candidate Lists** In order to have more reuse candidates per region, we extend the definition of a reuse candidate - a global array is a reuse candidate for a region if the array is not accessed throughout that region, and is not accessed in any of the functions called directly or indirectly from that region. This condition of no-access is a relaxation of the earlier-mentioned condition for growing into dead globals, where the requirement was that the variable is dead in the same regions. Satisfying this no-access constraint guarantees that when the overflow stack is live, the compressed global is not accessed. Conversely, when the compressed global is accessed again, it can be de-compressed in-place since the portion of stack that had overflowed is guaranteed to be popped off by then. In-place de-compression ensures that the global is never moved – moving data can complicate its addressing, and can cause incoming pointers to it to become invalid, and so is avoided. In implementing this constraint, finding the variables accessed in a certain region is possible even in the presence of pointers by using a pointer analysis [30, 9] scheme to find the list of all variables a pointer-based reference could access.

**Triggering compression** Once the reuse candidates per region have been determined, the process of compression is triggered when needed. To implement this, the reuse candidate lists are sorted as before, but an extra field is added to each candidate to indicate whether it is dead or live. In addition, the code inserted for when the stack is out of memory, shown in figure 5(ii), is extended as follows (modifications not shown). First, it selects the candidate at the head of the current region's Reuse Candidate List and checks whether it is dead or alive. Second, in case it is dead, it simply extends the stack into it. Third, in case it is alive, it calls a compression routine that compresses the global array in-place, makes an entry in a Compression Table storing the start address and compressed size of the array, and moves the end address of the global array into the stack pointer register. Finally, after compression, the stack pointer is checked against the end address of the compressed array, rather than its base address (line 8).

**Triggering de-compression** In order to trigger de-compression when needed, the compiler augments the code at the start of every region. Figure 6 shows this additional code which is added to the codes in both figures 5(i) and (ii). It ensures that if reuse has

```
ADDITIONAL PER-REGION CODE WITH COMPRESSION
1.  if (Reuse-Started) {
2.      for (each global array GA used in region CURRENT_REGION_CONSTANT_ID and that is currently compressed)
3.          De-compress GA in its original location
4.  }
```

**Figure 6: Extra pseudo-code for compression added to figures 5(i) and (ii).**

started, then all compressed global arrays accessed in the following region are de-compressed in their original locations (line 3). To find which arrays are compressed, it looks up each global array (code not shown) in the Compression Table mentioned above. If there is no entry corresponding to that array, it implies that the array is not compressed and can safely be accessed in this region. If a matching entry is found, the start address and compressed size of the array are looked up from the Compression Table, and the array is de-compressed in-place. In the case of figure 5(ii) the added code does not increase the common case overhead since it can be placed inside the body of the **else** part on line 7. The code is added only when the compression is employed for an application.

The above scheme of compressing live global arrays and reusing the space for stack creates many more opportunities to reuse space. Moreover, the additional common case overhead of this scheme is negligible when compared to the basic scheme, both of which are low. The overhead when compression is done is high, but is incurred only when the system would have otherwise crashed. At that point, anyone would prefer a slow system to a crashed system.

# 8. COMPRESSING GLOBALS FOR HEAP

The final scheme we present is to grow the heap, when it is out-of-memory, into the space freed by compressing live global variables. It is implemented by combining parts of two earlier schemes: the method to grow the heap into dead globals in section 5, and the method to grow the stack into compressed live globals in section 7. It has the following three components. First, it uses the same Reuse Candidate Lists as section 7, that are sorted according to the next-time-of-access of the global arrays, as described in section 5. Second, once the system has run out of heap space, it makes a call to the Out-of-Heap Function, discussed in section 5, which is now slightly modified to support compression. The modification involves selecting the candidate at the head of the current region's Reuse Candidate List, and instead of directly calling a free on that array, first checking to see if the candidate is live. If that is indeed the case, it first compresses the global array in place, exactly the way it was described in section 7, including maintaining book-keeping information in the Compression Table, and 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, just as in section 7. If it has, all compressed globals are de-compressed as in that section. The only additional task needed before de-compression is that the overflow heap is checked to see if it is empty, like in section 5, and if it is not, an out-of-memory error is declared.

Since this scheme is a combination of existing technologies, it does not use any new data structures and has the same run-time overhead as the older scheme of compressing globals for stack.

