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

SURUPA BISWAS, THOMAS CARLEY, MATTHEW SIMPSON, BHUVAN MIDDHA and RAJEEV BARUA
University of Maryland

Embedded systems usually lack virtual memory and are vulnerable to memory overflow since they lack a mechanism to detect overflow or use swap space thereafter. We present a method to detect memory overflows using compiler-inserted software run-time checks. Its overheads in runtime and energy are 1.35% and 1.12% respectively. Detection of overflow allows system-specific remedial action. We also present techniques to grow the stack or heap segment after they overflow, into previously un-utilized space such as dead variables, free holes in the heap and space freed by compressing live variables. These may avoid the out-of-memory error if the space recovered is enough to complete execution. The reuse methods are able to grow the stack or heap beyond its overflow by an amount that varies widely by application – the amount of recovered space 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

General Terms: Reliability, Languages

Additional Key Words and Phrases: 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 [Hennessy and Patterson 2002] 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

Author's address: Rajeev Barua, Dept. of Electrical and Computer Engineering, University of Maryland, College Park 20742, Maryland, USA

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 [Neville-Neil 2003]. 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 that it will run to completion without running out of space. 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 six 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-class functions, *i.e.,* functions called through pointers, in imperative languages like C, which also result in a partially unknown call-graph; (iv) languages, such as GNU C, that allow stack arrays to be of run-time-dependent size; (v) calls to the *alloca()* function, present in some dialects of C, which allows a run-time-dependent size block to be allocated on the stack; and (vi) interrupts, since their handlers allocate space on the stack which may be difficult to estimate. In all these cases, estimating the stack size at compile-time may be difficult or 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.

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 input-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 [Brylow et al. 2000]. Unfortunately, this approach of choosing the size of physical memory never guarantees an upper bound on memory usage for all data sets; 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; however, 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 has serious consequences on the reliability of embedded systems in two ways. 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. This work proposes a scheme for software-only out-of-memory protection and memory reuse in embedded systems that improves system reliability. Its three components are described in turn below.

Safety run-time checks The first proposed technique to improve system reliability is to modify the application code in the compiler to insert software checks for all out-of-memory conditions. In a naive implementation, checking for stack or heap overflow requires a run-time check for overflow at each procedure call and each *malloc()* site in the program. Optimizations are presented to eliminate some of these checks while retaining the guarantee of always detecting overflow.

With such checks the embedded system when it runs out of memory can take corrective action of at least the following three types. First, upon overflow, control of the system can be transferred to a manual operator, such as to an aircraft pilot upon failure of the auto-pilot. Second, the controlled system can be shut down, such as shutting down an assembly line upon failure of a robot helping in assembly. Third, the system can transition to a limited-functionality fail-safe mode. None of these actions are possible without overflow detection.

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. They can be implemented if the designer wants to use previously un-usable memory at the cost of some implementation complexity.

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 un-utilized 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. We present techniques to achieve the reuse shown in all

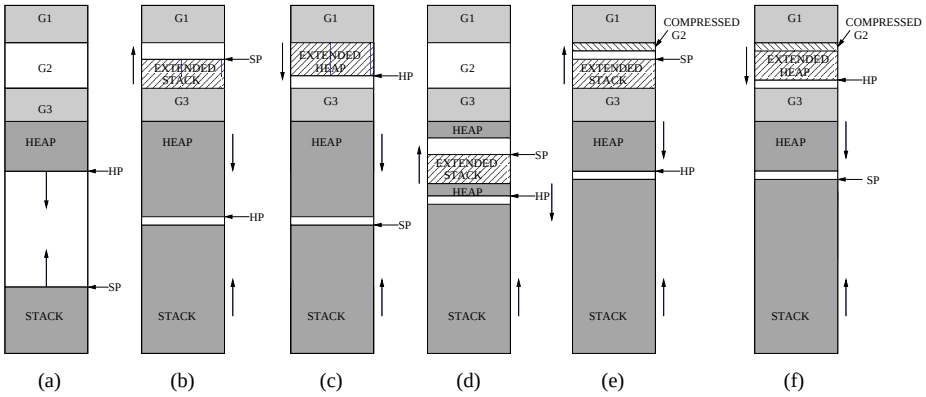


Fig. 1. Memory Layouts for our Schemes. (a) Normal operation; (b) Overflow stack in dead global *G2*; (c) Overflow heap in dead global *G2*; (d) Overflow stack in free hole in heap; (e) Overflow stack in compressed live global *G2*; (f) Overflow heap in compressed live global *G2*.

parts of figure 1.

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. Compression is used only after all dead space has been reclaimed by the reuse technique. Currently we investigate compressing globals for growing the stack and the heap. Figures 1(e) and (f) illustrate how the overflowing stack or heap, respectively, grows into space freed by compressing live global variables. The correctness and performance of this scheme is considered further in sections 7 and 8.

Remaining issues Let us examine whether each of our schemes can be used in real-time systems. Since the overhead of the safety run-time checks and reuse schemes is compile-time-predictable and small they are easily adapted to any real-time environment. The same is true for our compression schemes in the absence of memory errors. The only problematic case is when the compression schemes are used after the system has run out of memory; *i.e.,* is using reclaimed space. In this case the overheads are less predictable, so hard real-time guarantees are difficult to provide. However, soft real-time guarantees are still possible. In the vast majority of systems, a slow response is better than no response after an overflow.

It is worth noting that simply increasing the amount of physical memory in the system, which improves reliability, is not a substitute for SVM for the following four reasons. First, regardless of how much memory is provided, one cannot guarantee error-free execution at compile-time in the general case. SVM's detection of out-of-memory errors provides guaranteed protection in the unlikely event of overflow that is not provided by more memory. Second, the reliability at any *given* memory size and therefore system cost is improved. Thus SVM and increasing memory improve reliability in additive ways. Third, companies are unlikely to use much more memory than the maximum observed memory footprint in testing because of competitive pressures on cost. Fourth, future generations of chips with more

memory are unlikely to eliminate overflow either since application requirements grow with time too. For example, a decade ago embedded workloads for a single system rarely exceeded a few hundred lines of C code; today millions of lines of code are common. Therefore the probability of out-of-memory error does not reduce with time – it may even increase because of the difficulty of analyzing large applications and the reduced coverage of testing.

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 some remaining issues, such as the choice of data compression algorithm, the space requirement of our overhead routines, and liveness analysis. Section 10 describes experimental results. Section 11 concludes.

2. RELATED WORK

The problem of embedded systems' lack of hardware protection, and their consequent unreliability, has been widely recognized and lamented by industry practitioners. In an article in the Embedded Systems Programming magazine [Kleidermacher and Griglock 2001] 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 a whitepaper by Wind River [Win] 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) thresholds that trigger appropriate recovery actions." In an article appropriately titled "Programming Without A Net" [Neville-Neil 2003], the authors point out that even with a sophisticated operating system, an embedded system still does not have a good solution to the memory protection problem without hardware support; which is often unavailable.

