MTSS: Multi Task Stack Sharing for Embedded Systems

Bhuvan Middha Matthew Simpson Rajeev Barua

Department of Electrical & Computer Engineering
University of Maryland
College Park, MD 20742, USA

{bhuvan, simpsom, barua}@eng.umd.edu

ABSTRACT

Out-of-memory errors are a serious source of unreliability in most embedded systems [22]. 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. Since there is no swap space for the application to grow into, the system crashes if its memory footprint is exceeded by even one byte.

This work improves system reliability for multi-tasking embedded systems by proposing MTSS, a multi-task stack sharing technique, that grows the stack of a particular task into other tasks in the system after it has overflown its bounds. This technique can avoid the out-of-memory error if the extra space recovered is enough to complete execution. Experiments show that MTSS, on an average, is able to recover 47% of the stack space allocated to the overflowing task in the free space of other tasks. Therefore, even if we underestimate the stack size of a particular task by 47% on an average, it will still run to completion by reusing stack in other task's stack.

Alternatively, MTSS can also be used for decreasing the physical memory for an embedded system by reducing the initial memory allocated to each of the tasks and recovering the deficit by sharing stack with other tasks. Results show that MTSS used in this way can be used to reduce the memory required in multi-tasking embedded systems by 18% on an average, thus reducing the memory cost of the system. MTSS also offers good real time guarantees, since it uses a paging system that never incurs an episodic increase in run-time.

The overheads of MTSS are extremely low: the run-time and code size overheads are 1.8% and 2.6% on an average, making it a feasible method for increasing system reliability and reducing the memory footprint of embedded systems.

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

Keywords

Out-of-memory, runtime checks, reuse, stack overflow, heap overflow, reliability, cactus stack, meshed stack

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. This is because hardware-assisted virtual memory [15] detects physical memory overflow and provides swap space on disk upon overflow.

Unfortunately a great majority of embedded chips (we estimate over 95%) have no virtual memory [19]. Examples of such 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; there are many more. It is easy to see why: virtual memory hardware exacts a significant penalty in energy usage, area cost, and design complexity. Typically it checks that the address of *every* memory access is within segment bounds and it may also translate the address using a TLB. The energy cost of these frequent tasks can be prohibitive [23]. Even a simpler virtual memory proposed by a few systems, which provides segment bound checking but not swap space, is not widely used because of its energy cost. Furthermore, the trend of MMU-less embedded processors will not diminish with time [9] since not only virtual memory hardware accounts for a significant portion of a system's total energy consumption but the possibility of Translation Lookaside Buffer (TLB) misses is detrimental to real time guarantees as well.

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 pre-empted before completion) fits in the available physical memory at all times.

Unfortunately, accurately estimating the maximum memory requirement of an application at compile 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 compile time. 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 allows a run-time-dependent size block 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 compile-time is difficult. Indeed in cases (i), (iv) and (v) the maximum stack size is dependent on the input data and is unknowable at compile-time. As an example, a recursive function invoked with a command line argument leads to an unbounded stack at compile time.

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

Lacking an effective way to estimate the size of the stack and heap at compile time, the usual industrial approach is to run the application on different data sets and observe the maximum sizes of stack and heap [7]. Unfortunately, this approach of choosing the size of physical memory never guarantees an upper bound on memory usage for all data sets, 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 and memory overflow 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, and industrial accidents. In our past work [3], we have looked at the problem of overflow detection and reuse of memory within a task in order for the application to continue execution. This work builds upon the past work to reuse stack memory available across different tasks in the embedded system in order to further improve system reliability. This work is important in the light of the fact that the use of multi-tasking is dramatically rising in embedded software development [20, 21] and there is a large amount of memory available for reuse across different tasks in the system.

Our scheme is based on the observation that the most commonly used stack layout for multi-tasking systems, called a *cactus stack* [21, 24, 26], wastes a significant amount of memory. In such a scheme, the stack grows from the initial parent stack and bifurcates into multiple different branches, which vary depending upon the control flow of parallel tasks. All the parallel tasks share the initial common stack but have their own, branched stack after the beginning of the parallel region. The size of each branched stack is equal to the maximum observed stack size that is observed during testing across all input sets.

