MTSS: Multi Task Stack Sharing for Embedded Systems

BHUVAN MIDDHA, MATTHEW SIMPSON and RAJEEV BARUA
University of Maryland

Out-of-memory errors are a serious source of unreliability in most embedded systems. Applications run out of main memory because of the frequent difficulty of estimating the memory requirement before deployment, either because it depends on input data, or because certain language features prevent estimation. The typical lack of disks and virtual memory in embedded systems has a serious consequence when an out-of-memory error occurs. Without swap space, the system crashes if its memory footprint exceeds the available memory by even one byte.

This work improves reliability for multi-tasking embedded systems by proposing MTSS, a multi-task stack sharing technique. If a task attempts to overflow the bounds of its allocated stack space, MTSS grows its stack into the stack memory space allocated for other tasks. This technique can avoid the out-of-memory error if the extra space recovered is enough to complete execution. Experiments show that MTSS is able to recover an average of 54% of the stack space allocated to the overflowing task in the free space of other tasks. In addition, unlike conventional systems, MTSS detects memory overflows, allowing the possibility of remedial action or a graceful exit if the recovered space is not enough.

Alternatively, MTSS can be used for decreasing the required physical memory of an embedded system by reducing the initial memory allocated to each of the tasks and recovering the deficit by sharing stack with other tasks.

The overheads of MTSS are low: the run-time and energy overheads are 3.1% and 3.2% on an average. These are tolerable given reliability is the most important concern in virtually all systems, ahead of other concerns such as run-time and energy.

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, run-time checks, reuse, data compression, stack overflow, heap overflow, reliability, cactus stack, meshed stack, virtual memory

1. INTRODUCTION

Memory overflow can be a serious problem in computing, but to different extents in desktop and embedded systems. In desktop systems, virtual memory reduces the effect of memory overflow because hardware-assisted virtual memory [Hennessy and Patterson 2002] detects physical memory overflow and provides swap space on the disk upon overflow. Further, virtual memory provides efficient sharing of physical memory among processes because it discontiguously allocates fixed-sized blocks of memory, called *pages*, as memory is demanded by each process. This obviates the need for contiguous physical memory allocation for each process, which in turn, reduces wastage of memory and enables processes to share the same physical

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

memory space.

This work seeks to provide the same memory-sharing functionality of virtual memory in software because a great majority of embedded processors (we estimate over 95%) have no virtual memory [Kleidermacher and Griglock 2001]. Examples of embedded processor families that lack virtual memory support include Motorola's M68K series; Intel's i960; ARM's ARM7TDMI; ARM7TDMI-S and ARM966E-S; TI's MSP430; Atmel's 8051; Analog Devices Blackfin; Xilinx's Microblaze; Renesas M32R; and NEC's NEC750; among others. It is easy to see why: virtual memory hardware leads to an increasein the system's energy use, real-time bounds, area cost, and design complexity. Typically, it checks that the address of *every* memory access is within segment bounds and translates the address using a Translation Look Aside Buffer (TLB). The energy cost of these frequent tasks can be prohibitive [Panda et al. 2001]. Indeed, it was shown in a study [Montanaro et al. 1996] that virtual memory alone comprised 17% of their embedded system's total energy consumption, which is equivalent to a 20.5% increase in energy use from virtual memory.Even a simpler virtual memory scheme providing segment protection but no virtual-to-physical address translation is not widely used because of its energy cost. Additionally, this simplified scheme is not capable of sharing memory among processes, which is our goal. A second major drawback of virtual memory is that it can dramatically degrade real-time bounds because any memory reference can potentially cause a TLB miss. These drawbacks are well-known [Durrant 2000] explaining why virtual memory hardware is rare in embedded processors.

While the area cost of virtual memory is becoming less of a concern, energy and real-time bounds are becoming increasingly important. We see nothing in technology trends to indicate that the normalized cost of virtual memory, in energy or real-time bounds, will decrease over time.

Lacking virtual memory, any embedded system will encounter a fatal error if its memory footprint exceeds the physical memory by even one byte. Therefore, for correct execution, the designer must ensure that the total memory footprint of all the applications running concurrently (*i.e.*, running or preempted before completion) fits in the available physical memory at all times.

Unfortunately, accurately estimating the maximum memory requirement of an application at design time is difficult, increasing the chances of memory overflow. To see why, consider that the application data is typically divided into three segments: global, stack and heap. The size of the global segment is fixed at design time whereas the stack and heap grow at run time. Let us consider stack memory first. 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) functions called through pointers, which also result in a partially unknown call-graph; (iv) languages, such as GNU C and `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 allow a block of a run-time dependent size to be allocated on the stack; and (vi) interrupts, since their handlers allocate stack space that may be difficult to estimate. In all these cases, estimating the stack size at design time is difficult. Indeed, in cases (i), (iv) and (v) the maximum stack size is dependent on the input data and is unknowable at design time. As an example, a recursive function invoked with a command line argument can lead to an unbounded stack.

Estimating the heap size at design time is also difficult. The heap is typically used for dynamic data structures such as linked lists, trees and graphs whose sizes are highly input-dependent and thus, unknowable at design time.

Lacking precise design time estimation of stack and heap sizes, the usual industrial approach is to run the application on different data sets and observe the maximum sizes of the 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, memory overflow is still possible. Sometimes the memory requirement is multiplied by a safety factor; however, the factor is often limited for cost reasons and it still does not give any guarantees to prevent overflow.

The problem of out-of-memory faults has serious consequences on the reliability of embedded systems. Lacking virtual memory support, memory overflow in an embedded system can lead to loss of functionality of a controlled system, loss of revenue, industrial accidents and even loss of life. In our past work [Biswas et al. 2006], we looked at the problem of overflow detection and the reuse of memory *within* a task in order for the application to continue execution. This work extends the past work to reuse stack memory available across different tasks in an embedded system. We propose MTSS (Multi-Task Stack Sharing), a scheme to share stack space after overflow in multi-tasking systems. This is a significant contribution since multi-tasking is dramatically rising in embedded software development [Lamie 2000; Moore 2001] and there is a large amount of memory available for reuse across different tasks.

Since MTSS builds upon our previous work [Biswas et al. 2006], it gains the benefit of memory overflow detection. This allows for the possibility of remedial action or a graceful exit if the recovered space is not enough to complete execution, unlike conventional systems, where stack memory overflow goes undetected and results in a fatal crash. Remedial action may include safely shutting down the controlled system, flagging a warning sign, or transferring control of the controlled system to a manual operator. Such remedial action may be invaluable in safety-critical embedded systems.

The rest of the paper is organized as follows. Section 2 overviews MTSS. Section 3 outlines related work. Section 4 describes the run-time checks inserted by our compiler to detect stack overflow. Section 5 describes optimizations to reduce the overhead of run-time checks. Section 6 details our scheme for reusing stack space across different tasks. Section 7 considers the impact of certain real-world issues on our scheme. Section 8 specifies the systems to which MTSS applies. Section 9 describes our experimental platform. Section 10 discusses the results. Section 11 concludes.

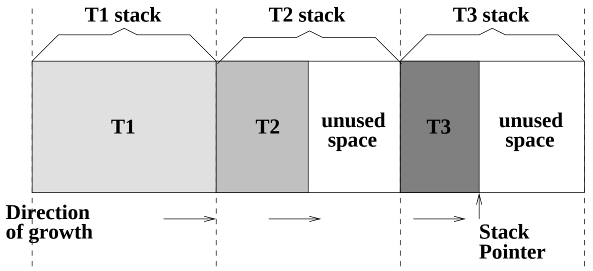

Fig. 1. Example showing wasted space in a simple version of a cactus stack layout. T_1's stack is full but it cannot use the unused space in the stacks of T_2 and T_3.

2. OVERVIEW OF OUR SCHEME

Our scheme is based on the observation that the most commonly used stack layout for multi-tasking systems, called a *cactus stack* [Moore 2001; Pizka 1999; Shantanu Sardesai and Dasgupta 1998], wastes a significant amount of memory. In its simplest version, a cactus stack allocates a separate stack for each task in the system. Figure 1 shows a system with such a stack with each of three tasks T_1, T_2 and T_3 allocated a separate stack space. The space wasted in this layout is immediately apparent, for example, when T_1's stack is full, the free space in the stacks of T_2 and T_3 cannot be used to avoid the overflow in T_1. The goal of MTSS is to enable any overflowing task to use stack space available anywhere. With MTSS the overflow will be postponed and hopefully avoided, thus increasing system reliability.

MTSS also applies to the more general case of a cactus stack where tasks that do not run forever are allocated space only during the time they are active (running, preempted or waiting for I/O). Here, tasks are spawned when triggered by internal milestones or external events and their space is freed upon termination. Since tasks spawn other tasks, the resulting tree-like representation of spawn relationships inspires the cactus-stack name. Here, MTSS enables stack-sharing among currently active tasks, rather than among all the potential tasks in the system.

MTSS recovers wasted space using an innovative *paging system* that has four steps. First, run-time checks are inserted at the beginning of each procedure to check for stack overflow. We show that many of these checks can be combined with others using the *rolling checks optimization* to reduce the overhead while retaining the guarantee that all overflows are detected. Our version of the optimizations are an improved version of those in our earlier work [Biswas et al. 2006]. Second, if an overflow is detected, then a fixed size block of memory called a *page* is allocated in the free space of another task that has free space. The page is allocated in the stack space at the far end of the stack base so that the chance that the native stack in that space will itself overflow is reduced. If multiple tasks have free pages, then the task with the least number of already allocated overflow pages is selected for the discontiguous growth of the overflowing stack. Third, if the current overflow page(s) is also filled, additional page(s) are allocated using the same scheme as above. Fourth, run-time checks are inserted by the compiler at each procedure return to check if the overflowing stack has withdrawn from the page. If the check succeeds, then that page is released back to the free list of pages. Using this scheme, all the free space is utilizable by any of the tasks in the system.

Our scheme offers the following advantages. First, it meets the objective of reusing memory across different tasks in the embedded system. Thus, a task will not run out of memory if the required amount of free space is available in any other task's stack. This increases the reliability of the embedded system. When only one task overflows, our results show that MTSS, on average, is able to recover 54% of the stack space allocated to the overflowing task in the free space of other tasks. Second, our scheme incurs very little run-time overhead in the common case when no stack in the system overflows. This is because in the common case, only the run-time check for overflow is executed on the entry and return of some procedures (after optimization). Results show that this overhead is less than 3.1% in run-time on an average across various multi-tasking workloads. Furthermore, a task grows in its own native stack until it runs out of space there; thus additional run-time for linking a page is only incurred on an overflow. Third, our scheme offers good real time guarantees since it never incurs a large episodic increase in run-time. Rather, due to fixed- size page allocation, the overhead is spread out over the program with a small overhead every time a page overflows. Results show that the increase in worst-case execution time (WCET) averages less than 37.5% for our benchmarks. This increase in the WCET is modest compared to the increase from hardware-assisted virtual memory, which achieves sharing of space across stacks like our scheme but incurs TLB misses that dramatically degrade the WCET.