Hardware schemes Our safety run-time checks apply to embedded platforms that lack hardware protection mechanisms; most commercial embedded processors are in this category. Example processors lacking virtual memory are the Atmel 8051, Motorola 68K, Intel i960Sx, ARM7TDMI, and TI MSP 430 families; there are many others. In such processors our safety run-time are valuable since they provide memory protection in software at low cost. The reason why most embedded processors lack protection hardware is their cost in area, run-time and power, and their severely negative impact on real-time bounds. For such processors, their vendors have decided that the cost of hardware memory management units (MMUs) that provide such protection is too high in a resource-constrained embedded environment [Durrant 2000]. It is easy to see why: MMUs must contain segment or page tables and their associated logic – these are expensive in area, run-time and power. Power consumption is an especially of concern since memory protection hardware is activated for *every* memory reference. Moreover virtual memory usu-

ally requires a Translation-Lookaside Buffer (TLB) in hardware – TLBs severely degrade real-time bounds because of the possibility of TLB misses.

Specialized hardware schemes for providing memory protection in embedded systems have also been devised. One is the Mondrian Memory Protection (MMP) [Witchel et al. 2002] scheme. MMP is designed to provide fine-grained word-level protection for systems requiring data sharing among processes. However, the necessary permissions tables are in hardware, and MMP requires additional CPU and memory system resources to access these tables – all these incur a cycle-time, energy and area cost. Another hardware approach [Carbone 2004] provides basic segment-level protection without requiring any translation lookaside buffers (TLBs). This technique relies only on the permissions capability of the MMU and not the virtual-to-physical address translation, but some hardware and energy costs remain[1]. Most importantly, all the schemes mentioned in this paragraph [Witchel et al. 2002; Carbone 2004] provides protection only against segmentation violations by using segment bounds in hardware, but neither provides any protection against memory overflow, which is the goal of this paper; nor do they attempt to reuse memory like we do.

Some embedded processors, instead of supporting virtual memory hardware, are equipped with a coprocessor, known as a Memory Protection Unit (MPU) [Jagger and Seal 2000], dedicated to lightweight memory access control. The MPU is used to partition the address space into, at most, eight regions varying in size from 4KB to 4GB. Each region is associated with individual access permissions, and the MPU allows access to each region based on the mode (user/privileged) and type (read/write) of the access. The incoming address is compared in parallel with all enabled regions to determine the appropriate access permissions. Although the MPU provides a limited form of protection, it provides no protection against memory overflow – that is not its goal. For example, even if a process's stack and heap are placed in different regions, since both regions allow writes from the process, the overflow of one into the other does not flag a violation.

Software-managed TLBs and software address translation are examples of combined hardware-software schemes used in place of entirely hardware-based virtual memory. Software-managed TLBs [Uhlig et al. 1994] allow the operating system to choose the page or segment table needed for address translation, providing for increased flexibility. Software address translation [Jacob and Mudge 2001] replaces the TLB, giving the operating system the ability to perform address translations. Because most embedded systems only rely on a single physical address space, address translation is not a goal of our work. Both software-managed TLBs and software address translation rely on significant virtual memory hardware support whose costs we aim to avoid.

Software schemes We are not aware of any software-only method for embedded systems that detects out-of-memory errors or reuses space in another segment in the same process when one segment is full.

One way to avoid run-time checks for the stack is to allocate stacks on the heap [Bobrow and Wegbreit 1973; Hauck and Dent 1968; Behren et al. 2003]. The

[1]Our recent work shows how to protect against segmentation violations in software [Simpson et al. 2005] rather than use the schemes in [Witchel et al. 2002; Carbone 2004].

stack frames are allocated on the heap and explicitly de-allocated upon procedure return. Our run-time checks are not needed since heap allocation routines implicitly contain a run-time check for overflow. Older schemes[Bobrow and Wegbreit 1973; Hauck and Dent 1968] use the unmodified system *malloc* and de-allocation *free* routines. This approach suffers from the following drawbacks compared to our scheme. First, since the size allocated for each call is unequal, these schemes suffer from external fragmentation and may not be able to utilize all the small, useless holes. Compaction to combine holes is undesirable because of its poor real-time bounds. Second, heap allocation also increases the run-time and energy use because of the overhead of the *malloc()* and *free()*, which can be in the thousands of cycles per call. Third, heap allocation routines degrade real-time bounds even more than the run-time since their worst case run-time is significantly more than the average case.

An improved way to allocate the stack for on the heap called the Capriccio system is proposed by Behren et al [Behren et al. 2003]; though in a completely different context and with a different goal. They 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 initially 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 eight ways. First, our system has a goal of detecting out-of-memory errors which is different from their goal of saving on virtual address space. Second, our method is applied, optimized, and evaluated for embedded systems rather than desktop servers. Third, our method works with any existing stack layout; their method requires a change in the stack layout to treat it more like the heap – their stack consists of un-ordered fixed-size chunks that are dynamically allocated. Fourth, our scheme does not incur the extra overhead of discontinuous stack growth unless the system is out of memory, which is rare; their scheme incurs this overhead whenever the small fixed-size chunks run out, which is more common. Fifth, 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. Sixth, our scheme can handle virtual function calls that appear in object-oriented languages; their scheme does not apply to such languages. Seventh, our reuse scheme can reclaim the space in dead global variables, which is not their goal. Eighth, our evaluation measures the impact on code-size and energy consumption which are important for embedded systems; they do not, given their focus on servers. Our results compare our overhead with that of Capriccio. To do so, we extracted the stack allocation on heap scheme from their full method which deals with many additional issues not relevant to embedded systems.

A different approach to increasing the amount of space available to a program is garbage collection. The goal of garbage collection is to reclaim unreachable heap objects. Recently, traditional garbage collection techniques have been adapted to embedded environments [Krapf et al. 2002; Chen et al. 2002]. However, of our five techniques four attempt to recover space from the global segment; garbage collection does not. Garbage collection is complementary to our scheme since it

also reclaims space but from a different source. A further distinguishing feature of our work is that we provide run-time checks for reliability which is not a feature of garbage collection.

Compression of program data [Zhang and Gupta 2002] has been discussed in the context of heap structures to reduce the memory footprint and hence the cost of embedded systems. On the other hand, the goal of our technique is to increase the reliability of the system by checking for memory overflows and smoothly transitioning to a reuse mode, or invoking corrective action, in case of a memory overflow. Other techniques reduce the amount of ROM required by compressing and compacting embedded code (not data) [Sundaresan and Mahapatra 2003; Larin and Conte 1999]; these are orthogonal to our methods.

Footprint estimation Methods for estimating the maximum depth of the stack [Regehr et al. 2003; Chatterjee et al. 2003; Heckmann and Ferdinand 2005; Analysis] are complementary to our work. Such work relies on analyzing the call graph to compute a worst-case estimate of the stack size when possible. Indeed, if for a particular program the size of the stack can be accurately estimated and no heap data is present then an out-of-memory error cannot occur. The compiler should turn off our method for such programs. However our method is valuable in the following three cases. First, if the program has heap data then the stack can overflow even if it has predictable size since the heap typically grows in the opposite direction. Heap size estimation is nearly impossible in the compiler. Heap data is not rare in embedded benchmarks – a survey of the MiBench embedded benchmark suite [Guthaus et al. 2001] shows that 17 out of the 29 benchmarks have heap data in the application code. If library code is included, then nearly all of the benchmarks have heap data. Second, in some cases the stack size estimates provided by analysis techniques may be too conservative to be acceptable. Third, in some cases the stack size cannot be accurately estimated if the maximum stack size depends on the input data, such as in the case of recursion and *alloca()* functions. In such cases, the common approach is to rely on user annotations. However user annotations have their drawbacks in that they may not be available for legacy code; and the programmer may not know enough about operating conditions to specify an upper bound on space usage. In all three cases above, our methods provide good back-up insurance against out-of-memory errors that works for all programs.