The stacks for all tasks that can be simultaneously active (running or pre-empted) are non-overlapping in memory. The heap is allocated from a free list shared across tasks. The inefficiency of this organization, in that it can waste space is apparent from the observation that all the tasks are unlikely to need their maximum stack space at the same instant of time. Thus when one task overflows, it is quite likely that the stacks of other tasks have substantial free space in them. Normally this space cannot be used for the overflowing stack since it is not contiguous with it. Our scheme aims to use this free space by growing the overflowing stack discontiguously in that space. If successful, the overflow will be postponed and hopefully avoided, thus increasing system reliability.

This paper proposes *MTSS*, a multi-task stack sharing scheme that is built on top of a cactus-stack layout. It aims to recover wasted space using an innovative *paging system* as follows: First, compile time checks are inserted at the beginning of each procedure which check for stack overflow [3]. Second, if an overflow is detected, then a fixed size block of memory, called a *page*, is allocated in the free space of one of the other tasks that has such free space. The page is allocated in the stack space at the far end from the direction of stack growth 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 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, compile time checks are inserted at each procedure return, to check if the overflowing stack has withdrawn from the page. In that case, that page is released back to the free list of pages. Thus, using this scheme, all the free space is utilizable across all 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 47% 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 compile time check for overflow is executed on every procedure entry and return. Results show that this overhead is less than 1.8% 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 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 the worst-case execution time (WCET) is less than 11% on an average 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 may incur page faults that dramatically degrade the WCET.

In an alternate configuration, our scheme can also 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 18% on average, thus reducing the dollar cost of the system.

The rest of the paper is organized as follows. Section 2 outlines the related work in the area of sharing stack space. Section 3 describes the compile time checks inserted to detect stack overflow. Section 4 describes our overall scheme in detail for reusing stack space across different tasks. Section 5 describes the experimental platform, which we use for our evaluations. Section 6 discusses the results. Section 7 concludes.

2. 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 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) [29] scheme is a hardware approach designed to provide fine-grained memory protection for systems requiring data sharing among processes. Another hardware approach [6] 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) [18]. 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. None of these schemes are related to software-managed TLBs [27] and software address translation [17], which are two techniques used to give the operating system more control over address translation and are, therefore, unrelated to the notion of protection or sharing.

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 [12, 28]. In older schemes, which used heap based allocation of stacks [4, 14], 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. In one of the recent schemes [28] 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. Five differences of our scheme with respect to [28] 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, their scheme does *not* utilize all the physical memory available in the system, while MTSS does. Their scheme uses different sized memory pools for different types of functions, which are not sharable. Fourth, 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 load on segment tables. Fifth, 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.

Two other attempts have been made to recover unused space from other tasks in a multi-tasking system. In the first scheme, run-time information of several parallel tasks is kept on a single stack, leading to a *meshed stack* organization [16]. In this scheme, new activation records are always generated on top of the stack. 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. This scheme suffers from an episodic increase in run-time when the garbage collector is invoked, leading to poor real time guarantees. Our scheme, on the other hand, offers better real time guarantees since the discontiguous stack growth overhead is non-episodic. This is because every time the stack overflows, one fixed size page is allocated from a list of free pages, which incurs the same cost throughout the execution of program. Also, 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 from memory.

In the other attempt for reusing memory across tasks, each thread shares stacks from a stack pool [21, 30]. In [30], 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. But, in the common case, though, 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 back when the thread becomes active. This leads to increased run-time overhead. Our scheme is applicable independent of the number of threads. In addition, the hybrid stack sharing scheme has two more drawbacks. First, even in this scheme, the task's stack can overflow, even when space is available in other stacks in the stack pool since no mechanism for sharing across stacks in the stack pool is implemented. Second, this scheme offers poor real time guarantees since every time, the number of active threads increase, the run-time of the system increases by potentially a very large value due to increased cost of the context switch.