In an alternate configuration, our scheme can be used to reduce the physical memory needed for an embedded system without reducing its reliability. In this configuration, the memory provided to each task is deliberately reduced to below what it needs and the deficit is recovered from the stacks of other tasks. Experiments show that MTSS used in this way can be used to reduce the memory required in multi-tasking embedded systems by 15.7% on average, thus reducing the dollar cost of the system.

3. RELATED WORK

The broad impact of this work is the reproduction in software of a portion of the functionality of virtual memory hardware. Virtual memory hardware detects physical memory overflow and provides stack space on disk, if present, upon overflow. Furthermore, it is capable of utilizing *all* the physical memory available in the system, since it performs non-contiguous allocation of each process segment, including stack, making use of fixed size *pages.* Thus, MTSS is *not* useful for systems with virtual memory support. However, hardware virtual memory is unappealing for use in embedded systems because, as mentioned earlier, many systems lack the support for such hardware, and even if they did have such support, the increased CPU, memory resources, and energy consumption associated with its functionality would not be as low as they could be with a software-only solution. Energy consumption is a particular concern since protection hardware is activated for each data and instruction memory access. Moreover, real-time guarantees are a concern for systems using TLBs because of the possibility of TLB misses.

Specialized hardware schemes for providing memory protection in embedded systems have also been devised. The Mondrian Memory Protection (MMP) [Witchel et al. 2002] scheme is a hardware approach designed to provide fine-grained memory protection for systems requiring data sharing among processes. Another hardware approach [Carbone 2004] provides basic segment-level protection without requiring any TLBs, relying only on the permissions capability of the MMU. Similarly, some embedded processors, like ARM926EJ-S, instead of supporting full virtual memory hardware are equipped with a coprocessor known as Memory Protection Unit (MPU) [Jagger and Seal 2000]. The MPU provides protection by dividing the address space into regions with individual access permissions. All these specialized schemes still incur some hardware and energy cost as compared to our software-only scheme and *more importantly, do not provide any way to share stack space among different processes*, which is the goal of this paper.

Several other attempts have been made to reuse memory across different tasks for multi-threaded applications. One such attempt consists of allocating stacks on the heap [Grunwald and Neves 1996; Behren et al. 2003]. In older schemes, which used heap-based allocation of stacks [Bobrow and Wegbreit 1973; Hauck and Dent 1968], the activation records are allocated on the heap, and explicitly deallocated when the procedure returns. Thus, no task runs out of memory, unless there is no space left globally. However, since the granularity of allocation is unequal, these schemes suffer from the increased run-time overhead of allocation (*malloc*) and deallocation (*free*) for *each* procedure call and return. The overheads of *malloc* and *free* are often in the thousands of cycles per invocation because of the complexities of heap management with requested blocks of arbitrary size.

In one of the recent stack-in-heap schemes [Behren et al. 2003] a stack management scheme is implemented that allows high-concurrency desktop servers to support large number of threads without allocating a large contiguous portion of virtual memory for their stacks. In their scheme, a thread's stack is allocated in a small fixed-size heap chunk, and is grown discontiguously into other heap chunks when one is full. This scheme inserts run-time checks similar to our scheme, and exhibit similar dynamic allocation efficiency, due to the presence of fixed-size heap chunks. Four differences of our scheme with respect to [Behren et al. 2003] are as follows: First, our scheme is applied, optimized, and evaluated for embedded systems; their scheme is applicable to desktop servers with virtual memory hardware. Second, our scheme does not incur the extra run-time overhead of discontiguous stack growth unless all the stack space in the task is exhausted, which is rare, while their scheme would incur that overhead whenever the small fixed-size chunks run out, which is more common. Third, our scheme is applied for a different goal, to improve the reliability and physical memory utilization of the system, not their goal of saving on virtual address space and reducing the load on segment tables. Fourth, 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. A quantitative comparison against the Capriccio scheme is presented in section 10.

It is worth mentioning the relation of MTSS to garbage collection (GC) to MTSS. GC [Hertz and Berger 2005] is meant to recover dead heap data automatically, whereas MTSS is meant to share live stack data; thus they have different goals. Nevertheless some have suggested allocating stack data on the heap, and using GC to recover the dead stack frames [Appel 1987]. However, as that paper states, this approach is preferable to stack allocation only when the amount of physical memory far exceeds the data size (7X in their paper). This is very wasteful in memory, and not suitable for resource-constrained embedded systems that we target in MTSS. Indeed this is too wasteful even in desktop systems - we do not know of a single commercial or leading open-source compiler that allocates stack data on the heap like suggested by [Appel 1987]. Of course, GC is very valuable for genuine heap data, as is evidenced by its widespread use, but stack data should be allocated on the stack for efficiency. In this scenario, MTSS is useful for sharing the stack.

Two other attempts have been made to recover unused stack space from non-overflowing tasks in a multi-tasking system. In the first scheme, the run-time data of several parallel tasks is allocated on a single stack, leading to a *meshed stack* organization [Hogen and Loogen 1993]. In this scheme, new stack frames are always generated on top of the stack, even if its parent procedure's stack frame is buried deep in the stack with the frames from other tasks in the middle. For this reason, non-contiguous allocation of stack frames is supported by this methods. If a procedure terminates and its activation record is not on the top of stack then it is not removed, but marked as garbage. Special garbage collectors are then invoked periodically to crunch the stack in place. However, the total run-time with their scheme is higher because of the need for scanning the entire contents of stack memory. A scan of memory is needed to correctly update pointers, as in any copying garbage collector. No such scan of memory is needed in our scheme since our scheme never copies any value in memory.

In the other attempt for reusing memory across tasks, each thread shares stacks from a stack pool [Wong and Dageville 1994; Moore 2001]. In [Wong and Dageville 1994], the authors propose a hybrid stack sharing scheme in which each thread is allocated a stack from a stack pool containing a fixed number of stacks. The size of each stack in the stack pool can be set by the user. When the number of threads are less than the number of stacks in the stack pool, it is the same as the cactus stack. However, in the common case when the number of threads is more than the number of stacks in the stack pool, all the threads share the stacks from the stack pool, leading to greater memory savings. However, when the number of *active* threads exceed the number of stacks in the stack pool, then on a context switch, in addition to the processor state, the whole contents of the task stack also need to be saved in the heap memory and similarly restored when the thread becomes active. This leads to increased run-time overhead and a dramatic degradation in real-time bounds. In addition, the hybrid stack sharing scheme does not fully accomplish our objective in that an overflowing stack cannot use space available in other stacks in the stack pool since no mechanism for sharing across stacks in the stack pool is implemented.

MTSS is applicable to all systems that have blocking tasks. However, it is not applicable to systems where only a single stack is used [Baker 1990]. This scenario is discussed in more detail in section 8.

Methods for estimating the maximum depth of the stack [Regehr et al. 2003; Brylow et al. 2001] 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 perfectly estimated and no heap data is present then stack overflow cannot occur. The compiler should turn off our scheme for such programs. However, the presence of 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 in that suite have heap data. In conclusion, our scheme is valuable in three cases: (i) if the stack size cannot be estimated because of the difficulties with estimation mentioned in section 1; (ii) if the estimates are too conservative to be acceptable; or (iii) if heap data is present. In all three cases, our scheme provides good back-up insurance against stack overflow and allows the application to continue execution and in many cases prevent the stack overflow altogether.

MTSS builds upon our previous work in [Biswas et al. 2006], which also uses run-time checks to detect stack overflow and recovers space from within the overflowing task. In our earlier work an overflowing stack is grown in dead global variables and space freed by compressing live variables. Two differences of our scheme with respect to [Biswas et al. 2006] are as follows. First, their scheme recovers space from within a task and makes no attempt to share space across stacks. Thus it has a different goal. Second, although the run-time checks for overflow are shared, the optimizations on run-time checks (the rolling-checks optimization) in this work are a new and improved version of those in our earlier work – our optimizations do not require profile data, whereas those in the earlier work do. This is an important practical advantage in compiler infrastructures. However, the work in [Biswas et al. 2006] is complimentary to our scheme in that it can be combined with MTSS to result in a system that detects a stack overflow using run-time checks and recovers space both within a task and across different tasks, leading to increased system reliability.

4. RUN-TIME CHECKS TO DETECT STACK OVERFLOW

MTSS builds upon the software scheme for detecting stack overflow in our previous work [Biswas et al. 2006]. This section briefly overviews the checks in that paper. To see how stack overflow can be detected, consider that the stack grows only at procedure calls. Figure 2 shows the check that we insert at the beginning of every procedure. Without loss of generality, we assume that the stack grows from higher-numbered addresses to lower. The stack pointer is decremented (not shown) at the start of each procedure by the size of the current procedure's frame. The code in Figure 2 is inserted immediately *after* the stack pointer is decremented. Thus, the check compares the updated stack pointer to the current allowable boundary of the stack. If the check succeeds, then stack overflow has occurred.

Without MTSS, the stack boundary is specified by the cactus stack layout or it is the heap pointer in case the heap is adjacent to the stack in question. MTSS modifies the stack boundary to be the *overflow pointer* of that task instead. The overflow pointers store the upper limit of overflow space for every task and are explained in further detail in Section 6.

The run-time checks are easily extensible to cases where the stack size is known only at run-time, such as with variable-sized stack arrays and stack allocation using *alloca()*. Such cases pose no problems since the overflow checks, themselves, occur at run-time, by which time the stack size becomes known. The details are in our previous work [Biswas et al. 2006].

```
1. if (Stack-Ptr < STACK_BOUNDARY)
2.   call routine to handle stack-overflow condition
3. }
```

Fig. 2. Code inserted at procedure entry for detecting stack overflow.

5. PROFILE INDEPENDENT ROLLING CHECKS OPTIMIZATION

The overheads of the added stack checks in the baseline scheme can be reduced by the *profile-dependent* rolling checks optimization [Biswas et al. 2006]. The intuition behind this optimization is that if a parent procedure calls a child procedure, then, instead of checking for stack space at the start of both procedures, in certain cases, it might be 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. The reduction in overhead can be more than half if the rolled child is called more frequently than the parent. The optimization implemented in [Biswas et al. 2006] is profile- dependent because it considers each function in the order of its frequency obtained through profile information. This ensures that the checks are rolled out of the most frequently executed functions first and the overhead reduction is the greatest. Further, it also uses an estimate of the stack size of the application obtained through profiling to implement the optimization.

The profile-dependent rolling checks optimization reduces the overhead of run-time checks but suffers from the following drawbacks. First, profile data is hard to obtain in many applications before deployment. Second, some compiler infrastructures do not provide support for automatic profile collection and use. Third, a profile-dependent analysis can yield poor results on other data sets which may have significantly different access patterns than the profiled data sets. Fourth, rolling checks out of library functions becomes hard, because the profile information within a library function can be very different across different applications and data sets.

In this paper we propose a profile-independent scheme to implement the rolling checks optimization. This scheme only depends on the application call graph and the static stack frame sizes of each function. A profile-independent rolling checks optimization scheme can handle library functions easily and does not suffer from the drawbacks described above. Like the older version, this new version also retains the guarantee that all memory overflows are detected by the checks.