3. SAFETY RUN-TIME CHECKS FOR OVERFLOW PROTECTION

The safety run-time checks are implemented by the following two tasks. First, for heap checks, if the *malloc()* (or other heap allocation) routine finds that no free chunks of adequate size are available, it reports an out-of-memory error. Such a check is nothing new since it exists by default in most versions of *malloc()* and thus adds no overhead. Second, the stack checks, which are new and add extra overhead, are implemented as follows. Figure 2 shows the code for the stack check inserted at the beginning of every procedure's body. For the sake of explanation, without loss of generality, the rest of the paper assumes that the *stack grows from higher addresses to lower*; *i.e.,* upwards in figure1. To understand figure 2, consider that the stack pointer is decremented (not shown) at the start of every procedure by the size of the current procedure's frame. The code in figure 2 is inserted immediately

PER-PROCEDURE SAFETY CHECK CODE

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

Fig. 2. Pseudo-code for Safety Run-time Checks.

after the stack pointer is decremented. Therefore the check compares the updated stack pointer to the current allowable boundary for the stack. An error is flagged if the stack pointer is less than the stack's allowable boundary; *i.e.,* a stack overflow has occurred. This boundary 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 cases to use is known at compile-time; thus, the compiler uses the correct boundary in the compiled code.

Sometimes the calling convention of the compiler places variables beyond the stack pointer for call arguments. In this case our scheme needs to be modified so that the caller function must include the arguments to callees as part of the estimated frame size of the caller. Therefore the caller must check that space is available for its frame plus the maximum size of any arguments to called procedures.

The above scheme is un-optimized; however, we can reduce the overheads of the added stack checks by the *rolling checks optimization*, described as follows. 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. To implement the rolling optimization, the check is removed from one or more children, and the check at their parent is modified from that on line 1 of figure 2 to check for *Stack-Ptr + Max-of-rolled-child-frame-sizes < ORIGINAL_BOUND*. To see the benefits from rolling, consider that if the child is called more frequently than the parent then the reduction in overhead can be more than half. For good performance it is more important to roll checks out of frequently called 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 from inside recursive cycles to outside since every time a parent procedure is called its child procedure can be called multiple times if the child is recursive.

The safety rolling checks are easily extended to handle any calls to the *alloca()* library function. The *alloca()* function, provided in some dialects of C, allocates additional space on the current procedure's stack frame. The amount of additional space is a argument to *alloca()*, and need not be known at compile-time. If a procedure calls *alloca()*, its basic safety run-time check in line 1 of figure 2 is modified to check that *Stack-Ptr + Size-argument-of-alloca* $<$ *ORIGINAL_BOUND*. During the rolling checks optimization, when a procedure contains a call to *alloca()*, its check cannot be rolled to its parent since the parent cannot, in general, evaluate the size argument to *alloca()*.

Figure 3 shows the complete pseudo-code for the rolling checks optimization, taking into account the issues mentioned above. It is too involved to describe in detail; we briefly outline it 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), *alloca()* calls (line 15-16), and pre-mature declarations (lines 21-24). It also handles recursive functions in the application as outlined earlier (lines 17-20). Finally lines 25-28 check if the parent already had its check rolled; if so, the child recursively checks (line 27) whether it can roll its check to the parent's parents (its grandparents).

Routine **roll_check**() takes a similar recursive approach to **can_roll**(); however it actually rolls the checks from the current procedure up to its ancestors instead of just checking than rolling can be done. The recursive step for **roll_check**() is in line 32-34. 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 (else part of line 32); in which case the child's check is rolled to it (line 35-40). 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 for the child is set to zero and for the parent is set to the sum of the parent and child frame sizes. 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

```
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 (not (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 (Curr_Proc contains alloca() call)
16.     return (false)
17. if (either Curr_Proc or Ancestor recursive
        but not both in same cycle))
18.     return (false)
19. if (Curr_Proc == Ancestor)
20.     return (false)    /* Termination for recursive cycles */
21. Longest_path ← Path in call graph from Ancestor to
        Curr_Proc, not including Curr_Proc, with largest sum of
        stack frame sizes among all such paths


22. Sum_stack_size ← Sum of stack sizes along Longest_path
23. if (Sum_stack_size > 10% of max. stack + heap size in profile)
24.     return (false)
25. if (Rolled_size [Ancestor] == 0)       /* No check on Ancestor */
26.     for (each parent P of Ancestor in the call-graph)
27.         if (not (can_roll (Curr_Proc, P, Ancestor)))
28.             return (false)
29. return (true)     /* Can roll check from Curr_Proc to Ancestor */


void        roll_check (Curr_Proc, Ancestor) {
30. if (Curr_Proc == Ancestor)
31.     return (false)    /* Termination for recursive cycles */
32. if (Rolled_size [Ancestor] == 0)    /* Ancestor has no run-time check */
33.     for (each parent P of Ancestor in the call-graph)
34.         roll_check (Curr_Proc, P)
35. else {          /* Can roll check from Curr_Proc into Ancestor */
36.     Longest_path ← Path in call graph from Ancestor to
            Curr_Proc, not including Curr_Proc, with largest sum of
            stack frame sizes among all such paths
37.     Sum_stack_size ← Sum of stack sizes along Longest_path
38.     Rolled_size [Ancestor] ← max (Rolled_size [Ancestor],
                Sum_stack_size + Rolled_size [Curr_Proc])
39.     Rolled_size [Curr_Proc] ← 0
40. }
41. return
```

Fig. 3. Pseudo-code for rolling checks optimization

38). The results section will show that the rolling checks optimization eliminates slightly over half of the overhead of the run-time checks in our scheme.

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. 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.

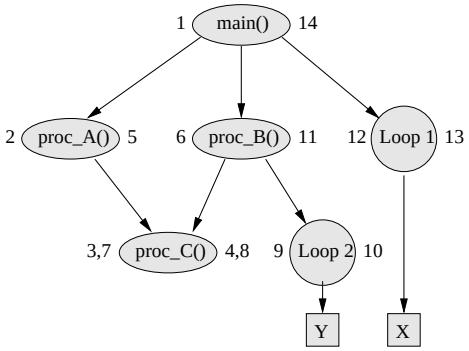
Identifying dead globals The method of choosing the global variable to grow the overflowing stack into has the following three steps. First, the compiler divides the program up into several blocks of code called regions. Which global variable(s) are chosen to grow into are *fixed at compile-time per region*, but different for different regions. Regions allow the decision of which global variables to grow into to be customized to where in the program code the overflow occurred. For each region, the compiler 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[2]. This deadness constraint enforces that none of the functions pushed on to the global variable (overflow) portion of the stack access the global array; that is, it enforces that the global array is truly dead throughout the run-time of the 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. 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.* Thus, our regions start just before loop-starts, just after loop-ends, just before function-starts and just after function-ends. Each region continues until

[2]Such liveness analysis is possible even for situations where the call-graph is not fully known. See section 9 for details.

```
main () {
  proc-A()
  proc-B()
  while (...){X =...} /* Loop 1 */
}
proc-A () {
  proc-C()
}
proc-B () {
  proc-C()
  for (...){Y =...}     /* Loop 2 */
}
proc-C () { ...}
```



(a) **(b)**

Fig. 4. Example showing (a) a Program Outline; and (b) its DPRG with Nodes, Edges & Timestamps.

