# ABSTRACT


Title of thesis: MTSS: MULTI TASK STACK SHARING FOR EMBEDDED SYSTEMS

Bhuvan Middha, Master of Science, 2006

Thesis directed by: Assistant Professor Rajeev Barua
Department of Electrical and Computer Engineering

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, that grows the stack of a particular task into other tasks in the system if the task attempts to overflow 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, is able to recover an average of 54% 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 54% on an average, it will still run to completion by reusing space in the stacks 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 to complete execution.

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. Results show that MTSS, used in this way can reduce the memory required in multi-tasking embedded systems by 16% 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 a large episodic increase in run-time.

The overheads of MTSS are low: the run-time and energy overheads are 3.9% and 3.8% 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. In this way, MTSS is a feasible method for increasing system reliability and reducing the memory footprint of embedded systems.

# MTSS: Multi Task Stack Sharing For Embedded Systems

by

Bhuvan Middha

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Master of Science
2006

Advisory Committee:
Professor Rajeev Barua, Chair/Advisor
Professor Shuvra Bhattacharyya
Professor Peter Petrov

# ACKNOWLEDGMENTS

First of all, I want to thank my family for lending their support throughout my stay at the University of Maryland.

I am very grateful to my advisor, Prof. Rajeev Barua. It was a great experience to work with him and to bounce of ideas with him. Many a times, he egged me on when things were not looking bright. And I also thank him for his understanding and help in different situations.

Special thanks to Prof. Peter Petrov and Prof. Shuvra Bhattacharyya for agreeing to serve on my dissertation committee and giving their valuable feedback.

I also want to thank my research group Tom, Sumesh, Alex, Angelo, Matt and Nghi for their help at various points. A special round of thanks to Matt and Nghi for taking part in brainstorming sessions with me. They have been great.

I also want to thank my roommates – Avinash, Anshul, Piyush and Ashish for bearing with me for 2 years, tolerating my idiosyncracies, mood swings and living with my odd working hours. They have been of of great help all throughout. Many thanks.

Lastly, I want to thank my friend circle here at Maryland for making this journey memorable. I would like to thank Amit, Ashish, Chandru, Sebastian, Smitha, Priyanka and Alokika for their help, support and words of motivation. I am especially grateful to both Ashish and Amit both of whom have been simply great and a constant source of inspiration.

TABLE OF CONTENTS


1 Introduction 1

2 MTSS: Overview 5

3 Run-time checks to detect stack overflow 8

4 Profile Independent Rolling Checks Optimization 9
- 4.1 Certainty Optimization . . . 10
- 4.2 Zero-Size Optimization . . . 11
- 4.3 Limited-Size Optimization . . . 11
- 4.4 Rolling-Checks Optimization Ordering . . . 12
- 4.5 Pseudocode of the rolling-checks optimization . . . 12

5 Multi-Task Stack Sharing 18
- 5.1 Receding the Overflow Pointer . . . 21
- 5.2 Holes in the Overflow Space . . . 22
- 5.3 Multiple-Page Allocations . . . 23
- 5.4 Choice of Page Size . . . 23
- 5.5 Re-using Heap for Stack . . . 24
- 5.6 Alloca Function Calls . . . 25
- 5.7 Alternate Choice of Page Overflow Heuristic . . . 25
- 5.8 Profile Independence . . . 26
- 5.9 Alternative with No Initial Stack . . . 26

6 Real World Considerations 27
- 6.1 Dynamic Tasks and Multithreading . . . 27
- 6.2 Communicating Tasks . . . 27
- 6.3 Simultaneous Access and Synchronization . . . 28
- 6.4 Handling Interrupts . . . 29

7 Applicable Systems 31
- 7.1 Background . . . 31
- 7.2 Non-applicable systems . . . 32
- 7.3 Applicable systems . . . 32

8 Related Work 34

9 Experimental Setup 39

10 Results 41
- 10.1 Overheads of run-time checks . . . 43
- 10.2 Maximum Satisfiable Overflow (MSO) . . . 48
- 10.3 Effect of Page Size . . . 49
- 10.4 Proportional Reduction Satisfiability (PRS) . . . 50
- 10.5 Comparison with non-contiguous stack allocation . . . 51
- 10.6 Real time bounds . . . 53
- 10.7 Additional Statistics . . . 56