# 9. COMPRESSION ALGORITHM

Since sections 7 and 8 involve compressing global arrays, a data compression algorithm is needed. For our situation, a good compression algorithm is one that has the following characteristics. First, it should compress program data to a high degree, so that a significant amount of free space is recovered. Second, it should have a very low or zero persistent memory overhead, which is the extra book-keeping space, if any, needed by the compression algorithm that persists until de-compression. Persistent storage is undesirable since it reduces the net space freed by compression. Third, since compression is done at run-time, the sum of the compression and de-compression times should be small.

We explored the following three compression techniques, all of which roughly satisfy the above criteria: (i) LZO, a modern implementation of the Lempel-Ziv dictionary-based compression algorithm [29]; (ii) WKdm, which uses a combination of dictionary-based and statistical methods and is characterized by a very small dictionary size [34] and (iii) WKS, a modified version of WKdm that supports in-place compression and de-compression, without having to copy data to an intermediate buffer [28].

Upon detailed evaluation, we chose WKS because it has (a) no persistent memory overhead, (b) has the best compression ratio when tested on global variables, and (c) requires a low number of cycles for compressing and de-compressing the data. For instance, we evaluated global data compression in block sizes ranging from 16 bytes to 8 KB. The average amount of space freed up by WKS is about 60% of the uncompressed space, and compression and de-compression took an average of 43 cycles per word compressed. Further details are not presented here for lack of space, but can be found in a technical report [28].

# 10. SPACE OVERHEADS OF ROUTINES

This section discusses the main memory space required to run the added routines for our reuse and compression methods (no added routines are needed for the optimized scheme of run-time checks). Space is needed for the following two reasons. First, calls are made to certain functions such as the Out-of-Heap Function (sections 5 and 8), the compression and de-compression functions (sections 7 and 8). Each of these functions requires some space on the stack. To ensure correct execution, the application cannot wait until the stack is full to make these calls; instead the application must make the calls when there is just enough space on the stack to make these calls, but no more. Their stack space is not wasted in the final analysis since our overhead routines are exited and their stack frames are popped off by the time they return to the application program, which can thereafter reuse the space. Nevertheless to limit the pre-mature invocation of our method, special care is taken in writing our functions to ensure that their stack space is small.

A second source of memory overhead from our schemes is to store the Reuse Candidate Lists for every region in the same memory device where program code is stored, which is usually read-only memory (ROM) in embedded systems. The reuse candidate lists can be stored in ROM because they are known at compile-time, and do not change at run-time. Results show that the lists typically are only a tiny fraction of the program code-size, and do not significantly change the required code-size.

# 11. LIVENESS ANALYSIS

Liveness analysis, needed for our reuse schemes for detecting dead globals, is a well-established dataflow analysis in the compiler literature [2]. It is always possible even in languages with pointers by using pointer analysis. The less precise the pointer analysis, the more conservative the liveness analysis, but it is never wrong.

| 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.**

A difficulty arises in doing compile-time liveness analysis in situations when the call-graph for the program is not fully known at compile-time. There are two situations when the call-graph may not be known at compile-time. 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. 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.

Fortunately there are technologies that allow liveness analysis even when the call-graph is not fully known. 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 has been investigated in [26]. Restricting the set of functions a virtual function may call, is possible at compile-time, in many cases, by using techniques such as [10] which use type information to narrow down what functions can be called. Even when the call set cannot be restricted to one, a conservative analysis is possible which considers if a variable can be live under any of the functions in the restricted set. For imperative languages such as C, which is the most widely prevalent language in embedded systems, unknown call-graphs are rare since first-order functions are rare [13], and hence this problem is mostly absent.

## 12. RESULTS

This section presents results for the different schemes proposed in the paper. The proposed techniques have been implemented in the public-domain GCC cross-compiler targeting the Motorola M-Core [24] embedded processor. The compiler is modified to automatically determine the program regions and reuse candidates for each region. Automating the code insertions, however, is not yet complete and therefore the current implementation involves manually inserting the required check code into the application sources at the beginning of functions and at the start of regions. Since the resulting executable code is exactly the same as what will be produced by automating the code insertions, manual coding causes no error of any kind in the results. One of the schemes, namely growing the stack into heap fragments has not yet been implemented; but the remaining techniques - safety runtime checks, reusing global for stack, reusing global for heap, compressing global for stack and compressing global for heap - have been implemented. Finally, the compiled applications are executed on the public-domain cycle-accurate simulator for the Motorola M-Core.