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.

More formally, figure 4(b) is the *Data-Program Relationship Graph (DPRG)* [Udayakumaran and Barua 2003] for the code in figure 4(a). The DPRG is a compiler data structure that 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. Further details on the properties of the DPRG 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 – the old frame pointer, which stores the stack pointer value for the (non-overflowing) parent procedure, is recovered at procedure returns from the current procedure's frame as is usual

PER-REGION REUSE CODE

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

(i)

SAFETY CODE AUGMENTED WITH REUSE CODE FOR THAT REGION

```
1.  if ((Stack-Ptr < ORIGINAL_BOUND + Space needed by reuse routines)
        OR (Reuse-Started)) {
2.    if (not (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.   }
20. }
```

(ii)

Fig. 5. Pseudo-code for inserted Safety Run-time Checks augmented for Reuse.

in most compilers. The old frame pointer in the current stack frame is correct because it is saved *before* the discontinuous stack pointer assignment upon overflow. Regarding procedure arguments passed on the stack, observe that they are often written by the caller in its frame and read by the callee. To correctly handle such memory arguments, when the callee causes an overflow, then its call arguments must be copied over from the original space to the overflow space. This overhead is suffered only in the rare case of a memory overflow, so its cost is not an issue. Moreover this is done for memory arguments and not the more common case of arguments in registers.

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 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.

Most of the benchmarks we evaluated exhibit a common pattern that lends itself to a new optimization. Most of the benchmarks begin and end with I/O library calls, with no heap allocation in the middle. Each I/O file, including *stdin* and *stdout*, is associated with a a heap allocated buffer. Each time a library I/O function is called on a particular file a check is performed to determine whether the buffer is allocated; if not, then it is allocated. Therefore, the first time an I/O function is called on a particular file, it's buffer is allocated on the heap. Our new optimization, called the I/O optimization, facilitates the rolling checks and reuse optimizations by pre-allocating I/O buffers. It facilitates rolling checks by ensuring that I/O calls do not allocate heap data. It promotes the reuse of global variables used in certain library functions by reducing their live range to only the beginning of the program where heap is being pre-allocated.

5. REUSING GLOBALS FOR HEAP

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 first two tasks are described below.

Need for 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. 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 malloc 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 global coming back to life. If the extended heap is empty the program runs successfully. If the extended heap is not empty, then we declare that we are out-of-memory. In the latter 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 when globals that come alive later are chosen first to grow into.

Sorting reuse candidate lists The dead globals in the Reuse Candidate Lists are sorted by next-time-of-access as follows. Let us first consider the simple case of when the current region is *not* within any loop. In this case the later regions to the current region are the nodes with a greater timestamp than the current region. Given the later regions, the next-region-of-access of a global is computed as the region with the smallest timestamp among the later regions that access the global. Finally, the globals in the Reuse Candidate Lists are sorted in the timestamp order of their next-region-of-access.

Sorting the dead globals in the Reuse Candidate Lists by next-time-of-access is somewhat more complicated when the current region is within one or more loops. Within loops we would like to take the common-case order of nodes into account rather than simply the timestamp order. The common-case ordering of nodes in a loop *is along the backward edge of the loop*, since the backward edge is usually much more frequent than the forward edge into the loop. Sorting the Reuse Candidate Lists in the presence of loops is done as follows. Initially, the outer-most loop L of the current region is found. Next, based on the backward-edge intuition of common-case ordering of loop nodes, the following *common-case execution order* among the later regions is imposed: first, regions inside L with greater timestamp than the current region – these regions follow in the same iteration; second, regions inside L with lesser timestamp than the current region – these follow in the next iteration; and third, regions outside L with greater timestamps that the current

region – these follow after the loop has exited. Subsequently, the next-region-of-access of each dead global in the Reuse Candidate List is computed using the above execution order of regions. Finally, the dead globals in the list are themselves sorted in the order of their next-region-of-access.

Comparing the conditions for when a global variable can be used for growing the stack vs. heap we observe that the conditions are more restrictive for the stack. For either stack or heap growth, the global must be dead in the current region. For stack, however, we saw in section 4 that, in addition, the global must also be dead in all functions that are called directly or indirectly from that region. To implement this restriction, we add a *can_be_used_for_stack* flag to each element in the Reuse Candidate List. If the flag is true then the global can be used to grow the stack but not if the flag is false. Globals can be used to grow the overflowing heap without restriction by relying on the run-time check.

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 NULL 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 below. This task simply involves replacing the return statement in *malloc()* with a call to the Out-of-Heap Function. Since this call is executed only when the program has actually run out of heap space, there is no overhead in the common case when the program is not out of memory. Moreover this method for growing the heap into globals is based on the earlier framework of reusing globals for stack and requires no additional data-structures.

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; 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 [Lea 2000]. This method of growing into free holes in the heap is unnecessary and should not be done when holes are periodically eliminated by heap compaction. However, heap compaction is rare in embedded systems given its negative impact on real-time bounds.

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. The implementation differs from the scheme for growing the