11 Conclusion 57

Bibliography 58

# Chapter 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 [18] 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 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 [22]. 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 exacts a significant penalty in energy usage, 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 [27]. Indeed, it was shown in a study [25] that virtual memory alone contributed 17% to an embedded system's total energy consumption, which is equivalent to a $(17/(100-17)*100 = 20.5\%$ 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. Consequent to these disadvantages, others too have defended the lack of virtual memory in embedded chips [12].

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 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 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 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 can lead to an unbounded stack at compile time.

Estimating the heap size at compile time is also difficult. The heap is typically used for dy-

namic data structures such as linked lists, trees and graphs whose sizes are highly input-dependent and thus, unknowable at compile time.

Lacking precise compile 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 [8]. 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 [6], 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 [23, 26] and there is a large amount of memory available for reuse across different tasks.

Since MTSS builds upon our previous work [6], 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 dissertation is organized as follows. Chapter 2 overviews MTSS. Chapter 3 describes the run-time checks inserted by our compiler to detect stack overflow. Chapter 4 describes optimizations to reduce the overhead of run-time checks. Chapter 5 details our scheme for reusing

stack space across different tasks. Chapter 6 considers the impact of certain real-world issues on our scheme. Chapter 7 specifies the systems to which MTSS applies. Chapter 8 outlines related work. Chapter 9 describes our experimental platform. Chapter 10 discusses the results. Chapter 11 concludes.

# Chapter 2

## MTSS: Overview

Our scheme is based on the observation that the most commonly used stack layout for multi-tasking systems, called a *cactus stack* [26, 28, 30], wastes a significant amount of memory. In its simplest version, a cactus stack allocates a separate stack for each task in the system. The initial stack size allocated to each task is obtained by observing the stack usage of that task across different datasets and picking the maximum. In this way the stack size allocated to each task is customized for that task and is generally not equal to that for other tasks. Figure 2.1 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 [6]. Second, if an overflow is detected, then a fixed size block of memory called a *page* is allocated in the free space of another

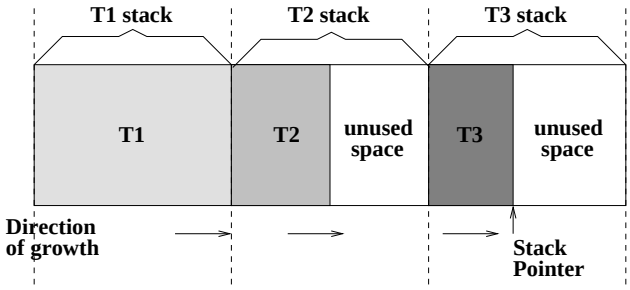


Figure 2.1: Example showing wasted space in a simple version of a cactus stack layout. $T_1$'s stack is full but it cannot use the unused space in the stacks of $T_2$ and $T_3$.

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 3.9% 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.

# Chapter 3

## Run-time checks to detect stack overflow

MTSS builds upon the software scheme for detecting stack overflow in our previous work [6]. 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 3.1 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 3.1 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 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 5.

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 [6].

1.**if** (Stack-Ptr < STACK_BOUNDARY)

2. call routine to handle stack-overflow condition

3.}

Figure 3.1: Code inserted at procedure entry for detecting stack overflow.

# Chapter 4

## 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 [6, 5]. 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 [6] 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 work, 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. The three components of our rolling-checks optimization are listed below.

## 4.1 Certainty Optimization

The first of our rolling-checks optimizations 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 [1]. 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

[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 $y$.

are certain.

## 4.2 Zero-Size Optimization

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.

## 4.3 Limited-Size Optimization

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.

## 4.4 Rolling-Checks Optimization Ordering

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 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).