Before we describe the implementation of our rolling checks optimization, let us consider two scenarios in which rolling the checks is not legal: these must be checked beforehand. First, 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. Similarly, if the child is called through a function pointer, then the child's check cannot be rolled. Second, rolling checks can be permitted inside of recursive cycles in the application program but not from inside recursive cycles to outside. In the latter case a recursive child can call itself multiple times, making rolling to the non-recursive parent invalid.

We now list the three components of our rolling-checks optimization. The first optimization is based on a new compiler analysis called *certainty analysis*. Certainty analysis aims to prove if one procedure always calls another procedure. The intuition behind this analysis is that the call graph represents potential calls, not actual calls. Therefore, it is possible that for a particular data set a parent may not call a child procedure at all. Then, rolling the child's check to the parent may declare a premature out-of-memory condition in the parent when none would have occurred otherwise. However, if a procedure f certainly calls g then the check in g can be rolled into f with no fear of premature declaration, reducing the overhead of the program. In case a procedure has multiple parents, its check can be rolled only when *all* the parents call the procedure certainly. To find whether a static call from f to g is dynamically certain we use post-dominator analysis, a well-known standard data-flow analysis in compilers [Appel and Ginsburg 1998]. In particular, f certainly calls g if the call site to g in f post-dominates the entry to f[1]. This optimization can be transitively applied to a chain of calls. For example, when f calls g and g calls h, then the checks for both g and h together can be rolled to f provided both calls are certain.

The second of our rolling-checks optimizations is the *zero-size optimization*. This optimization states that a procedure's check for overflow can be removed if it allocates no stack space (*i.e.*, its stack frame size is zero). Such a procedure arises when (i) all its parameters and local variables are register-allocated by the compiler and (ii) the procedure is a leaf procedure (one with no procedure calls inside it). In the latter case the return address is maintained in a register and is not saved to memory. Such procedures are fairly common in optimized GCC compilation of large C benchmarks, as our results show.

The third and final of our rolling-checks optimizations is the *limited-size optimization*. It rolls checks from a function whose frame size is less than a defined threshold of K bytes to its parents. If K is small (*e.g.*, 32 bytes) then it can be added to the check of each parent function that already has a run-time check without a large penalty of premature overflow declaration. Even if the parent does not call the child and an overflow is declared prematurely, the total amount of stack memory remaining must be less than K bytes. Hence, overflow will be declared only when the memory has $\leq K$ bytes free, *i.e.*, when the memory is nearly full, mitigating the effects of premature declaration. Our choice of K is investigated in the results section. This optimization can be cumulatively applied to a chain of calls. For example, when f calls g and g calls h, then the checks for both g and h together can be rolled to f provided the sum of their frame sizes $\leq K$ bytes. Further, care is taken to ensure that if a function has its check rolled to its parent (*e.g.* due to certainty optimization), then the check from its *limited-size* child (child with a stack frame size $\leq K$ bytes) is not removed.

Next, we discuss the order in which each of the rolling-checks optimization can be applied. First, we apply the certainty optimization, *i.e.*, roll checks out of functions that are certainly called by their parents. This optimization decreases the applicability of the limited-size optimization because it increases the required frame size of functions that are parents of limited-size calls. This can convert a limited-size function (frame size $\leq K$) into a non-limited-size function. Further, if a check is rolled out of the function due to certainty then its check cannot be rolled out of its limited-size children. On the other hand, applying the limited size optimization

[1]Program point y in a program is said to post-dominate program point x if every path from x to the exit of the program always goes through y.

first can require a few functions to have run-time checks (because their children's checks are rolled inside them) even though they are certainly called by their parents, thereby decreasing the applicability of the certainty optimization. This is a tradeoff and we chose the former option because of ease of implementation. Second, we apply the zero size optimization. This optimization is independent of the other two optimizations and can be applied in any order since zero-size procedures are leaf procedures and do not call any other function. Further, rolling their checks inside their parents does not change the applicability of other optimizations since the frame size of parent is unaltered. Finally, we apply the limited-size optimization, rolling checks out of limited-size children whose parents already have run-time checks, unless it is possible to recursively roll both the checks to the parent's parent (this is possible only when the sum of frame sizes of both parent and child is $\leq K$ bytes).

Now, we describe the overall implementation of the rolling checks optimization. First, the top level routine considers all the functions in the application in the order in which they appear in the application binary. Second, for each function we check if the run-time check can be legally rolled to all its parents by testing for two scenarios (mentioned earlier) in which rolling is not legal. Third, we apply the three rolling checks optimizations in the order described in the previous paragraph. Fourth, our compiler produces an output file that lists the functions that contain run-time checks after optimization along with their effective frame sizes. The effective frame size of a function is the sum of its own frame size, the maximum frame size among its rolled children and, in case the function was the parent in any limited-size optimization, then the user-defined limited-size threshold K is also added. This file is given as input to the MTSS compiler, which recompiles the application binary with the rolling information, inserting checks in appropriate functions.

A detailed pseudo-code of the rolling checks optimization appears in [Middha 2006].

The rolling checks optimization retains the guarantee that all stack memory overflows are detected by the optimized checks. Without optimizations, each function has an overflow detection check; thus, all stack overflows will surely be detected in the base case. Further, each of the three optimizations removes checks only when they are unnecessary (when no overflow can occur), as detailed in the description of each optimization. Hence, the optimized system too detects all stack overflows.

6. MULTI-TASK STACK SHARING

This section presents our scheme for reusing stack space across different tasks. When a stack overflow is detected by the run-time checks in section 4, MTSS allows the overflowing stack to grow in the free space available in the stacks of other tasks. The scheme is implemented as follows: First, run-time checks are inserted by the compiler to detect stack overflow in each task. Second, if an overflow is detected in a task, then a fixed block of memory called a *page* is allocated in another task's stack that has free space and the overflowing task is grown into it.

Our basic scheme is best understood with the help of an example. Figure 3(a) shows the normal behavior of the system in which none of the three tasks T1-T3 are out of memory. Figure 3(b) shows the snapshot of the system when T1's

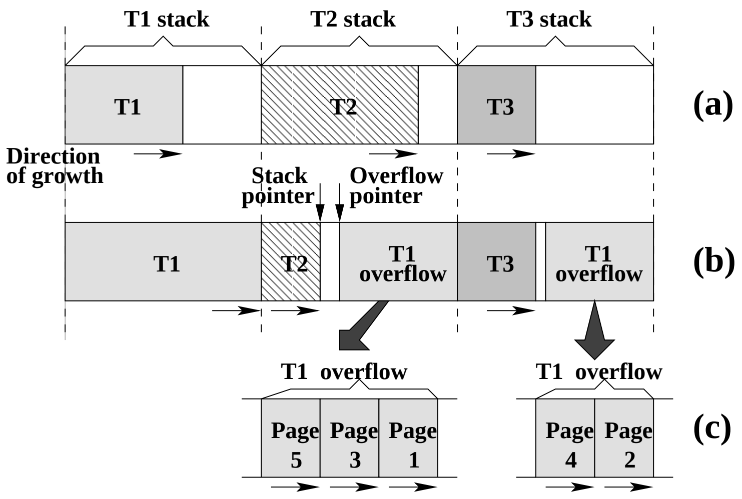

Fig. 3. Example showing reuse across tasks (a) Normal operation of Cactus Stack (b) Overflow handling in MTSS; and (c) Magnified view of overflow space

stack has overflowed its bounds into space in other tasks. Figure 3(c) shows a magnified view of the overflow space in Figure 3(b). Let us now consider the steps taken by our scheme when T1's stack overflows. Since free space is available in T2, page 1 is allocated in it and the stack is grown there. Thereafter, pages 2 to 5 are allocated alternately in the remaining space in T2 and T3 since, when a page is allocated in one, the other becomes the stack space with the least amount of overflow space. In this way, the overflow pages are distributed equally among the stacks with free space, reducing the chance that the native stacks with free space will also themselves overflow soon. If T1's stack overflows again, then the system is declared to be *out-of-memory*.

To implement the scheme, we use the following data structures. First, the set of stack pointers for inactive (context-switched out) tasks is stored as an array in memory. This information is maintained by the operating system, and it allows the active task to access the other stacks upon overflow. Second, an array of *overflow pointers*, one per task, is also maintained. The overflow pointer for a task stores the upper limit of the overflow space for that task. The free space available in a task stack is the difference between its stack pointer and overflow pointer. As an example, the overflow pointer of task T2 is shown in Figure 3(b). Third, an array of *overflow_started* global boolean variables is also maintained with one element per task. This variable is set to true if the task overflows its native stack bound and it is set to false when the stack recedes back to its native space.

To implement MTSS, the stack check at the beginning of a procedure is modified from that in Figure 2 to that in Figure 4. As shown in Figure 4 the constant STACK_BOUNDARY in Figure 2 is replaced by the *overflow pointer* for that particular task, which forms the upper limit on the overflow space for that task. Furthermore, if the task is already overflowing, then this condition is also detected and handled. This is implemented by checking whether the *overflow_started* variable is asserted or not.

```
1. if ((Stack-Ptr < Overflow-Ptr[current-task-id]) ||
       (Overflow-Started[current-task-id])) {
     /* Stack Overflow detected or already in overflow page */
2.   Call routine to handle stack-overflow condition
3. }
```

Fig. 4. Code inserted at procedure entry for detecting stack overflow with MTSS.

```
1. if (Overflow-Started[current-task-id]) {        /* Already in overflow mode */
2.   if (Stack-Ptr + Size-of-current-stack-frame > Overflow-Pointer[overflow-task-id])
         /* Stack has receded from the overflow page */
3.     Overflow-Pointer[overflow-task-id] = Overflow-Pointer[overflow-task-id] - pagesize
4. }
```

Fig. 5. Code inserted at procedure exit for receding the overflow pointer.

Once an overflow is detected, our scheme allocates a fixed block of memory (*page*) to grow the overflowing stack. The method of choosing the free pages is described as follows: First, if there is only one task with free pages then that task is chosen for growing the overflowing stack. Second, if there are multiple tasks having free pages then the task with the least value of already allocated overflow pages is chosen for discontiguous growth of the overflowing stack. This heuristic tries to minimize the chances that the task with free space will itself overflow in the future because of other tasks occupying its space. The heuristic works well as the results show.

When a task is in overflow space, the stack pointer of the task is compared against the page boundary instead of the *overflow pointer*. Thereafter, if the stack overflows in the page, then additional pages are allocated using the same scheme. This is also the reason why the second condition for checking the *overflow_started* variable is added in the check for detecting stack overflow in Figure 4 since page overflows need to be detected for overflowing stacks.

Once the out-of-stack condition is detected by the run-time checks, discontinuous stack growth is achieved by changing the original stack pointer to the near end of the overflow page. Thus, the stack pointer is set to *Overflow-Ptr[overflow-task-id] + pagesize*, where overflow-task-id represents the ID number of the task where MTSS grows the overflowing stack.