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.  }
```

Fig. 6. Extra Pseudo-code for Compression added to Figures 5(i) and (ii).

stack into dead globals in the following three ways. First, the reuse candidates are extended to include live global arrays. To avoid compression to the extent possible the Reuse Candidate List is sorted so that dead globals are placed ahead of live globals in the list. 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 Live arrays that can be compressed are added to the Reuse Candidate List. A live 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. Further, 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. The reuse candidate lists are sorted as before to place variables that will not be accessed the longest at the head of the list. An extra boolean field is added to each candidate to indicate whether it is dead or live in that region.

Triggering compression 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 its boolean field for 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 started, then all compressed global arrays accessed in the following

region are de-compressed in their original locations (line 3). De-compression is safe since compiler analysis described earlier in this section guarantees that by the time the global is accessed again the overflow stack in its space would have receded. 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 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. The impact of compression schemes on real-time bounds is described in the introduction.

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 two components. First, 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. Second, 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 scheme of compressing globals for stack.

9. REMAINING ISSUES

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 recovers significant space and is fast, since it used at run-time. 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 [Solomon 2000]; (ii) WKdm, which uses a

combination of dictionary-based and statistical methods and is characterized by a very small dictionary size [Wilson et al. 1999] 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 [Simpson et al. 2003].

Upon evaluation, we chose WKS because it has the best compression ratio when tested on global variables and is fast. 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 can be found in a technical report [Simpson et al. 2003].

Space Overheads of our Routines 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 two reasons. First, calls are made to certain functions such as the Out-of-Heap Function (sections 5 and 8) and the compression and de-compression functions (sections 7 and 8). Each of these functions requires some stack space. For correctness, the application cannot wait until the stack is full to make these calls; instead it makes the calls when there is just enough space on the stack for the 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 premature invocation of our method, special care is taken in writing our functions to ensure that their stack space is small.

Liveness Analysis Liveness analysis, needed for our reuse schemes for detecting dead globals, is a well-understood dataflow analysis in the compiler literature [Appel and Ginsburg 1998]. 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, function pointers may prevent knowledge of the call graph at compile-time, even with pointer analysis.

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 and correct in that it never declares a live variable to be dead. For object-oriented languages, liveness analysis has been investigated in [Persson 1999]. Restricting the set of functions a virtual function may call, is possible at compile-time, in many cases, by using techniques such as [Diwan et al. 1996] 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-class functions are rare [Engblom 1999], and hence this problem is mostly absent. When they are present, their number of instances is low. Even in such languages pointer analysis allows for conservative but correct liveness analysis.

Benchmark	Source	Description	Total Data Size (bytes)	Lines of Code
ADPCM	MIBench	Dynamic Pulse Code Modulation	10852	312
BITCOUNT	MIBench	Bit Counting Functions	8484	938
CRC32	MIBench	Cyclic Redundency Checksum	9028	293
HISTOGRAM	UTDSP	Image Enhancing Application	17850	219
JPEG	UTDSP	Image Encoding and Decoding	169000	3279
KS	PTRDist	Graph Partitioning Tool	31400	1042
LPC	UTDSP	Linear Predictive Coding	8000	556
QSORT	MIBench	Quick Sort	7687250	64
SPECTRAL	UTDSP	Power Spectral Est. of Speech	3200	228
SUSAN	MIBench	Digital Image Processing	383000	2131

Table I. Benchmark Programs and Characteristics

10. RESULTS

This section presents results for the different schemes proposed in this work. The proposed techniques have been implemented in the public-domain GCC v3.2 cross-compiler [compiler] targeting the Motorola M-Core embedded processor. The benchmarks are compiled with the *Newlib-1.8.1* library [Red Hat, Inc.]. After compilation the benchmarks are executed on the public-domain cycle-accurate simulator for the Motorola M-Core available as part of the GDB v5.3 distribution [project debugger]. The profiling required for the rolling checks optimization is also implemented in the compiler using compiler-inserted profile instrumentations in the application being compiled. The profile-measured statistic for rolling checks is the number of times each application procedure is called dynamically[3].

The names, sources, and other characteristics of the embedded benchmarks evaluated are shown in table I. 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.

Our experimental setup for estimating the energy consumption of programs with and without our method is as follows. An M-core power simulator [Baynes et al. 2003], kindly donated by that group, is used to obtain energy estimates for instructions and SRAM. This is an instruction-level power simulator similar to [Sinha and Chandrakasan 2001]; its instruction power numbers were measured using an

[3]Theoretically, any liveness analysis method can be used, either based on traditional dataflow or SSA form; any pointer analysis can also be used. Unfortunately gcc v3.2 does not include either, so we approximated the analysis using profile data. To see when a variable could be live, we noted when it was actually live in the profile data. Pointers were resolved at run-time to see what they actually pointed to. Using the profile information in this way is admittedly unsafe since theoretically pointers can refer to objects other than in the profile data. However we visually examined the live ranges produced in this way and verified that they were the same as true live ranges we derived by manually inspecting the code. (They did in nearly all cases; the violations were fixed manually). The reason the pointer analysis results were mostly correct using the (unsafe) profile data method is that our benchmarks use pointers to arrays in a fairly straightforward way (usually for hand-optimization of array accesses where a pointer can point to only one variable - we are only interested in pointers to arrays in our analysis). In this way our results are not compromised by our approximation. Fortunately gcc v4.0, currently in beta-release, includes both analyses, so this should not be a problem in the future.

ammeter connected to an M-core hardware board. DRAM power is estimated by a DRAM power simulator we built into the M-core simulator. It uses the DRAM power model described in [Janzen 2001; micron-datasheet 2003] for the MICRON external DDR Synchronous DRAM chip. The DRAM chip size is set equal to the data size in the energy model. Both the CPU and DRAM use aggressive energy saving technologies.

Safety run-time checks Table II 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. Comparing them we see that the run-time overhead reduces from 3.01% to 1.35% with optimizations. Next, comparing the fourth and fifth columns, we see that the code-size overhead reduces from 0.12% to 0.10% with the same optimization. The improvement in run-time from optimization is more than that for code size since the most frequent checks are rolled first. Finally, comparing the sixth and seventh column, we see that the energy consumption overhead reduces from 2.41% to 1.12% with the same optimization.