In [2] a stack sharing scheme for real-time tasks is proposed. This scheme is complementary to our scheme. Here, each task is assumed to have a fixed priority, and all tasks share a single stack. When a task T1 is preempted by a higher priority task T2, T1 continues to hold its stack space and T2 is allocated space immediately above T1. The only special requirement is that T1 cannot resume until all tasks occupying space above it have completed. This will always be the case since T1 will be preempted by higher priority tasks only. Thus in a real time pre-emptive system in which each task has a fixed priority, this scheme precludes the need for our scheme. However, for other systems, their scheme can be combined with our scheme in which each set of fixed priority intra-pre-emptible tasks are given a single stack, and memory reuse across such task sets is accomplished using our scheme. Our scheme is also applicable to non-real-time systems with pre-emptible tasks.

Methods for estimating the maximum depth of the stack [5, 25] 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 [13] 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 [3], which also uses run-time checks to detect stack overflow and recovers space from within the overflowing task. More specifically, in [3] an overflowing stack is grown in dead global variables and space freed by compressing live variables. Two differences of our scheme with respect to [3] are as follows: First, our scheme is applicable and optimized for multi-tasking systems; their scheme is optimized for single tasking systems. Second, our scheme recovers space by sharing stack across different tasks in the system, while they recover space from within a task. However, the work in [3] 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.

3. RUN-TIME CHECKS TO DETECT STACK OVERFLOW

Our scheme builds upon the software scheme for detecting stack overflow in our previous work [3]. A brief overview follows. To see how overflow can be detected, consider that the stack grows only at procedure calls. Figure 1 shows the check, which we insert at the beginning of every procedure. Without loss of generality, we assume that the stack grows from higher-numbered addresses to lower. Now, to understand figure 1, consider that 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 1 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 is the heap pointer in case the heap adjoins 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 4.

The overheads of the added stack checks in the baseline scheme can be reduced by the *rolling checks optimization* [3]. 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, it might be, in certain cases, enough to check once at the start of the parent that there is enough space for the stack frames of both parent and child procedures together. In this way, the check for the child is 'rolled' into the check for the parent, eliminating the overhead for the child. The rolling checks optimization reduces the run-time overhead since if a child is called more frequently than the parent then the reduction in overhead can be more than half. We have not yet implemented the rolling checks optimization in our scheme, but it is likely that the run-time overheads will be reduced even below 1.8%, once the optimization is in place. Results in [3] show that this optimization reduces the overhead by about half.

The safety checks are easily extended to handle any calls to the alloca() library function. The alloca() function, provided in some dialects of C, allocates additional space on the current procedures stack frame. The amount of additional space is an argument to alloca(), and need not be known at compile-time. If a procedure calls alloca(), its basic safety run-time check in line 1 of figure 1 is modified to check that

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

Figure 1: Code inserted at procedure entry for detecting stack overflow.

Stack-Ptr - Size-argumentof-alloca $< STACK\_BOUNDARY$. If the alloca function is called from within a loop or constructs it argument from a combination of local variables of the function, then the size of its argument is unknown both at compile time and at runtime during the beginning of function. This case is not handled by our scheme. In future work we will investigate mechanisms to handle this case.

4. MULTI-TASK STACK SHARING

This section presents our scheme of reusing stack across different tasks. When a stack overflow is detected by the run-time checks above, the scheme allows the overflowing stack to grow in the free space available in the stacks of other tasks. It 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 2(a) shows the normal behavior of the system in which none of the three tasks T1-T3 in the system are out of memory. Figure 2(b) shows the snapshot of the system when T1's stack has overflowed its bounds into space in other tasks. Figure 2(c) shows a magnified view of the overflow space in figure 2(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 (swapped 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, a set 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 is the difference between its stack pointer and overflow pointer. As an example, the overflow pointer of task T2 can be seen in Figure 2(b). Third, a set of *overflow_started* global boolean variables is also maintained. This variable is set to true if the task overflows its native stack bound, and it is unset when the stack recedes back to its native space. Below, we discuss our basic scheme for sharing stack among multiple tasks.

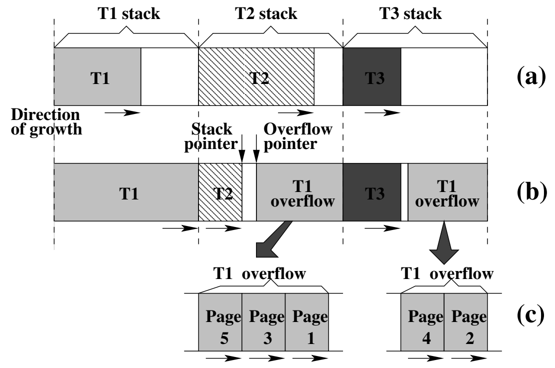

Figure 2: Example showing reuse across tasks (a) Normal operation of Cactus Stack (b) Overflow handling in MTSS; and (c) Magnified view of overflow space

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

Figure 3: Code inserted at procedure entry for detecting stack overflow with MTSS.