The names, sources and other characteristics of the embedded benchmarks evaluated are shown in table 1. The benchmarks selected are such that they have at least some global arrays each, since four out of the five reuse schemes proposed rely on recovering space from global arrays. Owing to the tedious nature of manually inserting code, the benchmarks chosen are such that they demonstrate the merits of the technique, without being too large to modify manually. One benchmark (JPEG) that is not favorable to our technique is also included.

| **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**

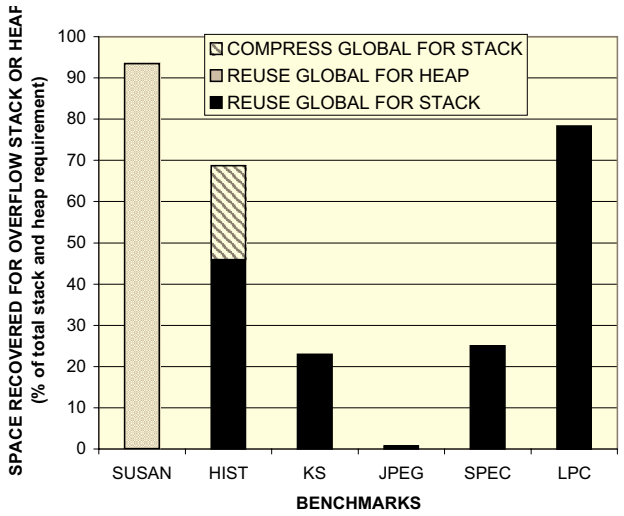


**Figure 7: Extra space recovered for stack and heap as a fraction of total stack and heap requirement for each benchmark**

**Safety Runtime Checks** Table 2 shows the overheads due to inserting the safety checks alone. The second column reports the run-time overhead without any optimization, whereas the third column records the reduced run-time overhead after applying the rolling check optimization proposed in section 3. The run-time overhead reduces from 2.5% to 1.1% with optimizations, and the code-size overhead with optimization is only 0.09%. Recall that the safety runtime checks is a stand-alone scheme that can be used with or without the reuse and compression schemes. Results show that their guaranteed detection of out-of-memory errors, thus allowing remedial action, is possible with very low overhead.

**Reuse and compression benefits** Figure 7 shows the improvement resulting from the use of our reuse and compression techniques for each benchmark. Since the goal of the scheme is to enhance the reliability of the system by providing additional memory in case of a space shortage, the improvement numbers on the y-axis have been expressed as percentages of the total dynamic (stack + heap) memory requirement of the system. The figure shows that the improvements range from 0.77%, in the case of JPEG, to 93.5% in the case of SUSAN. In other words, for SUSAN 93.5% of the maximum stack and heap combined usage can be placed in dead global arrays in case of a memory overflow.

| 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**

The above numbers are collected as follows. The program is first executed with an extremely large stack and heap space in order to determine the exact stack and heap footprints for a particular input data-set. Thereafter, the program is re-run with a heap and stack space that is less than the requirement determined in the first pass and it is observed whether the program can execute correctly. This process is repeated several times, with progressively smaller amounts of dynamic memory, until even the space freed up by our techniques is not enough to allow the program to run to the end. In KS, for instance, the program runs to completion even with a dynamic memory size that was 23% less than the actual dynamic memory requirement calculated in the first pass.

The significant space recovery shown in figure 7 for several benchmarks shows the promise of the method in improving system reliability. When the program is out of memory, the recovered space can be used to postpone and hopefully avoid a system crash. In this manner, the techniques improve reliability for a given memory size, and hence reduce the dollar cost of the system. The numbers under-estimate the benefits from the technique in two important ways. First, the implementation of the technique for growing the stack into free holes in the heap is not yet complete, and hence its improvements are not counted. Second, numbers cannot quantify the additional safety and reliability benefits from automatic detection of out-of-memory errors made possible by our method, which enables remedial action of various kinds.

The figure 7 also shows the contribution of the different reuse schemes to the total space recovered for each benchmark. Reusing globals for stack appears to be the most promising because predicting the lifetime of stack variables at compile-time is easier than doing the same for heap variables and also because three out of the six benchmarks, namely HISTOGRAM, LPC and SPECTRAL, do not have any heap allocation.

Some additional benchmark specific observations are as follows. For SUSAN, the space recovered is substantial since it has one 360 KB array which is used only when a specific option is chosen by the data set. In case a different option is chosen, the array in not used at all, and is automatically freed for heap usage by our scheme. The 360 KB array referred to above is actually declared on the stack in the *main()* procedure, and is retained on the stack throughout the lifetime of the program. Our compiler implements a simple optimization which promotes all arrays in *main()* to global variables so that our method can benefit from them. HISTOGRAM, LPC and SPECTRAL are selected because each of them uses global arrays with mutually exclusive lifetimes, thereby presenting opportunities for benefiting from our techniques. While all space freed up to the stack in LPC and SPECTRAL are from reuse of dead global arrays, in HISTOGRAM, one of the arrays in the candidate list is live throughout, making its reuse impossible. However, the array is not used throughout and thus, *compression* is feasible and is automatically invoked. The low improvement in JPEG resulted from it being extremely heap-intensive and having large heap structures whose compiler-derived live ranges spanned the entire program. The small benefit arose from reusing some global space for stack.