MTSS also requires the incoming arguments for a called procedure to be copied to the overflow page in the case of an overflow. Consider that without MTSS, in most compilers it is the job of the parent procedure to write values shared with its child at the end of its stack frame – these are the child's return address, old frame pointer and any arguments passed through memory. After the call, the top of the parent's frame overlaps with the bottom of the child's frame, thus allowing the child access to these shared fields. However, with MTSS, when an overflow occurs the child is not contiguous with the parent; thus unmodified accesses to the shared locations are no longer correct. To preserve correct functionality, upon overflow MTSS copies the shared values from the top of the parent's frame to the bottom of the child's frame which are not longer contiguous with each other[2]. Since this

[2]It is safe to do this copying even when addresses rather than values are copied in the arguments. This is because MTSS never moves any object, once it is allocated, for its entire lifetime. Once a variable is allocated in a parent procedure's frame, its address can be safely copied to its child procedures since the variable is never moved until it dies (when the parent returns).

code is executed only in the extremely rare case of overflow, it does not slow down the common case of no overflow.

When the run-time check at the start of procedure f() detects an overflow, it adjusts the stack pointer to an overflow page, copies the incoming arguments to the overflow frame and increments the overflow pointer. At this point, the procedure f() is ready to run oblivious to the overflow. No further action is needed to allocate the page since that is accomplished by incrementing the overflow pointer.

Receding the stack pointer Upon procedure return, MTSS must ensure that the stack pointer is moved back correctly even when the parent stack frame is discontiguous with the current stack frame. Thus simply incrementing the stack pointer at the procedure return is not enough. (Recall that the stack grows from high addresses to lower.) Fortunately most compilers (such as GCC) already maintain the *old frame pointer*, which is the value of the stack pointer of the parent frame. In these compilers, the stack pointer is assigned to the old frame pointer just before each procedure's return. *In this case, MTSS needs to do nothing since this way of receding the stack pointer is correct even when the parent stack frame is discontiguous with the current stack frame.* In a minority of compilers there is an option of eliminating the old frame pointer – MTSS must disable this option, which ensures that the old frame pointer is always used.

Receding the Overflow Pointer When a running task returns from a procedure, its stack frame is de-allocated. If that frame was allocated on an overflow page, then upon return it is possible that the overflow page becomes empty. If so, the empty page must be recovered as free space by receding the overflow pointer. For example in figure 3(c), when task T1 recedes from page 5 upon a procedure return, then the empty space in page 5 must be recovered. We observe that recovering the space in page 5 can be done by receding the overflow pointer for task T2 in figure 3(b) to the right by the length of a page. Further, *since a page can become empty only upon procedure returns, the check for receding the overflow pointer needs to be inserted at compile-time at every procedure return in each task.*

The check that MTSS inserts at each procedure return to recede the overflow pointer is shown in figure 5. Line 1 checks if overflow has started. If it has, then line 2 checks if the stack pointer incremented by the current procedure's frame size is greater than the overflow pointer. This is the condition of stack underflow in the page since the left side of the $>$ is the value of the stack pointer after the current frame is removed. (Recall that greater addresses are to the *left* in figure 3.) This check should be inserted before the stack pointer is receded as described in the previous section using the old frame pointer. If the check on line 2 detects an underflow, then line 3 recedes the overflow pointer to the right by the length of one page as shown.

Holes in the Overflow Space If multiple stacks overflow their bounds, then the result could yield *holes* in the overflow space, as depicted in figure 6. To understand figure 6, let us consider that there are three tasks T1-T3 in the system. Let us further assume that task T1 overflows its bounds and starts growing in page P1 in task T2 as shown in Figure 6(a). Subsequently, T2 also overflows its bounds and starts growing in page P2 in task T3. Thereafter, both T1 and T2 overflow their bounds once again leading to the allocation of pages P3 and P4 in task T3,

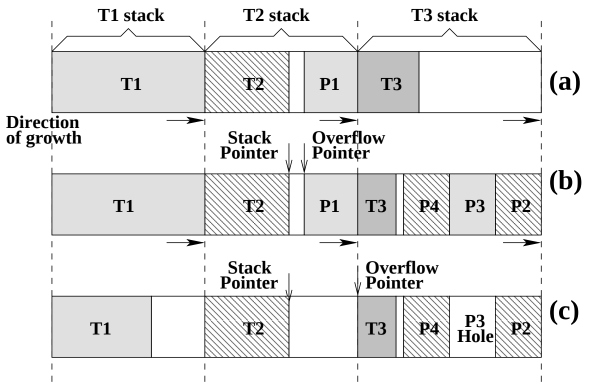

Fig. 6. Example showing holes in overflow space (a) T1 overflows in T2 (b) T1 and T2 overflow in T3; and (c) T1 recedes leaving holes in overflow space

as shown in Figure 6(b). Now, if the stack of T1 recedes back to its native space, it vacates pages P1 and P3. This is shown in Figure 6(c). Of these, page P3 is called a *hole* since it is not at the overflow-pointer-end of the overflow space, but, rather in the middle. For this reason, it cannot be reclaimed by receding the overflow pointer and it must be reclaimed through a different mechanism. We reclaim holes by classifying every page in a task stack as either free or filled. This information is maintained as a bit-vector per task, with a bit for each page. A value of 1 signifies that the corresponding page is filled and a 0 indicates that it is free. Subsequently, before allocating a free page, we traverse this bit-vector to check for the presence of holes and allocate free pages in holes, if possible, before moving upwards in the stack space. Although this situation does not arise if only one task overflows in the system, it can happen and must be handled as above. In our experiments, we observe that the presence of holes is rare. Due to the possibility of the holes in the overflow space, the body of the check in Figure 5 is modified so that the *overflow pointer* is receded only when the receding stack page is at the overflow-pointer-end of the overflow space.

Multiple-Page Allocations The base scheme to share the stacks among multiple tasks is enhanced by incorporating multiple page allocations. Multiple page allocations are required if the procedure frame of the overflowing task is larger than a single page because a procedure frame cannot be allocated discontiguously. If it were, then the addressing mechanism of stack variables would have to be changed upon overflow leading to an extremely complex implementation. Multiple page allocations in our scheme are implemented as follows. First, the required number of pages are calculated by dividing the frame size with the page size and taking the ceiling. Second, each task is searched for the availability of multiple pages instead of a single page. If the overflow space contains holes, then the scheme looks for the availability of contiguous holes equal to the number of pages required. Third, the check for page overflow is modified to handle multiple pages, *i.e.*, the stack is now declared to have overflown its page, if it grows by an amount equal to the number of pages allocated to it. Fourth, the overflow pointer is grown and receded by number of allocated pages rather than a single page.

Our scheme declares a system to be out of memory if there is no task in the system that has a number of pages corresponding to a procedure frame available contiguously, even though the total space available discontiguously might be larger. We do not consider compaction of holes to create more space because this would adversely impact the real time guarantees.

Choice of Page Size Next, we discuss why allocating fixed-size blocks of memory is advantageous for our scheme and the choice of page size for our scheme. Allocating fixed size blocks of memory gives us at least three advantages over variable-sized allocation. First, variable-sized allocation leads to *external fragmentation* (holes in the memory of a non-desired size). This results in increased run-time for allocation upon overflow as compared to a fixed-size allocation since allocating memory requires a scan through all the holes in order to determine a fit. Second, for variable-sized allocation a mechanism to merge holes, such as compaction is usually also needed to limit the number of small, useless holes. This will severely degrade the real-time guarantees of the reuse scheme. Third, if the variable-sized allocation scheme allocates exactly the amount of stack space required by the overflowing procedure, then the number of page overflows may increase. For example, if the overflowing procedure in turn calls another procedure, it will result in another page overflow. On the other hand, allocating additional memory than required results in wasted space and makes the implementation more complex.

With fixed size allocation, page size is an important consideration. Both small and large page sizes have their own advantages and disadvantages, as in hardware virtual memory, but with different tradeoffs. Fixed-size allocation leads to *internal fragmentation* (space wasted within a page if it is too small to be used by the next stack frame). Smaller page sizes increase internal fragmentation as compared to larger page sizes and worsen the real-time guarantees of the system. This is because the probability of a page overflow increases as the page size reduces. This also leads to increased run-time overhead in the presence of stack overflows. However, smaller pages are better able to utilize the remaining free space in a stack because it is possible to allocate an overflow page even if the space remaining in a task stack is small. Our experiments explore the choice of page size further.

Re-using Heap for Stack Our method can be easily extended to allow for reuse of the heap when a stack frame overflows and there is no stack space available across all the tasks in the system. In a multi-tasking system, the heap is shared by all the tasks; therefore, we can inherit the scheme proposed in our previous work [Biswas et al. 2006] that allows an overflowing stack to be grown discontiguously in the heap. Since the method to reuse the heap is inherited from previous work, to be fair, we do not count the space recovered from the heap towards the benefit from our method in our experiments.

Alloca Function Calls *Alloca()* library function calls allocate a run-time dependent amount of memory, specified in their argument, on the stack frame of the calling procedure. They are handled by adding their size argument at run-time to the frame size, and requesting the resulting size from MTSS. Since the size of a requested frame in MTSS is only needed at run-time, this poses no problem.

The above solution does not work when *alloca's* argument is not only run-time dependent, but computed inside the function, since this computation cannot proceed until the frame is allocated, which has not happened yet. We handle this case by converting *alloca* into a heap object. This increases the run-time overhead. Fortunately, none of the 29 benchmarks in the MIbench suite had such an occurrence of alloca function, nor do any of our 24 benchmarks. Hence, we can assume that this is an extremely rare scenario, and the increased overhead would be negligible in practice for most applications.

Alternative with No Initial Stack An alternative implementation of the scheme consists of giving zero bytes to each task stack in the beginning, and then to *demand* page in stack blocks as necessary from a common stack memory pool. However, this scheme will have the following disadvantages: First, it will incur increased run-time and energy overhead as the number of page overflows will increase. Second, it will lead to increased fragmentation of memory generating more holes. This is because memory will now be allocated from a common pool on procedure calls, and freed on procedure returns, which will depend on the control flow of each task, leading to the generation of additional holes. This will reduce memory utilization. To offset the reduction in memory utilization, compaction of holes might be necessary, which will spoil the real time guarantees. Consequent to these drawbacks, this alternative with a zero-size initial stack is not used by MTSS.

7. REAL WORLD CONSIDERATIONS

Dynamic Tasks and Multithreading MTSS can be extended to handle the creation and deletion of dynamic tasks in the system. This is implemented as follows: First, the operating system is modified to notify our system about the creation and deletion of new tasks. Second, the algorithm is modified to handle variable number of tasks while considering tasks for sharing. Third, a pool of stack space is maintained for dynamic tasks. Any incoming dynamic task can be allocated any amount of initial space – an estimate can be used if available, or simply one page can be conservatively allocated at the cost of more frequent future overflows. The same scheme can be used for multi-threading, which corresponds to *spawning* a new task at different places in the program, thereby creating a dynamic task in the system or *joining* a spawned task, thereby deleting a task from the system.