Stack overflow is detected using the run-time checks inserted by the compiler at the beginning of every procedure, as shown in figure 1. Here, the STACK_BOUNDARY 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. The modified check is shown in figure 3.

Once an overflow is detected, our scheme allocates a fixed block of memory called a *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 of overflow in this task and works well in principle as we show in the results section.

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 of checking the *overflow_started* variable is added in the check for detecting stack overflow in figure 3 since page overflows need to be detected for overflowing stacks.

Once the out-of-stack condition is detected by the run-time checks, growing the stack discontinuously in other task's stack pages is done by changing the original stack pointer to *overflow pointer* + *page size*. Further calls occur as usual - procedure returns are handled by *copying* the old frame

```
1. if (Overflow-Started[current-task-id]) {
     /* Stack Overflow */
2.   if (Stack-Ptr > Overflow-Pointer[overflow-task-id])
3.     Overflow-Pointer[overflow-task-id] =
           Overflow-Pointer[overflow-task-id] - pagesize
4. }
```

Figure 4: Code inserted at procedure exit for receding the overflow pointer.

pointer, which stores the stack pointer value for the (non-overflowing) parent procedure, in to the new stack page. The old frame pointer in the current stack frame is correct because it is saved *before* the discontinuous stack pointer assignment upon overflow. The same mechanism is used to return to the previous overflow pages, for overflowing stacks spanning multiple discontinuous pages. Moreover, overflowing procedure arguments are copied over to the new stack page to ensure correct semantics.

Receding the overflow pointer The overflow pointers maintained per task represent the upper limit of overflow space of each task. In order to reduce the possibility of native stack overflow due to the presence of overflowing stacks of other tasks and to maximize the amount of memory available for reuse, overflow pointers must be receded as soon as the overflowing stack recedes from the page. To understand this more clearly, let us first consider how overflow pointers are implemented in our scheme. First, each overflow pointer is assigned to the base of a particular task's stack. Second, overflow pointers are always grown in the direction opposite to that of the growth of the stack pointer, that is from lower memory addresses to higher memory addresses. Third, each time a page is allocated for an overflowing stack in a task, the overflow pointer for that task is incremented by the size of the page. This is required, since overflow pointers maintain the upper limit for each task's stack. When one page is allocated for an overflowing stack in a task T, the upper limit for that task is reduced by the page size. In other words, in effect, the available stack size in task T is *reduced* by the size of the page, until the overflowing stack remains in task T.

To recede the overflow pointer, run-time checks are inserted by the compiler at the exit of every procedure. Figure 4 shows the check inserted at the exit of every procedure call. To understand figure 4, consider that the stack shrinks only at procedure return and is incremented by the size of the procedure frame. The code in figure 4 is inserted immediately *after* the stack pointer is incremented. Thus, the code first checks, if the overflow has already started. If not, then there is no overflow pointer to adjust and the code returns. If the overflow has already started, then the check compares the updated stack pointer to the current value of overflow pointer of the task in which the overflowing stack is being grown (this is represented by the *overflow-task-id* in figure 4). If the check succeeds, then the overflowing stack has receded from this page and the overflow pointer is decremented by the size of the page.

Holes in the Overflow Space If multiple stacks overflow their bounds, then the result could yield in *holes* in the overflow space, as depicted in figure 5. To understand