**Reuse and compression overheads** Table 3 shows the increase in run-time and code-size caused by our reuse techniques. The increase in run-time is incurred due to the insertion of the reuse checks. Recall from figure 5 that the reuse code is more expensive than the safety check since it has two predicates OR-ed together, and because of the assignment of the Current-Region variable at the start of regions. Our rolling check and region-merging optimizations however, reduce the run-time increase significantly. The optimized run-time overhead is 3.2% on average, which is higher than for the safety checks, but is still low.

Table 3 also shows the increase in code-size in its last two columns for the optimized case. Code size is increased from two components - an application-specific part from the inserted run-time checks, and a fixed part from the same extra handler routines for our method linked into all applications. The fixed part is the same for all benchmarks, except HISTOGRAM, for which it is higher because it also uses compression and de-compression routines. Table 3 shows that the application-specific increase in code-size is almost insignificant – only 0.26% on average for our benchmarks. Table 3 also shows that the fixed code-size increase is 2.07% on average for our benchmarks; this number is expected to be much smaller for real embedded systems, which typically have much larger applications than our benchmarks. Further the fixed size routines in our method have not currently been carefully engineered during their programming to reduce code size; we expect that programming them carefully will reduce the fixed code-size overhead further from the already low 2.07% number.

Currently the Reuse Candidate Lists are placed in heap instead of ROM for implementation convenience, and hence their code-size is not counted in table 3. We do, however, count their impact in the earlier experiment in fig. 7, when their space is subtracted from the space saved and only the net space recovered is reported. When the candidate lists are placed in ROM, we have computed that their impact on code-size will be less than 0.5%.

# 13. CONCLUSION

This paper presents a flexible memory management method for embedded systems whose main goal is to improve the reliability of such systems in case of out-of-memory errors. It proposes three techniques for providing reliability. The first technique is to modify application code automatically in the compiler to check for all out-of-memory conditions. Such a system of software-only run-time checks can be invaluable in embedded systems without memory protection. This is a stand-alone technique that can be implemented without the remaining techniques, if desired. The second technique is to reduce the memory footprint of the program by allowing segments that are out of memory to grow into non-contiguous free space in the system, when available. The third technique involves compressing live data and using the resulting free space to grow the stack and the heap when they overflow. Results show that the overhead from the system of run-time checks is very low. The additional space recovered by the schemes for reusing dead space and compressing live data, ranges between 0.7% to 93.5% of the com-

bined stack and heap size for our benchmarks. In future work, we wish to explore the opportunities of reusing space across tasks in multitasking environments.

## 14. REFERENCES


[1] High availability design for embedded systems. Technical report, Wind River, Inc. http://www.windriver.com/-whitepapers/high_availability_design.html.

[2] Andrew W. Appel and Maia Ginsburg. *Modern Compiler Implementation in C*. Cambridge Univ. Press, January 1998.

[3] *Atmel Microcontrollers based on 8051 Architecture*. http://www.atmel.com/products/8051.

[4] Joel F. Bartlett. Compacting Garbage Collection with Ambiguous Roots. Technical report, DEC Western Research Laboratory, Palo Alto, CA, February 1988.

[5] Rob Von Behren, Jeremy Condit, Feng Zhou, George Necula, and Eric Brewer. Cappricio: Scalable Threads for Internet Services. In *Proc., ACM Symposium on Operating Systems Principles (SOSP)*, October 2003.

[6] H-J. Boehm and M. Weiser. Garbage Collection in an Uncooperative Environment. *Software-Practice and Experience*, pages 807–820, September 1988.

[7] D. Brylow, N. Damgaard, and J. Palsberg. Stack-size Estimation for Interrupt-driven Microcontrollers. Technical report, Purdue University, June 2000. http://www.brics.dk/ damgaard/Download/zilog-test.pdf.

[8] G. Chen, R. Shetty, M. Kandemir, N. Vijaykrishnan, and M.J. Irwin. Tuning Garbage Collection in an Embedded Java Environment. In *Eighth International Symposium on High-Performance Computer Architecture (HPCA'02)*, pages 92–106, Boston, Massachusettes, February 2002. IEEE.