Communicating Tasks MTSS does not impact the correctness of implementing communicating tasks. To understand this, consider that there are primarily two methods of intertask communication [Avi Silberschatz]. First, tasks may use *shared memory* as a means to exchange data.The shared memory is located in the memory space of one task, which other tasks, if permitted can access. This shared memory space is never allocated as part of the stack segment of the memory; instead, it is similar to the global segment in its characteristics. Indeed the stack only stores local variables which always are restricted in scope to only one process. *Since MTSS only modifies the stack layout and has no impact on shared global data, the correctness of the implementation remains unchanged.* Further, since MTSS does not touch the shared memory segment, no additional synchronization problems are introduced.Second, tasks can use *message passing* as a means to exchange data. In this mode, explicit send and receive messages are exchanged for communication. This mode is primarily used when the two tasks reside on different machines across the network. MTSS can be used with message passing as well since all data, including stack data, is addressable only by its local process with message passing. Thus MTSS can be used to share the stack space with any other tasks on the same processor. Since MTSS maintains send/receive buffers on the stack as contiguous blocks, communication routines accessing those routines work unchanged with MTSS.

Simultaneous Access and Synchronization Since MTSS handles multiple tasks, deadlocks and race conditions can occur when shared variables are accessed. However, MTSS does not require variables to be protected (*e.g.*, using semaphores) in the common case when the stack does not overflow. To understand why, consider the check shown in Figure 4 to detect overflow. Here, the *Stack-Ptr*, and *overflow_started* variables are local to each task's context; however, *Overflow-Ptr* is shared across multiple tasks and may need to be protected from simultaneous access.

The only potential race condition involving an access of *Overflow-Ptr* can be seen by looking at figure 3(b). In the figure, suppose *Overflow-Ptr[T_2]* has been read by T_2, but thereafter T_2 is preempted by a task T_1 before the comparison `(Stack-Ptr < Overflow-Ptr[`T_2`])` is performed. In this case task T_1 can overflow into task T_2, allocate a page and increment *Overflow-Ptr[T_2].* When T_2 gets the CPU again, it will perform the comparison using the old value of *Overflow-Ptr[T_2].* This can potentially lead to incorrect semantics since T_2's latest stack frame could overlap in memory with an overflow page if there is not enough space between the two.

However, we prove that this incorrect overlap of memory can never happen even in the presence of the race condition above. Suppose the above race condition happens. There are two possible cases of what might happen just when control switches to task T_1. In the first case, there is no space on the stack of T_2 to allocate the pages needed by T_1. In the second case, there is space for the needed pages. If we can prove that correct semantics are preserved in both cases, we are done.

If there is no indeed space for a page in T_2 (first case), then T_1 will read *Overflow-Ptr[T_2]* from memory and realize that there is no space in T_2's stack for the required pages. Hence it will not allocate the pages and the problem scenario cannot occur.

If there is space for required pages in T_2 (second case), then T_1 will read *Overflow-Ptr[T_2]* from memory, find that there is space, and will allocate a page in T_2's stack. However this is not a problem because of a key observation: the check for overflow is inserted *after* a procedure decrements *Stack-Ptr* to allocate space for its frame. Hence, by the time control switches to T_1, the procedure currently executing in T_2 would have already allocated its stack frame and needs no more space. Moreover since T_1 also has space for its required overflow page (by assumption), this overflow page cannot overlap with the stack of T_2. Therefore, here too, no incorrect overlap of T_2's latest stack frame can happen with an overflow page.

Since both cases above are error-free, this proves that no synchronization lock is required to protect the accesses to *Overflow-Ptr[T_2]* in the common case when the stack does not overflow. To see what this means for the code, we make another observation: In line 1 of Figure 4, if *Overflow-Started* is true, the result of the check `(Stack-Ptr < Overflow-Ptr[`T_2`])` is irrelevant to the result of the check. Combining both these observations, we see that regardless of whether overflow has started or not, no lock is needed for accessing *Overflow-Ptr* on line 1 of Figure 4.

In the uncommon case when a task stack overflows its bounds, MTSS requires a mutual exclusion lock to protect *Overflow-Ptr* against simultaneous access. Such accesses occur inside the body of the check in Figure 4 (not shown in Figure). Our implementation uses the *pthread_mutex* type variable available in the *pthreads* library, although any mutual exclusion mechanism can be used. Our ARM experimental platform has hardware support for an atomic test-and-set instruction which reduces the lock/unlock overhead to a few cycles. However, even if hardware support were absent, it would make no difference to the common-case overhead since the locking overhead is not encountered until after overflow. Hence, the overhead of locking is largely irrelevant to the efficiency of MTSS.

Handling Interrupts The interrupt stack is either separate or part of the task stack in any embedded system. When interrupt frames are allocated on the task stack, the overhead of switching contexts is lower but the memory required increases since the stack frames of interrupt handlers need to be allocated on **each** of the task's stacks. This multiplies their space usage by the number of tasks. On the other hand, when interrupts have their own stack the memory requirement is lower but the overhead of switching contexts is higher. MTSS can handle both configurations provided interrupt service routines (ISRs) are compiled with our compiler and the necessary run-time checks are inserted. To understand why, consider that in the case of separate interrupt stacks, if the interrupt stack overflow is detected by the run-time check during the execution of an ISR, then MTSS can start overflowing the interrupt stack in some other task stack by allocating necessary pages as described in Section 6. In the case when ISR's are executed on the task stack, the corresponding task stack overflow will be detected by the run-time check during the execution of ISR and another appropriate task will be selected by MTSS for growing the overflowing task.

8. APPLICABLE SYSTEMS

Background Embedded systems can be typically classified as *real-time systems* or *non real-time systems.* Further, based on the scheduling alternatives, a particular system can be classified as either *preemptive* or *non-preemptive.* In non-preemptive systems, a thread that has started to execute is always allowed to execute until one of two things happens: either the executing thread is terminated, or more commonly the executing thread enters a *waiting* or a *blocking* state, for I/O or by calling a sleep function. In preemptive systems, in addition to the above conditions, a task is preempted whenever a high priority task becomes ready to run. Further, in case of preemptive systems with same-priority tasks the scheduler is invoked within a defined period, and it context switches the currently running tasks with another task that is ready to run (round-robin scheduling). Most real time systems implement priority-based preemptive scheduling, which implies that at every instant of time the highest priority task that is ready to run will be the task that is running.

Similarly, tasks in embedded systems can be classified as *single-shot tasks* or *blocking tasks.* A single-shot task is one that can have only three different states – ready, running and terminated. When it becomes ready to run, it enters the ready state. Once it gets the CPU, it enters the running state. If it is preempted by a higher priority task, it can go back to the ready state and when it is finished, it enters the terminated state. A single-shot task has *no* waiting state – the task does not yield the processor and waits for an event to occur. In comparison, a *blocking* *task* has an extra state: the waiting state. This means that a blocking task can yield the processor and wait for a time or an event to occur, such as an I/O completion message or an external-environment event. Unlike single-shot tasks, many blocking tasks run forever and lower priority tasks can run while the higher priority ones are in their waiting state.

Non-applicable systems MTSS is an approach to share stacks among multiple tasks and is *not* applicable to systems in which a single stack is used. A single stack can be used for multiple tasks if they are all *single-shot tasks*, either preemptive or non-preemptive. To see why, consider that in the non-preemptive case, single-shot tasks run to completion whenever they start. Hence, only one task has a stack at any one time and one shared stack is, therefore, sufficient. The size of the shared stack is chosen to be the maximum required among all the tasks in the system.

Less obvious is the fact that in the case of preemptive systems with single-shot tasks, all the tasks can share a single stack. They can do so by *interleaving* their stack frames in a single combined stack. A scheme that does this for fixed-priority tasks is described in [Baker 1990]. In this scheme, when a task T_1 is preempted by a higher priority task T_2, T_1 continues to hold its stack space and T_2 is allocated space immediately above T_1 *in the same stack*. The only special requirement is that T_1 cannot resume until all tasks occupying space above it have completed. This will always be the case since T_1 will be preempted by higher priority tasks only. Moreover, none of the tasks will enter into a blocking state thus making a single stack feasible. In both these cases of single-shot tasks, since there is only a single stack, MTSS is not needed.

Applicable systems Conversely, MTSS is applicable to *all* systems without virtual memory that have blocking tasks regardless of whether they are preemptive or non-preemptive, whether they are real-time or not and irrespective of their scheduling policy. This class represents a majority of the multi-tasked systems used today. To see why MTSS applies to blocking tasks we need to prove that such tasks cannot share a single stack. This is proved below.

To understand the proof, we consider a system with n blocking tasks, T_1 to T_n in increasing order of priority. Now, consider that at a particular point in time T_i is running and assume that after a certain point in time T_i goes into a blocking state by calling the sleep function or by performing I/O. Since T_i has not finished execution yet, its stack needs to be retained. Next, we assume that T_i is replaced by task T_j $(j < i)$ by the scheduler. When T_i finishes its I/O it becomes active, preempting T_j since T_i has a higher priority. The claim is that a single stack S is not possible in this scenario. Suppose there was a single stack. Then, stack frames for T_j would be allocated immediately above those for T_i in the single stack. When T_i tries to resume execution, it will not be able to grow any further contiguously since the space above it would be occupied by T_j, preventing T_i's execution. Therefore T_i and T_j must have different stacks.

Even in the case of non-preemptive systems, we can arrive at the same conclusion. This is because although T_i cannot preempt T_j, when it becomes active T_j can itself enter a blocking state after some point of time. This will again prevent T_i from beginning execution since T_j occupies the top of the stack. Further, it is practically infeasible to wait for T_j to complete execution (after coming out of its blocking state) before T_i can begin execution because of the significant delays that high-priority tasks such as T_i will incur. This proves that any system with blocking threads requires more than one stack, making MTSS feasible.

Some real time systems implement the scheduler proposed in [Wang and Saksena 1999]. They propose a scheme for scheduling fixed priority tasks with preemption thresholds. This scheme introduces the notion of a preemption threshold in addition to a priority of a task to develop a new scheduling model, which unifies the concept of preemptive and non-preemptive scheduling. They claim that using their model, a set of periodic and sporadic tasks can be efficiently implemented using a small number of event-handling tasks. A smaller number of tasks at implementation results in fewer pre-emptions and context switches. Further, it also results in significant memory savings due to the need for fewer stacks. Thus, their scheme substantially reduces the stack requirement of the system. However, MTSS is still applicable in this system, since it contains more than one stack which can then be shared amongst tasks.

9. EXPERIMENTAL SETUP

This section presents the experimental platform used for evaluating our scheme. We have implemented our scheme inside the ARM GCC v3.4.3 cross compiler [GCC] targeting the ARM7TDMI [ARM 2003] embedded processor. The ARM GCC compiler is suitably modified to insert run-time checks as required by our method.