The above results show that safety run-time checks, which guarantee detection of out-of-memory errors, are possible with very low overhead. Quantitative results cannot evaluate the benefit of remedial action that our out-of-memory checks allow, which can be invaluable.

Benchmark	**Run-time Increase(%)**		**Code Size Increase(%)**		**Energy Increase(%)**	
	Without Optimization	With Optimization	Without Optimization	With Optimization	Without Optimization	With Optimization
ADPCM	0.06	0.05	0.05	0.02	0.05	0.04
BITCOUNT	6.18	0.90	0.14	0.05	4.30	0.62
CRC32	0.21	0.21	0.05	0.02	0.17	0.17
HISTOGRAM	4.41	2.97	0.10	0.09	3.69	2.48
JPEG	2.29	0.28	0.23	0.22	1.94	0.24
KS	4.38	1.15	0.14	0.13	3.47	0.91
LPC	4.61	2.90	0.14	0.13	4.02	2.53
QSORT	5.71	3.62	0.02	0.02	4.40	2.76
SPECTRAL	1.36	1.26	0.17	0.16	1.20	1.10
SUSAN	0.92	0.19	0.15	0.14	0.83	0.30
Average	*3.01*	*1.35*	*0.12*	*0.10*	*2.41*	*1.12*

Table II. Overheads for Safety Checks

Figure 7 shows the contribution to the run-time overhead of the safety run-time checks from the different restrictions on the rolling checks optimization. *Most of the bars except the last one ("PLUS Size Threshold") are not visible because they are close to zero.* To understand this figure, recall that section 3 had listed several restrictions on rolling checks: heap allocation, first-class functions, recursion, the size threshold, virtual functions and alloca() calls. These restrictions are *necessary*

for safety and cannot be removed if safety is desired, but nevertheless it is interesting to see the contribution from each restriction to the overhead. The left-most bar for each benchmark in the figure shows the overhead in the hypothetical case without any restrictions, which is always zero, since without restrictions the checks can always be rolled to *main()*. The remaining bars add in restrictions one-by-one in the order shown in the figure, until the right-most bar shows the overhead with all restrictions. Two restrictions – virtual functions and alloca() calls – are not shown since they cannot occur in ANSI C, which is the language of our benchmarks.

The results in figure 7 show the following three trends. First, the average overhead contributions from heap allocation is very small – only 0.004% on average (looks like zero in the figure). This is because *malloc()* call sites in programs prevent rolling, and there are very few *malloc()* call sites in our benchmarks. Of course, how often these *malloc()* call sites are executed has no impact on rolling, so the total amount of heap data could be large. Second, the additional overhead contribution from recursion is 0.09% and from first-class functions is 0.27%; these are rare in our benchmarks and in embedded codes in general. Third, the additional overhead from the size threshold is 1.35% - 0.27% = 1.08%. In other words, the size threshold contributes most of the overhead from our safety checks. This contribution to the overhead can be reduced to almost zero by increasing the size threshold to above 10%.

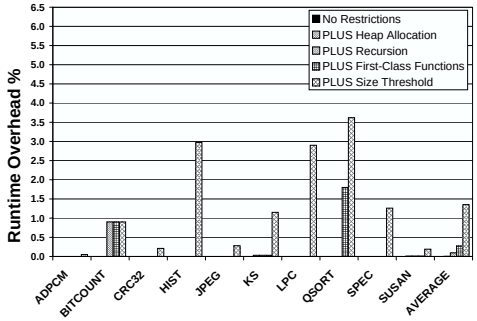


Fig. 7. Run-time overhead contribution to safety checks from each restriction to the rolling checks optimization.

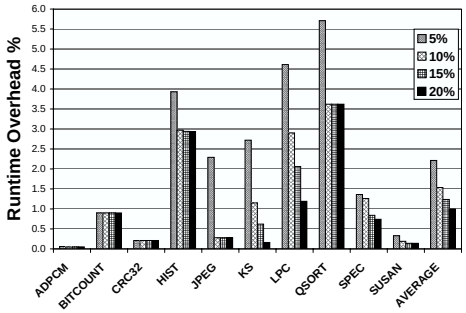


Fig. 8. Impact of varying size threshold on run-time overhead.

Figure 8 shows the impact of varying the size threshold on the run-time overhead of our safety checks. All the other figures in the paper assume that the size threshold, described in section 3, is fixed at 10% of the combined maximum stack and heap segment sizes of the program. Here we observe that when we vary the size threshold from 5% to 20%, the run-time overhead decreases from 2.21% to 1.00%. This decrease is as expected since when the threshold is increased, the rolling checks optimizer can roll more checks, eliminating their overhead. Since there is a significant decline in overhead from by increasing the threshold from 10% to 20%, we see that run-time overhead can be reduced if desired at the cost of possibly declaring an out-of-memory condition slightly earlier than it really happens. This is true for the reuse checks as well, where the overheads are higher.

Benchmark	Reasons for Overflow					
	Difficulties in Estimation				Heap-related	
	Recursive Functions	Function Pointers	alloca	Variable-sized Stack Arrays	Heap Present	Recursive Data Structures
ADPCM		L			L	
BITCOUNT	A	A L			L	
CRC32		L			L	
HISTOGRAM		L			L	
JPEG		L			A L	
KS		L			A L	A
LPC		L			L	
QSORT		A L			L	
SPECTRAL		L			L	
SUSAN		L			A L	

Table III. Possible reasons for memory overflow. For the feature in each column, "A" for a benchmark means that feature is present in the benchmark's application code; "L" means that feature is in the library routines that are reachable from the benchmark's application code. Otherwise the entry is blank.

Table III lists if each benchmark has the particular program constructs that are the underlying reasons for overflow in programs. As shown in the table header, programs may overflow because their stack size is hard to estimate because of certain program constructs, or because a heap is present. Each entry lists if that feature is present in the application code ("A") and reachable library code ("L") for the benchmark in that row. From the table we see that all the benchmarks have function pointers and heap; in most cases these features are in the I/O library routines called by most of the benchmarks. Recursion, function pointers and heap are less common in the application code, but it is quite common to have at least one of these. The heap is not usually predictable in size even if recursive data structures are absent since arrays whose sizes are not known at compile-time are the other major type of heap data structure. Virtual functions are not listed since the benchmarks are written in C and they cannot be present. Variable-sized stack arrays and *alloca()* functions are not present since these are non-ANSI-C features, whereas the benchmarks and their *Newlib* library [Red Hat, Inc.] are written in ANSI-C to make them portable to a wide number of platforms. Interrupts, which can complicate stack size estimation, are not shown since they are not part of the benchmark but arise from the operating system environment which we do not model. From the table we can see that all ten benchmarks have at least one feature that can lead to an unpredictable memory footprint, justifying the value of our memory-overflow detection checks.

Comparison with non-contiguous stack allocation The Capriccio scheme [Behren et al. 2003], described in section 2 allocates the stack in fixed-size chunks from the heap using a custom allocator, and it uses run-time checks to detect stack overflow with a chunk. Figure 9 compares the overheads of our scheme and of Capriccio. It shows that the average overhead from our scheme is 1.35% versus 4.43% for Capriccio. To understand why, consider that non-contiguous allocation has essentially the same number of checks as contiguous allocation in the common

case of no overflow. In our contiguous allocation, this inexpensive stack-overflow check is the only overhead. However, non-contiguous in Capriccio has two additional sources of overhead when a fixed-size chunk overflows. First, a new chunk is allocated upon overflow. In addition this chunk is freed when the stack withdraws from it. Second, when an overflow is detected, the stack pointer is changed to a non-contiguous chunk and the call arguments copied over to the new chunk. Thus the overhead for our scheme is always less than or equal to that of Capriccio.

Upon closer examination we see that in most applications the overheads for the two schemes are comparable, except in two benchmarks – QSORT and HISTOGRAM – the overhead is much higher with Capriccio than with our scheme. In these benchmarks a function call within a frequently executed loop causes a chunk overflow. The resulting overhead of non-contiguous allocation severely hurts the run-time. In our scheme where the allocation is contiguous before overflow, this overhead is not incurred under normal operating conditions. The results show that although the Capriccio scheme does well for many benchmarks, it does quite poorly for a few. In contrast our scheme has consistently low overheads for all our benchmarks.

Benchmark	**Run-time Increase(%) from**	
	Our Scheme	Capriccio
ADPCM	0.05	0.05
BITCOUNT	0.90	0.90
CRC32	0.21	0.21
HISTOGRAM	2.97	18.61
JPEG	0.28	1.02
KS	1.15	1.53
LPC	2.90	3.22
QSORT	3.62	14.14
SPECTRAL	1.26	1.62
SUSAN	0.19	0.21