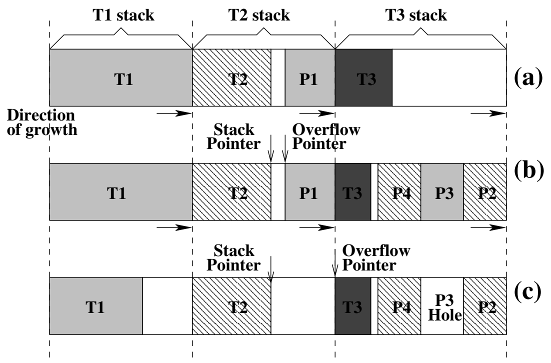

Figure 5: Example showing holes in overflow space (a) T1 overflows in T2 (b) T1 and T2 overflow in T3; and (c) T1 recedes leaving holes in overflow space

figure 5, let us consider that there are three tasks T1-T3 in the system. Let us assume that task T1 overflows its bounds and starts growing in page P1 in task T2 as shown in Figure 5(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 as shown in figure 5(b). Now, if the stack of T1 recedes back to its native space it vacates pages P1 and P3. This is shown in figure 5(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. Instead holes 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 in an array data structure for each task. Subsequently, before allocating a free page, we traverse this list 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.

Multiple-Page Allocations The base scheme to share stack across 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. This is because a procedure frame cannot be allocated discontiguously. If it were, then we would have to modify the stack pointers within the procedure in the case of overflow but not otherwise, 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 pagesize, and taking the ceiling. Second, each task is searched for 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. So, 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 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, since this would adversely impact the real time guarantees.

Choice of page size Next, we discuss why allocating fixed sized blocks of memory is advantageous for our scheme and what is a good page size for our scheme. Allocating fixed size blocks of memory gives us the following advantages over variable-sized allocation: First, variable-sized allocation leads to *external fragmentation* (holes in the memory). This leads to increased run-time for allocation on overflow as compared to a fixed size allocation since allocating memory requires a scan through all the holes in order to determine a fit; further a mechanism to merge holes is usually also needed to limit the number of small, useless holes. Second, if the variable-sized allocation scheme allocates exactly the amount of stack space required by the overflowing procedure and no more, then the number of page overflows may increase. If the overflowing procedure in turn calls another procedure, then it will result in another page overflow. Allocating additional memory than required might lead to wastage and make the implementation more complex. Third, with variable-sized allocation compaction will be required at frequent intervals, as otherwise space recovery will suffer. This will degrade the real time guarantees of the reuse scheme.

With paging, page size is also an important consideration. Both small and large page sizes have their own advantages and disadvantages, as in hardware virtual memory, but with different tradeoffs. Let us consider the advantages of small page sizes. Fixed size allocation leads to *internal fragmentation*. Smaller page sizes reduce internal fragmentation as compared to larger page sizes and therefore, are capable of recovering more space as compared to larger page size. On the other hand, small page sizes worsen the real time guarantees of the system. This is because the probability of a page overflow increases as the page size reduces. Furthermore, smaller page sizes lead to increased run-time overhead in the presence of stack overflows since the probability of page overflow increases. 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 space available across all the tasks in the system. Since in a multi-tasking system the heap is shared by all the tasks, we can inherit the scheme proposed in our previous work [3] that allows overflowing stack to be discontiguously grown in the heap. Since the method to reuse the heap is inherited from previous work, to be fair, in our experiments we do not count the space recovered from the heap towards the benefit from our method.

Alloca function calls The alloca() library function calls are handled by adding the size of alloca's function argument to the calling procedure's frame size before allocating pages for it. All the other steps of the algorithm are applied to this modified frame size.

Dynamic Tasks MTSS can be extended to handle 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 for a start, at the cost of more frequent future overflows.

5. EXPERIMENTAL SETUP

This section presents the experimental platform which is used for evaluating our scheme. We have implemented our scheme inside the ARM GCC v3.4.3 cross compiler [10] targeting the ARM7TDMI [1] 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 [8] for implementing the proposed techniques. μC`linux` is a derivative of Linux 2.0 kernel intended for microcontrollers without Memory Management Units. We use the *SCHED_OTHER* scheduling policy for scheduling the different tasks in the system. This policy 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. We also modify the operating system to provide a new system call that returns the value of stack pointer of an inactive (swapped out) task. This is implemented by saving the value of 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 embedded processor available as part of the GDB v6.3 distribution [11] for running the operating system as well as the multi-tasking applications. 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 OS. Thus, the overall framework consists of multi-tasking applications running on top of μC`linux` 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 in a real embedded system.