Since we run multi-tasking applications, we also need the support of an operating system for scheduling the application. We use the μC`linux` operating system [Dionne 1998], modified as needed by the proposed techniques. μC`linux` is a derivative of the Linux 2.0 kernel intended for micro-controllers without Memory Management Units, precisely the systems to which MTSS applies. We use the default scheduling policy for non-real-time systems for scheduling the different tasks in the system, which chooses processes based on their dynamic priority. The dynamic priority is based on the *nice* level of each task and is increased for each time quantum the process is ready to run, but is denied to run by the scheduler. This ensures fair progress among all processes. Other scheduling policies can also be used with MTSS. MTSS is conceptually equally applicable to all scheduling policies for blocking tasks, although the memory recovered may slightly differ because of variations in the exact timings when processes switch contexts. We modify the operating system to provide a new system call that returns the value of the stack pointer of an inactive (context-switched-out) task. This is implemented by saving the value of the stack pointer of a task on a context switch into the array of stack pointers maintained by our method. This information is utilized by our scheme to select the task for growing the overflowing stack.

We use the public domain, cycle accurate simulator for the ARM v5 ISA targeting the ARM7TDMI embedded processor for running the operating system as well as the multi-tasking applications. This simulator is available as part of the GDB v6.3 distribution [GDB]. We enhance the simulator to enable it to run μC`linux` along with the application. Specifically, we add support for I/O modules such as timers and interrupt controllers required by the Operating System. Thus, the overall framework consists of multi-tasking applications running on top of μC`linux`

Workload	**Benchmark**	**Description**	**Lines of Code**	**Allocated Stack (Bytes)**
Automotive	Basicmath	Basic Math	132	1024
	Qsort	Quick Sort Algorithm	78	65536
	Bitcnt	Bit Manipulation	383	1024
	Susan	Digital Image Processing	2183	13824
Security	Blowfish	Block Cipher Encryption	2362	6144
	PGP	Public Key Encryption	34973	65536
	Rijndael	Block Cipher Encryption	1812	1536
	SHA	Secure Hash Algorithm	286	10240
Telecomm	ADPCM	Pulse Code Modulation	759	768
	FFT	Fast Fourier Transform	505	1280
	CRC32	Cyclic Redundancy Check	307	1024
	GSM	Voice Encoding/Decoding	6062	2176
Network	Dijkstra	Shortest Path Algorithm	371	1216
	Patricia	Tries for Network Routing Tables	620	1280
	Treeadd	Recursive sum in balanced B-tree	287	1280
	TSP	Traveling Salesman Problem	603	1856
Mediabench	Histogram	Global Histogram Equalization	243	1180
	Edge-Detect	Image Edge Detection	358	1224
	G721	Voice Compression	1800	1404
	Pegwit	Public Key Encryption	7182	10668
Ptrdist	Anagram	Anagram Searching	674	1444
	Ks	Graph Partitioning Tool	805	2732
	Ft	MST computation	2189	1276
	Yacr2	Channel Router	4001	1648

Table I. Multi-tasking benchmark programs and characteristics

operating system, which in turn runs on top of the ARM GDB simulator. Since we use a full-fledged operating system, our setup accurately models all the software used in a real embedded system.

The energy consumed by programs is estimated using the instruction-level power model proposed in [Tiwari et al. 1994]. In that model, the overall energy used is estimated as the sum total of energy consumed by each instruction, where the energy for each instruction type (opcode) is estimated using synthetic workloads composing only of that instruction in an infinite loop and measuring the current drawn by the circuit. Thereafter no current measurements are needed per application; instead the pre-calculated energy per instruction type is used to calculate the total. Experimental results in that paper show that the approach is quite accurate and has a small percentage error in estimating the energy use. The energy numbers for each ARM instruction are extracted from [Sinha and Chandrakasan 2001]. We modified our simulator to add the energy of each instruction, based on its opcode, to a counter measuring the total energy.

10. RESULTS

This section presents the results for the proposed scheme for reusing stack across multiple tasks in an embedded system. Since real multi-tasking workloads are hard to find[3], the multi-tasking workloads that are used for evaluation are constructed by combining together multiple benchmarks from MIBench, PTRDist and Mediabench embedded benchmark suites. Table I shows the names and characteristics of the resulting workloads that we use for our evaluation. A domain in the embedded benchmark suite (such as automotive domain in the MIBench suite) is combined to form a multi-tasking workload. Each domain targets a specific embedded market, and typical embedded multi-tasking workloads for a domain consist of one or more similar tasks. Hence, combining benchmarks in this way forms a reasonable set for evaluation. We evaluate 6 workloads, each containing four benchmarks, for a total of 24 benchmarks. The first four workloads are from the MIBench suite; the last two are given the names of their suites. Unless otherwise stated, all the results are generated for a fixed page size of 128 bytes.

The initial stack memory allocated to each task as shown in column 5 in Table I, is calculated as the maximum observed stack size across different input data sets. This guarantees that a task does not overflow with its initial allocation of stack for those data sets. We then perform several experiments, in which a task is allocated less stack space than required, causing it to overflow. This activates MTSS, allowing stacks to be shared across all tasks.

Overheads of run-time checks Table II shows the overheads due to the insertion of run-time checks to detect overflow. The second column reports the run-time overhead without any optimization, whereas the third column records the reduced run-time overhead after applying the profile independent rolling-checks optimization proposed in Section 5. Similarly the other columns record the code size overheads and energy overheads respectively. Comparing the different columns, we observe that the run-time overhead reduces from 12.28% to 3.05%; the reduction is significant and makes MTSS a feasible scheme for embedded systems. Similarly, energy overheads are also reduced considerably and go down from 12.82% to 3.18% after applying the rolling-checks optimization. MTSS suffers from increased code size overheads because in each function we insert two checks, one at the beginning of the function to detect stack overflow and another at the function return to correctly recede the overflow pointer as described in Section 6. Both these checks are *inlined* inside each function to reduce the run-time and energy overhead. However, this increases the code size overhead. If code size is important for a particular system then the checks can be *outlined.* Our experiments show *outlining* the checks brings down the code size overhead to less than 2.5%, but increases the run-time overhead to about 5% on an average. Since run-time and energy are usually more important than code-size, outlining is not used. In summary, these results show that the safety run-time checks required for implementation of MTSS are possible with very low overhead.

The *ptrdist* workload has higher run-time overheads as compared to other workloads because of the presence of multiple benchmarks with small-sized recursive functions. Recursive functions lead to the execution of run-time checks for every invocation with few intervening instructions; moreover these checks cannot be rolled as explained in Section 5. Both these factors increase overhead.

[3]None of the publicly available embedded benchmarks such as EEMBC, MIBench and Ptrdist provide multi-tasking workloads

Workload	**Run-time Increase(%)**		**Code Size Increase(%)**		**Energy Increase(%)**	
	Without Optim-ization	With Optim-ization	Without Optim-ization	With Optim-ization	Without Optim-ization	With Optim-ization
Automotive	4.18	3.90	37.69	7.04	4.09	4.03
Security	11.54	2.72	32.44	4.45	13.14	2.17
Telecomm	2.74	0.72	38.25	7.70	2.84	0.48
Network	10.51	2.20	44.66	7.15	10.49	2.23
Mediabench	21.75	2.79	36.03	7.21	22.63	3.52
Ptrdist	22.97	5.97	39.69	6.75	23.72	6.63
Average	*12.28*	*3.05*	*38.13*	*6.72*	*12.82*	*3.18*

Table II. Overheads for Safety Checks

Figure 7 shows the execution overhead of the run-time checks from different components of rolling checks optimization. As described in Section 5, the rolling checks optimization is split up into three parts: the *zero size optimization* that rolls checks out of functions with frame size of zero, the *certainty optimization* that rolls checks out of functions that are certainly called by their parents, and the *limited size optimization* that rolls checks out of functions whose frame size is less than a certain user defined threshold. The first bar in Figure 7 shows the execution overhead incurred when the rolling checks optimization is *not* implemented. The second bar shows the run-time overhead incurred when only the *zero size optimization* is implemented. The third bar shows the run-time overhead incurred when both *certainity* and *zero size optimization* are implemented and the third bar shows the execution overhead when all three optimizations are implemented.

These results indicate that the certainty optimization yields a small reduction in run-time overhead, but that both the *zero size* and *limited size optimizations* have a significant impact. The zero size optimization can eliminate the checks on around 30% of the dynamic procedure invocations since they have a zero-size stack frame. For the threshold value of K=128 bytes, the *limited size optimization* can optimize away checks from another 40% of dynamic procedure invocations on an average.

We observe that the performance of the *limited size optimization* varies significantly with the threshold value K. The *limited size optimization*, as explained in Section 5, rolls checks out of functions that have a frame size of less than K bytes. Therefore, as K increases, the number of functions containing run-time checks will decrease, reducing the run-time overhead. Figure 8 plots the run-time overhead across all the workloads for different values of limited size threshold K. As shown in the figure, the run-time overhead steadily reduces from 4.7% to 3.05% as the threshold increases from K=32 bytes to K=128 bytes. Clearly, if we continue to increase the limited size threshold, the overhead will reduce to zero because all checks will then be rolled to the *main* function in the application. Increasing K will reduce the run-time overhead, but will increase the risk of premature declaration, since MTSS will declare an overflow when the memory has less than K bytes free. In this paper, we use a limited threshold value of 128 bytes for all *library* functions as a good profile-independent compromise value. However, for application functions, since it is trivial to obtain the application stack size by profiling, we use 10% of application stack size as the limited size threshold. This still keeps the chance of premature overflow declartion low, while reducing the run-time overhead.

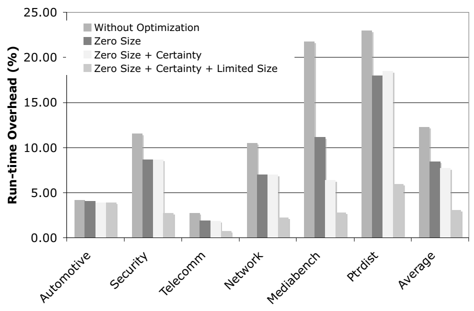

Fig. 7. Run-time overhead contribution to overflow detection checks from each component of the rolling checks optimization.

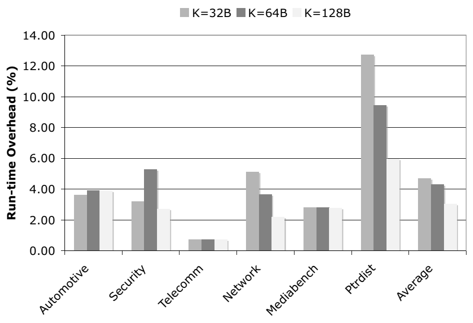

Fig. 8. Impact of varying limited size threshold on run-time overhead.

Comparison with profile-dependent rolling checks optimization Table III shows the comparison of the profile-independent rolling checks optimization with the profile-dependent rolling-checks optimization presented in [Biswas et al. 2006]. The profile-dependent scheme is described in brief as follows: First, it considers all functions in decreasing order of their frequency count. This ensures that the checks are rolled from more frequently called functions first. Second, for each function it checks if the rolling is legal or not. These checks are similar to the legality checks described in section 5. Third, since each function call is a potential function call, which may not be dynamically executed, it rolls checks from function g to f only when the sum of stack sizes of g and f together is less than 10% of the stack size of the application. This reduces the penalty of pre-mature overflow declaration. Thus, the profile-dependent scheme has an equivalent *limited-size* optimization proposed in section 5.