Average	*1.35*	*4.43*

Fig. 9. Comparison of run-time overheads of our scheme and Capriccio.

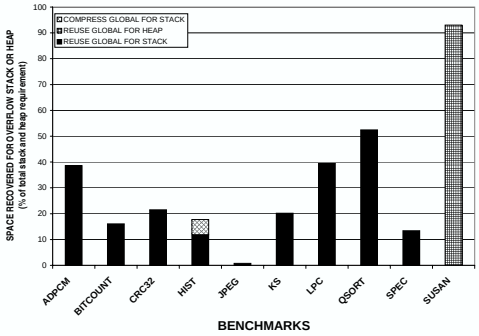


Fig. 10. Extra Space recovered for Stack and Heap as a Fraction of Total Stack and Heap Requirement

It is worthwhile to note that our contribution is more than simply bringing down the overhead from 4.43% to 1.35% in run-time, significant as that is. Instead our contribution is presenting a solution to the problem of detecting memory overflow in embedded systems for the first time. No one else, as far as we know, has ever made the connection that the Capriccio method for achieving high-efficiency desktop servers with virtual memory would be valuable for embedded systems without virtual memory. Earlier, an industry practitioner wanting to solve the problem for embedded systems would have no guidance on how to do so. Now they will. Further we improved upon their overhead by devising a scheme of mostly contiguous growth. Finally our reuse and compression schemes utilize free space in the system, such as from dead globals and compressed live globals, which was not their goal.

Reuse and compression benefits Figure 10 shows the benefits of the reuse and compression schemes in terms of the amount of space they recover. The results express the recovered space as a percentage in the total memory footprint of the growing segments – stack and heap – since those are the segments whose sizes are at risk of being under-estimated. The gain is highly application dependent since the amount of recoverable space varies greatly by application. This is not surprising since some benchmarks have one or more large dead global variables, compressible live global variables and free holes in the heap; other benchmarks have none of these. The figure shows that the recovered space can accommodate from 0.77%, in the case of JPEG, to 93.5% in the case of SUSAN, of the total stack and heap size of the benchmark. *In other words, even if we under-estimate the required heap and stack size by 93.5%, for example for SUSAN, the application will still run to completion.* Given the wide range of space recovered an average is not meaningful, and is not presented.

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 20.2% less than the actual dynamic memory requirement calculated in the first pass.

The significant space recovery shown in figure 10 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.

Figure 10 also shows the contribution of the different schemes to the total space recovered for each benchmark. Reusing globals for stack appears to be the most promising. To understand figure 10 further, the following benchmark-specific observations are necessary. First, 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 is a stack variable in the *main()* procedure. Our compiler implements a simple automatic optimization which promotes all arrays in *main()* to global variables so that our method can benefit from them. Susan has a substantial amount of heap data, which, upon overflow, grows into the space for the large array, explaining why for SUSAN the reuse global-for-heap scheme is pre-dominant. Second, HISTOGRAM, SPECTRAL and LPC have global arrays with mutually exclusive lifetimes. Such arrays are primarily input arrays and output arrays; the input arrays are dead in the later half of the program, and the output arrays are dead in the earlier half. Our scheme is able to grow overflowing segments into both arrays at different times. Third, HISTOGRAM presents additional opportunities. One of the arrays in the candidate list is live throughout, making its reuse impossible; however, the array

Benchmark	**Size of globals** (% of total data size)	**Max. size of stack** (% of total data size)	**Max. Size of heap** (% of total data size)
ADPCM	57.0	23.3	19.7
BITCOUNT	45.0	30.5	24.5
CRC32	47.3	34.7	17.9
HISTOGRAM	92.3	2.0	5.7
JPEG	6.3	0.7	93.0
KS	73.1	6.1	20.8
LPC	74.2	13.0	12.8
QSORT	99.9	0.0	0.1
SPECTRAL	47.0	21.0	32.0
SUSAN	48.2	3.1	48.7

Table IV. Size of the global, stack and heap segments for each benchmark, as a % of the total data size.

is *not actively accessed* throughout and thus, compression is feasible and is automatically invoked. Fourth, ADPCM, BITCOUNT, CRC32 and QSORT are able to recover most of their space declared from a dead global array declared in the heap allocation library routines. This array is used for all heap allocation calls, which in these benchmarks are only in the I/O library routines. With the I/O optimization described in section 4, these applications use the heap only at the start of the program, so the array can be reused for the rest of the program. Fifth, the improvement in JPEG is small because it has large heap structures whose live ranges span the entire program. The small benefit arose from reusing some global space for stack.

Figure 10 also shows that two of our schemes – reusing heap for stack and compressing globals for heap – did not yield any improvement for our benchmarks[4]. The following are three reasons why these techniques are not useful for the particular set of benchmarks considered here. First, for some of the benchmarks the heap for stack technique is not useful because the the compiler-derived live ranges of the heap structures spanned the entire length of the program. Therefore there were no free holes created that could be used for growing the stack. Second, for two of the benchmarks – KS and JPEG – the compressing global for heap technique is not useful because, although there were live arrays which were compressed, the heap objects in their place remained alive when the compressed arrays needed to be accessed again, resulting in a dynamically detected out-of-memory condition. Third, in SUSAN the compressing global for heap technique is not useful because the only global is a large dead global, and no opportunities for compressing live globals exist.

Table IV shows for each benchmark, the maximum sizes of the global, stack and heap segments as a percentage of the total data size for that benchmark. Table V shows some region-specific statistics. For each benchmark, it shows (i) the number of regions (after region merging); (ii) the average number of dynamic instructions

[4]Of course, they might yield improvement for other benchmarks.

Benchmark	# of Regions	Ave. runtime in cycles per region invocation	Ave. length of Reuse Candidate Lists
ADPCM	2	20.4M	1.0
BITCOUNT	2	29.8M	0.5
CRC32	2	21.5M	0.5
HISTOGRAM	3	11.3M	1.0
JPEG	2	0.9M	7.5
KS	3	1.4M	2.4
LPC	2	6.2M	5.0
QSORT	2	1.3M	1.0
SPECTRAL	2	54.1M	2.0
SUSAN	1	138.5M	1.0

Table V. Region statistics per benchmark.

each time a region is entered; and (iii) the number of elements in the per-region Reuse Candidate Lists, averaged across all the regions. The number of regions is usually small, and not more than three for our benchmarks. This is because of the region-merging optimization described in section 4 which combines regions with identical reuse candidate lists. To see why, consider that some of our benchmarks had a set of input arrays and another set of output arrays; the input arrays are dead in the later half of the program, and the output arrays are dead in the earlier half. Thus there are only two reuse scenarios possible and thus only two regions in such a case. Further, for several of the benchmarks, the average number of elements in the Reuse Candidate Lists is greater than one, which shows that it may be worthwhile to allow overflowing regions to grow into more than one dead global variable or free hole in heap, as our method does.

Reuse and compression overheads Table VI shows the increase in run-time, code-size and power consumption caused by our reuse techniques. The increase in run-time is due to the insertion of the reuse checks. Our rolling check and region-merging optimizations however, reduce the run-time increase significantly. Comparing the second and third column of the figure, we see that the optimizations are able to reduce the run-time overhead from 6.93% to 3.23% on average. The optimized run-time overhead is higher than for the safety checks, but is still low.