6. RESULTS

This section presents the results for the proposed scheme for reusing stack across multiple tasks in an embedded system. The multi-tasking workloads that are used for evaluation are constructed by combining together multiple benchmarks from one domain of the MIBench embedded benchmark suite [13]. Each domain in the MIBench embedded benchmark (such as automotive) 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. Table 1 shows the names and characteristics of the resulting workloads that we use for our evaluation. Combined together, we evaluate four workloads each of four benchmarks, for a total of 16 benchmarks. Unless otherwise

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

Table 1: Multi-tasking benchmark programs and characteristics

stated, all the results are generated for a fixed page size of 128 bytes.

Stack Allocation The initial stack memory allocated to each task as shown in column 5 in table 1 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. 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.

An alternative implementation of the scheme consists of giving 0 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 runtime and energy overhead as the number of page overflows will increase. The current scheme on the other hand incurs very low overhead in the common case of no overflow. 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 control flow of each task, leading to generation of additional holes. This will reduce memory utilization. To offset this, compaction of holes might be necessary, but that will spoil the real time guarantees.

Overheads of run-time checks Table 2 shows the overheads due to run-time checks inserted at the beginning and end of every procedure. These overheads are in the common case when no task in the system overflows. The results indicate an average code size overhead of 2.6% and an average run-time overhead of 1.8%. These results show that the multi-task stack sharing scheme is possible with very low overheads. Moreover, quantitative results cannot evaluate the benefit of increased reliability of the embedded system due to multi-task stack sharing.

The automotive workload has somewhat higher run-time overhead due to the presence of one benchmark *bitcnt* with small-sized recursive functions. Recursive functions lead to the execution of run-time checks for every invocation with few intervening instructions, increasing the run-time overhead.

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

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

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 total stack allocated to the task. Figure 6 shows the maximum satisfiable overflow for each task in different workloads. In figure 6 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 *average* workload represents the average of all tasks across all workloads. The figure shows that on an average we can recover 47% of stack space per task by reusing stack across tasks. In other words, even if we underestimate the size of a task stack by 47% on an average, the workload will still run to completion. 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 overflowing task. However, the memory allocated to each task is often limited in embedded systems due to cost constraints.

For some tasks in figure 6, such as task 1 (*blowfish*) in the *security* workload, the space recovered is 0%. To understand this, consider that *blowfish* has a total stack requirement of 5632 Bytes, and it contains a procedure of size 4608 Bytes, as its main procedure. Procedure frames needs to be allocated contiguously on a stack. Thus, if 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.

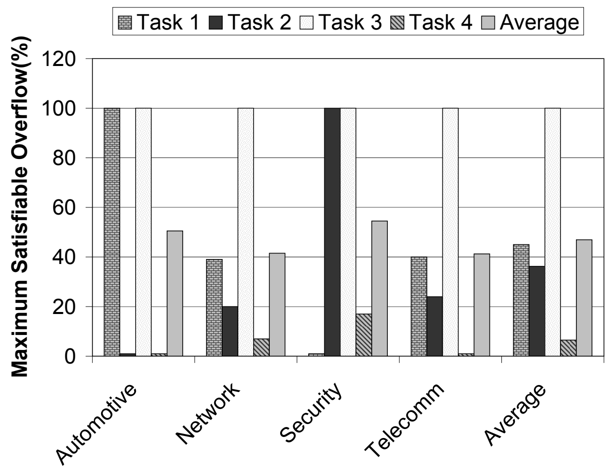

Figure 6: Maximum Satisfiable Overflow for different tasks in different workloads

The numbers in figure 6 are collected as follows. The workload is first executed with the stack size for each task equal to its observed requirement for a particular input data set. 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.

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, which yields the largest stack size seen during testing. Surely this will cause 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 7 plots the PRS numbers for different workloads. The difference in the MSO and PRS numbers is that MSO numbers are calculated at per task level, while PRS numbers are calculated at the workload level. The figure shows that on an average we can recover 18% of stack space across all the multi-tasking workloads, reducing the cost of the memory needed. The run-time at the PRS configuration will be higher than that for MSO because of the more frequent overflows, but is still upper-bounded by the worst-case real-time bounds measured later in this section.