Table III shows that the average run-time overhead of MTSS is 3.05% Vs 5.45% for the profile-dependent optimization. The profile-dependent scheme suffers higher overheads because it does not have an equivalent for the *certainty optimization* proposed in MTSS, in which case the check from a function g is rolled into its parent f, irrespective of the stack frame sizes, if f calls g certainly. Further, it does not explicitly implements *zero-size optimization.* So, a function with a frame size of zero may still have a runtime-check, if the analysis prevents it from being rolled into its parent.

Workload	**Run-time overheads with**	
	(MTSS)	**(Biswas et al.)**
Automotive	3.90	3.90
Security	2.72	5.93
Telecomm	0.72	1.59
Network	2.20	7.21
Mediabench	2.79	2.81
Ptrdist	5.97	11.26
Average	*3.05*	*5.45*

Table III. Comparison of profile-independent and profile-dependent rolling checks optimization

Maximum Satisfiable Overflow (MSO) Maximum Satisfiable Overflow is defined as the maximum amount of stack space that can be recovered for each task expressed as a percentage of the maximum stack size observed across the available input data sets for that task. Figure 9 shows the maximum satisfiable overflow for each task in different workloads. In figure 9, each bar represents the MSO of a particular task in the corresponding workload. The last bar in each workload is the average across all tasks. The last workload, labelled *average*, plots the average MSO per workload for all the workloads. The figure shows that on an average we can recover 54% of stack space per task by reusing stack across tasks. In other words, even if we underestimate the size of a task's stack by 54% on an average, the workload will still run to completion.

The numbers in Figure 9 are collected as follows. The workload is first executed with the stack size for each task equal to its maximum observed requirement for the input data set we use. Thereafter, to calculate the MSO amount for a particular task T, we successively decrease the stack size allocated to T, keeping the stack size for the other tasks unchanged. This activates our method since task T overflows. We then observe if the workload still runs to completion without incurring an out-of-memory fault. This is repeated several times with progressively lesser amount of stack space allocated to T each time, until it no longer runs to completion. The percentage difference between the original stack space allocated to T (with no overflow) and the minimum stack space allocated to T at which the program still runs to completion is the MSO for task T.

As seen from the figure, the space recovered is highly application dependent and depends on both the stack usage of the task and the workload of which it is a part. Furthermore, the space recovered also depends on the *initial stack allocation* of each task, since more space in other tasks will allow more space to be recovered for the overflowing task. We use a conservative safety factor of 1.1 in generating these results; that is, each task is allocated a stack size equal to its maximum observed stack size multiplied by the safety factor. If we increase the safety factor, the MSO will increase. However, a higher safety factor is often not used in embedded systems since their memory amount is limited due to cost constraints.

For some tasks in Figure 9, such as task 1 (*blowfish*) in the *security* workload, the space recovered is 0%. This is because *blowfish* has a total stack requirement of 5632 Bytes, and it contains a procedure of size 4608 Bytes as its main procedure. Procedure frames need to be allocated contiguously on a stack. Thus, if the stack size of *blowfish* is underestimated by even 1 byte, it will require a contiguous space of 4608 Bytes across other tasks to continue execution. No task in the security workload contains 4608 bytes of free space contiguously. Therefore, no space can be recovered for *blowfish.* This also points to the fact that all other tasks in the security workload are using their stack deeply. Therefore, even though *PGP* and *SHA* have large stack sizes of 65K and 10K respectively, the required 4608 bytes cannot be allocated in either of them. On the other hand, for task 3, *rijndael*, in the same workload, we can recover 100% of stack space. This indicates that even if no stack is allocated to *rijndael*, the workload will still run to completion by recovering space from the stacks of other tasks.

Effect of Page Size Figure 10 shows the effect of the page size on the MSO for the

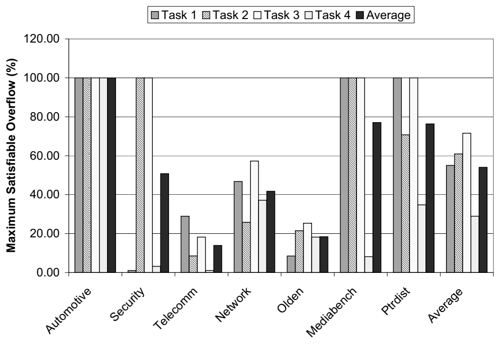

Fig. 9. Maximum Satisfiable Overflow for different tasks in different workloads.

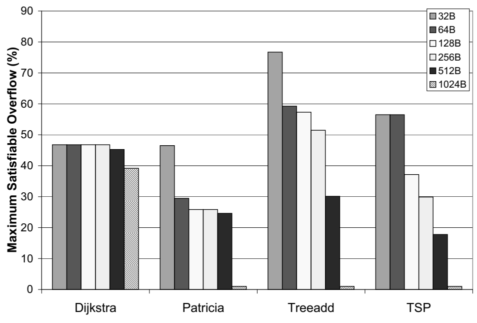

Fig. 10. Effect of page size on MSO for the network workload

network multi-tasking workload. The figure shows that as the page size increases, the MSO of a task decreases in general. This is because smaller pages are better able to utilize the remaining free space in a stack even if the space remaining is small. We use a page size of 128 bytes since it offers reasonable space recovery along with a low overhead.

Proportional Reduction Satisfiability (PRS) An alternate use of MTSS is to decrease the physical memory required by an embedded system while maintaining the same reliability. This is in contrast to its primary use discussed above as a measure to increase reliability for the same amount of memory. When used to reduce the amount of memory, each task is given less stack space than is needed by the input data set. This causes overflow, which is then satisfied by MTSS.

To measure the amount of memory savings in this alternate use, we define the *Proportional Reduction Satisfiability* (PRS) of a workload to be the percentage by which its total stack space can be reduced (by an equal fraction across the tasks) such that the workload still runs to completion with MTSS. To calculate the PRS for a workload, we *proportionally* reduce the stack size of each task in the workload, hence the name *Proportional Reduction Satisfiability.* This process is repeated with successively greater reduction percentages until the workload incurs an out-of-memory fault. The percentage difference between the original stack space allocated to the workload (with no overflow) and the minimum proportional stack space allocated to the workload at which the program still runs to completion is the PRS for the workload.

Figure 11 plots the PRS numbers for different workloads. The difference in the MSO and PRS numbers is that MSO numbers are calculated per task, while PRS numbers are calculated per workload. The figure shows that on an average, across all the multi-tasking workloads, we can recover 15.7% of the stack space needed, reducing the memory cost of the system. The run-time at the PRS configuration will be higher than that for the MSO configuration because more frequent overflows will be incurred, but it is still upper-bounded by the worst-case real-time bounds measured later in this section.

Comparison with non-contiguous stack allocation The Capriccio scheme [Behren et al. 2003] described in Section 3 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. Table IV compares the run-time overheads of our scheme with Capric-

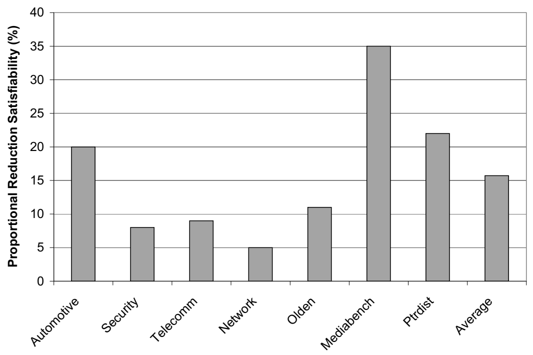

Fig. 11. Proportional Reduction Satisfiability for different workloads

cio[4]. The table shows that the average overhead from MTSS is 3.1% versus 10.7% from Capriccio. Capriccio suffers from higher run-time overhead because of the following reasons. First, it does not pre-reserve stack for any task in the workload. Thus, there is no case in Capriccio in which stacks do not overflow. MTSS, on the other hand, pre-reserves stack for each task (based on its observed stack size across multiple data sets) and incurs the discontiguous growth overhead only upon overflow. Under normal operation, overflow in MTSS will be extremely rare. Second, Capriccio links a new stack (worth an overhead of 27 instructions) whenever an external or a library function is encountered, increasing its run-time overhead. In our scheme, run-time checks are inserted in library functions also (and optimized away using the rolling checks optimization). Thus, only the overhead of overflow detection checks is incurred in MTSS for library function as well.

Capriccio also consumes more memory as compared to MTSS since it requires huge stack chunks to be linked for pre-compiled library functions. The reason for a large chunk is that in their desktop environment, library functions are used by a variety of applications, some without stack sharing; thus they cannot have software checks. Lacking overflow checks, a huge amount of space must be conservatively given for the library function stacks to avoid overflow. In embedded systems, precompiling the libraries with our compiler is feasible since the application set is tightly controlled, and MTSS deployment for all applications is possible. In their paper, a huge stack chunk of 2MB is linked every-time a library function is encountered. They conjecture that as long as threads do not block frequently within library functions, they can reuse a small number of library stack chunks throughout the application. Assuming that Capriccio links only one library stack chunk per workload, Table IV shows the memory consumed by MTSS vs Capriccio. As shown in the Table, Capriccio needs a total stack memory allocation of 2080KB which is 65 times more than that required by MTSS.

One can imagine a modified version of Capriccio which is targeted for embedded systems as opposed to the original one, which is targeted towards desktop

[4]The implementation of the Capriccio scheme assumes an overhead of 6 instructions for the run-time check, 20 instructions for unconditional stack-linking and 27 instructions for conditional stack-linking. These overheads are reported in [Behren et al. 2003]. We use these overheads since the paper does not mention the pseudo-code of the checks and therefore an equivalent implementation on the ARM ISA is not possible. These overheads are likely to be higher for the ARM ISA, which will further increase the run-time overhead of the Capriccio scheme.

systems. The modified one would place a premium on memory and allow discontiguous growth inside library functions by compiling the libraries with run-time checks. When this is done, a huge stack chunk would no longer be needed for library functions, dramatically reducing their memory requirements to close to the actual memory footprint of the libraries, as in our method. However, modification to Capriccio is likely to significantly increase its run-time overhead because a significant number of function calls are library calls. For example, for our set of embedded workloads 53% of all dynamic function calls are to library functions on average. Thus, the overhead of Capriccio will likely approximately double with this modification, from its already high value of 10.7% in run-time. No formal comparison with this scheme is presented because it is a speculative scheme that no one has proposed.

Workload	**Run-time Increase(%) from**		**Stack Memory (Kilo-Bytes)**	
	MTSS	Capriccio	MTSS	Capriccio
Automotive	3.90	7.72	87	2135
Security	2.72	8.18	90	2138
Telecomm	0.72	7.38	6	2054
Network	2.20	14.52	7	2054
Mediabench	2.79	1.77	16	2064
Ptrdist	5.97	17.27	8	2056
Average	*3.05*	*10.69*	*32*	*2080*

Table IV. Comparison of run-time overheads of MTSS and Capriccio.