Table VI also shows the increase in code-size in the fourth, fifth and sixth columns. 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. Columns four and five show the average increase in code size from the run-time checks reduces from 0.26% to 0.23% with optimizations. The improvement in run-time from optimization is more than that for code size since the most frequent checks are rolled first. Column six shows that the fixed part of the code size increase (from the added routines) is the same (6.7Kbyte) for all benchmarks, except HISTOGRAM, for which it is higher because it also uses compression and de-compression routines. As a percentage of total code size, the increase from the fixed part averages 1.93% for our benchmarks. Thus the total code size increase is 0.23% + 1.93% = 2.16%.

Benchmark	Increase in Run-time (%)		Increase in Code-size			Increase in Energy use (%)	
			From Checks (%)		from routines (KB, %)		
	Without optimization	With optimization	Without optimization	With optimization		Without optimization	With optimization
ADPCM	0.13	0.11	0.06	0.03	6.7, 1.71	0.10	0.08
BITCOUNT	13.23	1.92	0.19	0.09	6.7, 1.70	9.38	1.36
CRC32	0.45	0.45	0.06	0.03	6.7, 1.72	0.36	0.36
HISTOGRAM	11.47	7.72	0.30	0.28	14.3, 4.00	9.22	6.20
JPEG	4.90	0.60	0.50	0.49	6.7, 2.10	4.26	0.52
KS	9.38	2.47	0.34	0.30	6.7, 1.70	7.62	2.00
LPC	11.97	7.54	0.37	0.34	6.7, 1.50	10.04	6.32
QSORT	12.23	7.76	0.02	0.02	6.7, 1.75	9.56	6.07
SPECTRAL	3.55	3.28	0.44	0.42	6.7, 1.60	3.00	2.76
SUSAN	1.98	0.40	0.34	0.30	6.7, 1.50	1.83	0.65
Average	*6.93*	*3.23*	*0.26*	*0.23*	*1.93%*	*5.54*	*2.63*

Table VI. Overheads for Memory Reuse and Compression Schemes

The last two columns of table VI shows the percentage increase in energy consumption due to our reuse and compression schemes in the common case. We see that the average increase in energy consumption reduces from 5.54% to 2.63% by using our optimizations.

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 VI. We do, however, count their impact in the earlier experiment in fig. 10, 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 almost zero. This is not surprising since we saw in table V that the number of regions in the code is very small.

11. CONCLUSION

This work 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 from our benchmarks show that the overheads from the system of run-time checks for detecting memory overflow are extremely low: the run-time, code-size and energy consumption overheads are 1.35%, 0.10% and 1.12% on average. When the reuse functionality is included, the run-time, code-size and energy consumption overheads increase to only 3.23%, 2.16% and 2.63% respectively. The reuse methods are able to grow the stack or heap beyond its overflow by an amount that varies widely by application – the amount of recovered space ranges from 0.7%

to 93.5% of the combined stack and heap size.

REFERENCES

High availability design for embedded systems. Tech. rep., Wind River, Inc. http://-www.windriver.com/whitepapers/high_availability_design.html.

Analysis, S. . S. U. AbsInt Inc. http://www.absint.com/stackanalyzer/.

Appel, A. W. and Ginsburg, M. 1998. *Modern Compiler Implementation in C*. Cambridge Univ. Press.

Baynes, K., Collins, C., Fiterman, E., Ganesh, B., Kohout, P., Smit, C., Zhang, T., and Jacob, B. 2003. The performance and energy consumption of embedded real-time operating systems. *IEEE Trans. Comput. 52,* 11 (Nov.), 1454–1469.

Behren, R. V., Condit, J., Zhou, F., Necula, G., and Brewer, E. 2003. Cappricio: Scalable Threads for Internet Services. In *Proc., ACM Symposium on Operating Systems Principles (SOSP)* (New York).

Bobrow, D. and Wegbreit, B. 1973. A model and stack implementation of multiple environments. In *Communications of the ACM*. 591–603.

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

Carbone, J. 2004. Efficient memory protection for embedded systems. *RTC Magazine*. `http://www.rtcmagazine.com/home/article.php?id=100120`.

Chatterjee, K., Ma, D., Majumdar, R., Zhao, T., Henzinger, T. A., and Palsberg, J. 2003. Stack size analysis of interrupt driven software. In *Proceedings of the International Static Analysis Symposium (SAS)*. 109–126.

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

compiler, T. G. Free Software Foundation. *http://gcc.gnu.org/*.

Diwan, A., Moss, J. E., and McKinley, K. 1996. 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*. ACM Press, 292–305.

Durrant, M. 2000. Running Linux on low cost, low power MMU-less processors. http://-www.linuxdevices.com/articles/AT6245686197.html.

Engblom, J. 1999. 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)* (Vancouver, Canada).

Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., and Brown, R. B. 2001. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE 4th Annual Workshop on Workload Characterization*.

Hauck, E. and Dent, B. 1968. Burroughs b 6500/b 7500 stack mechanism. In *Proceedings of AFIPS, SJCC, vol 32*. 245–251.

Heckmann, R. and Ferdinand, C. 2005. Verifying safety-critical timing and memory-usage properties of embedded software by abstract interpretation. In *Proceedings of Design, Automation and Test in Europe (DATE'05)*. 618–619.

Hennessy, J. and Patterson, D. 2002. *Computer Architecture: A Quantitative Approach*, third ed. Morgan Kaufmann, Palo Alto, CA.

Jacob, B. L. and Mudge, T. N. 2001. Uniprocessor virtual memory without TLBs. *IEEE Transactions on Computers 50,* 5 (May), 482–499.

Jagger, D. and Seal, D. 2000. *ARM Architecture Reference Manual*. Addison Wesley.

Janzen, J. 2001. Calculating Memory System Power for DDR SDRAM. In *DesignLine Journal*. Vol. 10(2). Micron Technology Inc. http://www.micron.com/publications/designline.html.

Kleidermacher, D. and Griglock, M. 2001. Safety-Critical Operating Systems. *Embedded Systems Programming 14,* 10 (September). http://www.embedded.com/story/-OEG20010829S0055.

Krapf, R. C., Mattos, J. C. B., Spellmeier, G., and Carro, L. 2002. A Study on a Garbage Collector for Embedded Applications. In *15th Symposium on Integrated Circuits and Systems Design*. IEEE, Porto Alegre, Brazil, 127–134.

Larin, S. Y. and Conte, T. M. 1999. Compiler-Driven Cached Code Compression Schemes for Embedded ILP Processors. In *32nd Int'l Symposium on Microarchitecture*. IEEE, Haifa, Israel, 82–92.

Lea, D. 2000. A Memory Allocator. http://gee.cs.oswego.edu/dl/html/malloc.html.

micron-datasheet 2003. *128Mb DDR SDRAM data sheet*. (Dual data-rate synchronous DRAM) Micron Technology Inc. http://www.micron.com/products/dram/ddrsdram/.

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

Persson, P. 1999. 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*. ACM Press, 45–54.

project debugger, G. T. G. Free Software Foundation. *http://www.gnu.org/software/gdb/-gdb.html*.

Red Hat, Inc. *Newlib C Library*. Red Hat, Inc. http://sources.redhat.com/newlib.

Regehr, J., Reid, A., and Webb, K. 2003. Eliminating stack overflow by abstract interpretation. In *Proceedings of the 3rd international conference on embedded software (EMSOFT)*. Springer-Verlag.

Simpson, M., Biswas, S., and Barua, R. 2003. Analysis of Compression Algorithms for Program Data. Tech. rep., U. of Maryland, ECE department. August. http://www.ece.umd.edu/-∼barua/matt-compress-tr.pdf.

Simpson, M., Middha, B., and Barua, R. 2005. Segment Protection for Embedded Systems Using Run-time Checks. In *Proceedings of the international conference on Compilers, architectures and synthesis for embedded systems*.

Sinha, A. and Chandrakasan, A. 2001. JouleTrack - A Web Based Tool for Software Energy Profiling. In *Design Automation Conference*. 220–225.

Solomon, D. 2000. *Data Compression: The Complete Reference*. Springer-Verlag Inc., New York.

Sundaresan, K. and Mahapatra, N. R. 2003. Code Compression Techniques for Embedded Systems and Their Effectiveness . In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI'03)*. IEEE, Tampa, Florida, 262–263.

Udayakumaran, S. and Barua, R. 2003. 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*. ACM Press, 276–286.

Uhlig, R., Nagle, D., Stanley, T., Mudge, T., Sechrest, S., and Brown, R. 1994. Design tradeoffs for software-managed TLBs. *Transactions on Computer Systems (TOCS) 12,* 3, 175–205.

Wilson, P. R., Kaplan, S. F., and Smaragdakis, Y. 1999. The case for compressed caching in virtual memory systems. In *Proceedings of the USENIX Annual Technical Conference*. Monterey, CA.

Witchel, E., Cates, J., and Asanović, K. 2002. Mondrian memory protection. In *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 304–316.

Zhang, Y. and Gupta, R. 2002. Data Compression Transformations for Dynamically Allocated Data Structures. In *Proceedings of the International Conference on Compiler Construction LNCS 2304*. 14–28.

Received August 2004; revised August 2005; accepted October 2005