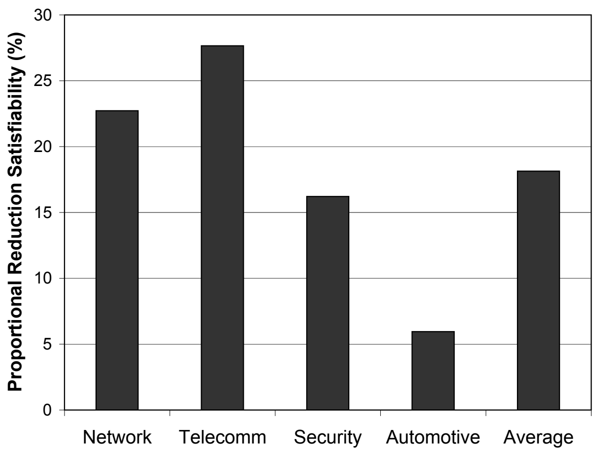

Figure 7: Proportional Reduction Satisfiability for different workloads

Effect of Page Size Figure 8 shows the effect of the page size on the MSO for the *network* multi-tasking workload. The figure shows that as the page size increases, the MSO of a task decreases in general. This follows from the discussion in section 4, in that larger page sizes leads to larger *internal fragmentation*. Therefore, a workload may be declared out-of-memory if a task requires a few bytes of overflow space and no other task in the system contains free space equal to the size of the page. This happens since our scheme allocates stack space in the granularity of pages.

The *treeadd* benchmark is an anomaly in that smaller page sizes of 32 and 64 bytes lead to lesser space recovered. To understand why, consider that *Treeadd* contains recursive functions with small procedure frame sizes. With small page sizes, the space remaining in the page is substantial but less than that required for the next frame, and so is wasted. For example, consider a recursive procedure *P* with a frame size

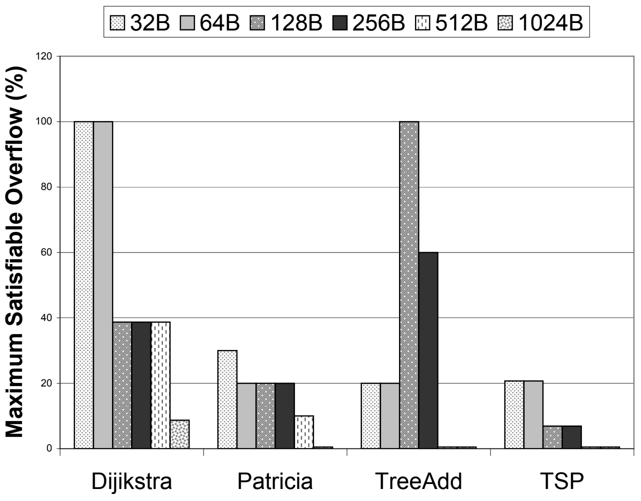

Figure 8: Effect of page size on MSO for network workload

of 40 bytes that causes a task's stack to overflow its bounds. With a page size of 64 bytes one page is allocated, wasting 24 bytes in internal fragmentation. This process is repeated for every invocation of P leading to large scale memory wastage and premature declaration of the out-of-memory condition. A page size of 128 bytes, on the other hand, wastes eight bytes for every three invocations of P (128 - 3*40 = 8). This is a factor of nine lower in wastage compared to a page size of 64 bytes, leading to better utilization of memory.

Real time bounds Figure 6 shows the variation of real time guarantees for different workloads expressed as a percentage run-time overhead. The real-time bound increase for a workload is an average of the increase for a particular task. These numbers are calculated by simulating an environment in which every page overflows, thereby incurring the overhead of page allocation every time. These numbers represent an upper bound on the run-time overhead of our scheme; the actual run-time increase is usually much lower (it averages 1.8% in the common case of no overflow). As shown in figure 6, on an average the run-time overhead in the worst case is about 11%. This overhead is low enough to warrant the use of our scheme in pre-emptive 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 page faults. 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.

If the real-time bound with our default page size of 128 bytes is found to be too high, a higher page size can be used to reduce the real-time bound. Figure 6 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 therefore, offers better real-time guarantees. To understand this, consider that 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 8.