Real time bounds Unfortunately, we cannot measure the Worst Case Execution Time (WCET) for MTSS. The WCET for any workload is defined as the worst run-time across all datasets. However, it is not easy at design-time to construct a dataset that initiates worst case behavior because of certain language features. A recursive function, for example, can cause the application stack to overflow an unbounded number of times, making it impossible to estimate WCET.

However, in order to get an estimate of the order of WCET we try to measure the WCET for a *particular dataset* for each workload. This is not the true WCET, but it is an attempt to characterize the order of WCET for MTSS. Figure 12 shows the worst-case execution time (WCET) overhead for different workloads expressed as a percentage of the run-time of the unmodified application. The average WCET overhead (for fixed datasets) across all workloads in the system is 37.5%. The actual run-time increase is usually much lower (it averages 3.1% in the common case of no overflow). These WCETs were never actually observed by us; instead, they were calculated using a combination of theoretical analysis and experiments.

The theoretical WCET overhead for each workload is calculated in three steps. First, we calculate the minimum stack requirement of each benchmark in the workload. This is obtained by summing the stack frame sizes of a sequence of functions starting from *main* that are certainly called independent of the input data set. This guarantees that the benchmark would have surely been allocated at least that much space, regardless of which input data sets are used in testing. Second, we simulate an experiment in which every task is given its above-computed lower-bound stack space, thus ensuring that every page that can overflow will overflow. The overflow pages are grown in an artificial task with unbounded amount of memory. This ensures that the application runs to completion, allowing us to measure the run-time of the application in the presence of overflows. Third, we modify our algorithm so that it runs through its *worst case* at each page overflow encountered. To implement this, all the tasks are checked for the presence of free space (even if a task with free space has already been discovered) before discontiguously growing the stack. Further, for each task, *all* its pages are checked for the presence of holes. In this way, this artificial simulation truly yields the theoretically worst-case number of overflows, each incurring the theoretically worst possible overhead upon overflow.

As is, the WCET overhead of MTSS is low enough to warrant the use of our scheme in preemptive real time systems. In particular it is much lower than the worst-case run-time of hardware virtual memory which our scheme seeks to replace, which has very poor real-time guarantees because of the possibility of TLB misses. Indeed, the use of virtual memory in safety-critical real time systems has been avoided precisely because of this reason [Bennett and Audsley 2001].

However, if the real-time bound for a particular application is found to be too high with MTSS in a hard-real-time system, MTSS should not be used. Soft real-time systems can use MTSS without problems since the average case overhead is much lower.

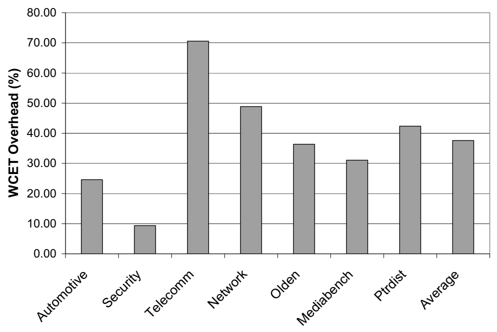

Fig. 12. Worst case run-time overhead for different workloads

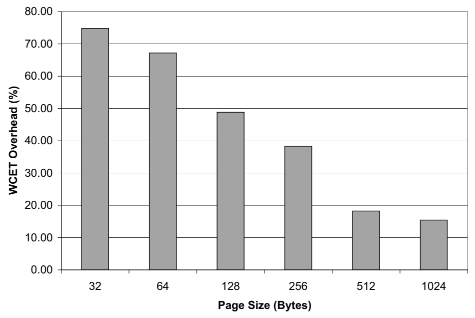

Fig. 13. Variation of page size on real-time guarantees

If the real-time bound with our default page size of 128 bytes is found to be too high in hard real-time system, a higher page size can be used to reduce the real-time bound. Figure 13 shows the variation of page size on real time guarantees for the *network* multi-tasking workload. As the figure shows, an increase in page size reduces the worst case run-time overhead and offers better real-time guarantees. This is because a large page size reduces the chances of page overflow, and therefore, does not incur the overhead of page allocation frequently. However, large page sizes recover less space as shown in figure 10. This is a tradeoff and an appropriate page size should be chosen based on the workload(s) that will be frequently executed by the embedded system.

Additional Statistics We also measure the frequency of holes on our scheme. Among the multi-tasking workloads we used, only a few holes are generated in the overflow space. On an average the number of dynamic page allocations that lead to the generation of a hole when that page is freed is less than 5% of the total pages allocated to a particular task after it overflows.

Some of the workloads, for example, the *telecomm* multi-tasking workload, did not generate any holes in the overflow space. To understand why, consider that holes are generated only when multiple tasks overflow in the same task. The *telecomm* workload always had multiple tasks overflowing in different overflow spaces, never generating holes. These results indicate that a *hole compaction* scheme will not yield significant benefits for our scheme.

An experiment is also performed to calculate the average number of pages in multiple-page allocations. This number depends on the frame sizes of the overflowing procedures and the page size used. With 128-byte pages, we observed that the maximum number of pages allocated at one time for a single overflowing stack frame is just four in the *network* multi-tasking workload; the median is 1, and the average across all stack frames is 1.25.

11. CONCLUSION

This work presents a method for reusing stack across tasks in multi-tasking embedded systems without hardware virtual memory support. The main objective of the method is to improve the reliability of such systems in the presence of out-of-memory errors. This is achieved by sharing stack across multiple tasks, in case of stack overflow, through the use of an innovative *paging system*. Results indicate that the overheads of our scheme in the common case of no overflow are low: the run-time and energy use overheads are 3.1% and 3.2%, respectively, on average. Our scheme is able to recover 54% space on an average for the overflowing task in the multi-tasking workload. Alternately, when MTSS is used to reduce the amount of physical memory in the system instead of increasing reliability, it is able to reduce the stack space required by 16% on average for our workloads. Our scheme provides good real time guarantees, and therefore, can be used for real-time systems.

The future work would explore if MTSS can profit from having a few fixed page sizes instead of a single size at a time. The future work would also quantify the effect of different types of task scheduling on MTSS. Finally future work will look at how systems with virtual memory can take advantage of MTSS, in situations where multiple threads are running in the same virtual address space.

REFERENCES

APPEL, A. W. 1987. Garbage collection can be faster than stack allocation. *Information Processing Letters 25,* 4, 275–279.

APPEL, A. W. AND GINSBURG, M. 1998. *Modern Compiler Implementation in C.* Cambridge University Press.

ARM 2003. *ARM7TDMI Technical Reference Manual*, Fourth ed. Document No. ARM DDI0210B.

AVI SILBERSCHATZ, PETER BAER GALVIN, G. G. *Operating Systems Concepts, Seventh Edition*, Seventh ed. John Wiley & Sons Inc.

BAKER, T. 1990. A stack-based resource allocation policy for realtime processes. In *Proceedings of the Real-Time Systems Symposium.* 191–200.

BEHREN, R. V., CONDIT, J., ZHOU, F., NECULA, G. C., AND BREWER, E. 2003. Capriccio: Scalable threads for internet services. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles.* ACM Press, 268–281.

Bennett, M. and Audsley, N. 2001. Predictable and efficient virtual addressing for safety-critical real-time systems. In *Proceedings of the 13th Euromicro Conference on Real-Time Systems, Delft, The Netherlands*. IEEE Computer Society, 183 – 190.

Biswas, S., Simpson, M., Carley, T., Middha, B., and Barua, R. 2006. Memory Overflow Protection for Embedded Systems using Run-time Checks, Reuse and Compression. *ACM Transactions in Embedded Computing Systems* To Appear.

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.

Brylow, D., Damgaard, N., and Palsberg, J. 2001. Static checking of interrupt-driven software. In *Proceedings of the 23rd international conference on software engineering*. 47–56.

Carbone, J. 2004. Efficient memory protection for embedded systems. *RTC Magazine*.

Dionne, D. J. 1998. uClinux – Embedded Linux Microcontroller Project.

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

GCC. The GCC Compiler. *http://gcc.gnu.org/*.

GDB. GDB: The GNU Project Debugger. *http://www.gnu.org/software/gdb/gdb.html*.

Grunwald, D. and Neves, R. 1996. Whole-program optimization for time and space efficient threads. In *Proceedings of the Seventh Intl. Conference on Architectural Support for Programming Languages and Operating Systems*. ACM Press, 50–59.

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.

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

Hertz, M. and Berger, E. D. 2005. Quantifying the performance of garbage collection vs. explicit memory management. *SIGPLAN Not. 40,* 10, 313–326.

Hogen, G. and Loogen, R. 1993. A new stack technique for the management of run-time structures in distributed implementations. Tech. rep., RWTH Aachen, Germany. http://citeseer.ist.psu.edu/hogen93new.html.

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

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

Lamie, B. 2000. A multitasking revolution.

Middha, B. 2006. MTSS: Multi Task Stack Sharing for Embedded Systems. M.S. thesis, University of Maryland, College Park, MD.

Montanaro, J. et al. 1996. A 160MHz, 32b, 0.5W CMOS RISC microprocessor. *IEEE Journal of Solid State Circuit 31,* 11, 1703–1714.

Moore, R. 2001. Unbound stacks and stoppable tasks. http://www.programmersheaven.com/articles/smx/article3.htm.

Panda, P. R., Catthoor, F., Dutt, N. D., Danckaert, K., Brockmeyer, E., Kulkarni, C., Vandercappelle, A., and Kjeldsberg, P. G. 2001. Data and memory optimization techniques for embedded systems. *ACM Transactions on Design Automation Electronic Systems 6,* 2, 149–206.

Pizka, M. 1999. Thread segment stacks. In *In Proceedings of International Conference on Parallel and Distributed Processing Techniques and Applications*.

Regehr, J., Reid, A., and Webb, K. 2003. Eliminating stack overflow by abstract interpretation. In *Proceedings of the 3rd International Conference on Embedded Software*. Springer-Verlag, 306–322.

Shantanu Sardesai, D. M. and Dasgupta, P. 1998. Distributed cactus stacks: Runtime stack-sharing support for distributed parallel programs. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications.*

Sinha, A. and Chandrakasan, A. P. 2001. In *JouleTrack: a web based tool for software energy profiling.* 220–225.

Tiwari, V., Malik, S., and Wolfe, A. 1994. Power analysis of embedded software: A first step towards software power minimization. In *IEEE Transactions on Very Large Scale Integration (VLSI) Systems.* 437–445.

Wang, Y. and Saksena, M. 1999. Scheduling fixed priority tasks using preemption threshold. In *Proceedings of the Sixth International Conference on Real Time Computer Systems and Applications.*

Witchel, E., Cates, J., and Asanović;, K. 2002. Mondrian memory protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems.* ACM Press, 304–316.

Wong, K.-F. and Dageville, B. 1994. Supporting thousands of threads using a hybrid stack sharing scheme. In *Proceedings of the ACM Symposium on Applied Computing.* ACM Press, 493–498.