[9] Manuvir Das. Unification-based pointer analysis with directional assignments. In *Proceedings of the SIGPLAN '00 Conference on Program Language Design and Implementation*, pages 35–46, Vancouver, BC, June 2000.

[10] Amer Diwan, J. Eliot Moss, and Kathryn McKinley. Simple and effective analysis of statically-typed object-oriented programs. In *Proc. of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 292–305. ACM Press, 1996.

[11] Document No. ARM DDI 0084D, ARM Ltd. *ARM7TDMI-S Data sheet*, October 1998.

[12] Michael Durrant. Running Linux on low cost, low power MMU-less processors. August 2000. http://www.linuxdevices.com/articles/AT6245686197.html.

[13] Jakob Engblom. Static properties of commercial embedded real-time programs and their implication for worst-case execution time analysis. In *Proc. of the IEEE Real-Time Technology & Applications Symposium (RTAS)*, June 1999.

[14] M. Game and A. Booker. Codepack: Code compression for PowerPC processors. *MicroNews 5(1)*, 1999.

[15] John Hennessy and David Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, Palo Alto, CA, third edition, 2002.

[16] *Intel i960Sx 32-bit Microprocessor*. Intel Corporation. http://www.intel.com/design/i960/documentation/docs_sx.htm.

[17] David Kleidermacher and Mark Griglock. Safety-Critical Operating Systems. *Embedded Systems Programming*, 14(10), September 2001. http://www.embedded.com/story/OEG20010829S0055.

[18] Rafael C. Krapf, Jlio C. B. Mattos, Gustavo Spellmeier, and Luigi Carro. A Study on a Garbage Collector for Embedded Applications. In *15th Symposium on Integrated Circuits and Systems Design*, pages 127–134, Porto Alegre, Brazil, September 2002. IEEE.

[19] Sergei Y. Larin and Thomas M. Conte. Compiler-Driven Cached Code Compression Schemes for Embedded ILP Processors. In *32nd Int'l Symposium on Microarchitecture*, pages 82–92, Haifa, Israel, November 1999. IEEE.

[20] Doug Lea. A Memory Allocator. April 2000. http://gee.cs.oswego.edu/dl/html/malloc.html.

[21] C.D. Lo. The Design of a Self-Maintained Memory Module for Real-Time Systems. In *The 3rd IEEE International Workshop on System-on-Chip for Real-Time Applications*, Alberta, Canada, July 2003. IEEE.

[22] *Windows CE.NET*. Microsoft Corporation. http://www.microsoft.com/embedded/ce.net/default.aspx.

[23] Motorola. *M68000 User's Manual*. Prentice Hall, Englewood Cliffs, NJ.

[24] *M-CORE - MMC2001 Reference Manual*. Motorola Corporation, 1998. http://www.motorola.com/SPS/-MCORE/info_documentation.htm.

[25] George V. Neville-Neil. Programming Without A Net. *ACM Queue: Tomorrow's Computing Today*, 1(2):16–23, April 2003.

[26] 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.

[27] Wind River. High Availability Design for Embedded Systems. http://www.windriver.com/whitepapers/-high_availability_design.html.

[28] Matthew Simpson, Surupa Biswas, and Rajeev Barua. Analysis of Compression Algorithms for Program Data. Technical report, U. of Maryland, ECE department, August 2003. http://www.ece.umd.edu/ barua/matt-compress-tr.pdf.

[29] David Solomon. *Data Compression: The Complete Reference*. Springer-Verlag Inc., New York, 2000.

[30] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Symposium on Principles of Programming Languages (POPL)*, St. Petersburg Beach, FL, January 1996.

[31] Krishnan Sundaresan and Nihar R. Mahapatra. Code Compression Techniques for Embedded Systems and Their Effectiveness . In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI'03)*, pages 262–263, Tampa, Florida, February 2003. IEEE.

[32] *MSP430 Ultra-Low-Power MCUs*. Texas Instruments, 2004. http://focus.ti.com/lit/ml/slab034g/slab034g.pdf.

[33] Sumesh Udayakumaran and Rajeev Barua. Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. In *Proceedings of the international conference on Compilers, architectures and synthesis for embedded systems*, pages 276–286. ACM Press, 2003.

[34] 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.

[35] Youtao Zhang and Rajiv Gupta. Data Compression Transformations for Dynamically Allocated Data Structures. In *Proceedings of the International Conference on Compiler Construction LNCS 2304*, pages 14–28, April 2002.