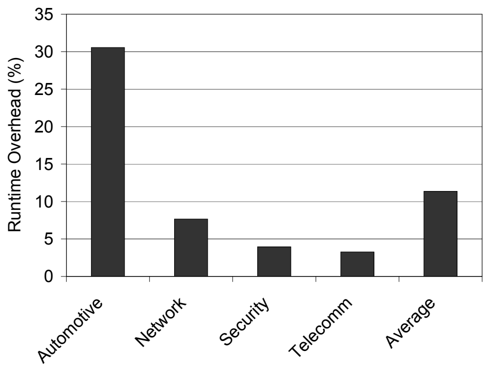

Figure 9: Worst case run-time overhead for different workloads

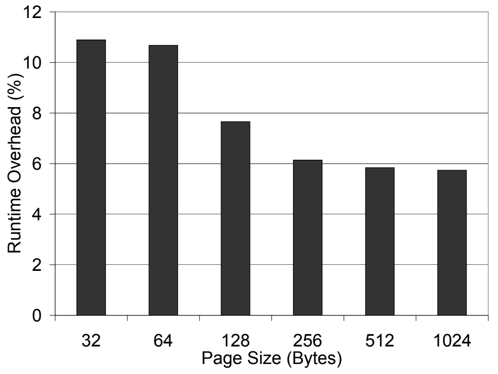

Figure 10: Variation of page size on real time guarantees

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 pages, which lead to generation of holes is less than 5% of the total stack pages allocated to a particular task.

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 the maximum number of pages allocated is just four in the *network* multi-tasking workload, with the median being 1, and the average close to 1.25.

7. CONCLUSION

This work presents a method for reusing stack across tasks in a multi-tasking embedded system whose main goal is to improve the reliability of such systems in case 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 extremely low: the run time, code size and energy consumption are 1.8%, 2.6% and 1.2% on average. Our scheme is able to recover 47% 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 18% on average for our workloads. Our scheme provides good real time guarantees, and therefore can be used for real-time systems.

In future work, we wish to explore the effect of MTSS on presence of pages of multiple sizes, rather than one fixed size. We also wish to explore the effect of different types of task scheduling on MTSS.

8. REFERENCES

[1] *ARM7TDMI Technical Reference Manual*, fourth edition, May 2003. Document No. ARM DDI0210B.

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

[3] S. Biswas, M. Simpson, and R. Barua. Memory overflow protection for embedded systems using run-time checks, reuse and compression. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 280–291. ACM Press, 2004.

[4] D. Bobrow and B. Wegbreit. A model and stack implementation of multiple environments. In *Comm. of the ACM*, pages 591–603, Oct 1973.

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

[6] J. Carbone. Efficient memory protection for embedded systems, 2004. www.rtcmagazine.com/home/article.php?id=100120.

[7] N. D. D. Brylow and J. Palsberg. Stack-size estimation for interrupt-driven microcontrollers. Technical report, Purdue University, June 2000.

[8] D. J. Dionne. uClinux – Embedded Linux Microcontroller Project. *http://www.uclinux.org/*.

[9] M. Durrant. Running linux on low cost, low power mmu-less processors, August 2000. www.linuxdevices.com/articles/AT6245686197.html.

[10] The GCC Compiler. *http://gcc.gnu.org/*.

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

[12] D. Grunwald and R. Neves. 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*, pages 50–59. ACM Press, 1996.

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

[14] E. Hauck and B. Dent. Burroughs b6500/b7500 stack mechanism. In *Proceedings of AFIPS, SJCC, vol 32*, pages 245–251, 1968.

[15] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2003.

[16] G. Hogen and R. Loogen. A new stack technique for the management of runtime structures in distributed implementations. Technical report, RWTH Aachen, Germany, 1993. citeseer.ist.psu.edu/hogen93new.html.

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

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

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

[20] B. Lamie. A multitasking revolution, 2000. www.netsilicon.com/pdf/article_expresslogic2.pdf.

[21] R. Moore. Unbound stacks and stoppable tasks, 2001. www.programmersheaven.com/articles/smx/article3.htm.

[22] G. V. Neville-Neil. Programming without a net. *ACM Queue: Tomorrow's Computing Today*, 1(2):16–23, April 2003.

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

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

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

[26] D. M. Shantanu Sardesai and P. Dasgupta. 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*, July 1998.

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

[28] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: Scalable threads for internet services. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 268–281. ACM Press, 2003.

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

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