## 4.5 Pseudocode of the rolling-checks optimization

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 below. Figure 4.1 shows the pseudo code for top level Rolling Checks Optimization. Routine **do_rolling_optimization()** is the highest-level routine for the optimization. It considers all the functions of the application in the order in which they appear in the application binary. In order to roll a check, it first ensures that the check can be legally rolled to all its parents (lines 3-6), before it actually applies the three optimizations (lines 7-9). Routine **can_roll()**, shown later, is a recursive routine that checks if the current procedure (*curr_proc*) can be rolled in to the Ancestor (both arguments to **can_roll()**). It handles the following exceptions that can prevent rolling. First, it checks if the called child is an unresolved virtual function call (line 12-13) in which case the check cannot be rolled. The second exception in the **can_roll()** function (lines 14-17) handles recursive functions. Finally, lines 18-20 in the **can_roll()** procedure check if the parent already had its check rolled, if so the child recursively checks whether it can roll its check to the parent's parents (its grandparents).

Figure 4.2 shows the pseudocode of the certainty optimization. The top level routine **do_certainty_optimization()** checks if each parent P of the *curr_proc* certainly calls *curr_proc* or not (lines 1-2). The routine **certainly_calls()** shown later, checks if a parent procedure certainly calls the child: First, it checks if the parent function has its check rolled, in which case it recursively checks for certainty on parent's parents (lines 5-7). Otherwise, it checks if *curr_proc* post-dominates *Ancestor*, since, post-domination implies certainty (lines 9-11). The notion of post-domination is defined only within a procedure call, so if *Ancestor* does not call *curr_proc* directly, then the routine recursively checks for certainty for each child C of *Ancestor* if their exists a path from the child C to *curr_proc* in the call graph. The routine **certainty_roll_check()** rolls the checks from the current procedure up to its ancestors. The recursive step for **certainty_roll_check()** is in

lines 17-19. The primary termination condition of recursion is when the parent has a check on it (*else* part on line 20); in which case the child's check is rolled to it (lines 21-24). The *Rolled_size* variable for each procedure initially stores the size of the frame for that procedure. When a check is rolled the *Rolled_size* for the child is set to zero and for the parent is set to the sum of the parent and child frame sizes. The algorithm ensures that if a parent has multiple children, then the *Rolled_size* is set to be the maximum needed across all its children (line 23).

Figure 4.3 shows the **do_zero_size_optimization()** routine for the zero-size optimization described above. It removes checks from each function that have a frame size of 0 by setting its corresponding Rolled_Size variable to zero. (lines 1-2)

Figure 4.4 shows the **do_lim_size_optimization ()** routine for limited-size optimization. It rolls checks out of all functions that have a frame size less than a user defined threshold of K bytes (line 1). Lines 2-3 roll the check of the limited size function to each of its parents. The routine **lim_size_roll_check()** first checks if the parent has its check rolled, in which case it recursively rolls the checks to parent's parent (lines 4-6). If the parent already has a run-time check, then lines 8-10 check if *curr_proc* is the first limited size child of this parent. If the check succeeds, then the user defined threshold value of $K$ is added to the Rolled_Size of the parent. We check for the first child because we do not want to add $K$ bytes to the parent's rolled size in case it has multiple children whose frame size is less than K bytes.

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.

```
void do_rolling_optimization () {
1.    for (each procedure Curr_Proc in program binary)
2.        can_roll_to_all_parents ← true
3.        for (each parent P of Curr_Proc)
4.            if (not (can_roll (Curr_Proc, P))
5.                {can_roll_to_all_parents ← false; break;}
6.        if (can_roll_to_all_parents)
7.            do_certainty_optimization (Curr_Proc)
8.            do_zero_size_optimization (Curr_Proc)
9.            do_lim_size_optimization (Curr_Proc, K)
10.   return
11.}

boolean can_roll (Curr_Proc, Ancestor) {
12.   if (call to Curr_Proc is virtual function call)
13.       return (false)
14.   if (either Curr_Proc or Ancestor recursive
          but not both in same cycle)
15.       return (false)
16.   if (Curr_Proc == Ancestor)
          /* Termination for recursive cycles */
17.       return (false)
18.   if (Rolled_Size[Ancestor] == 0)
19.       for (each parent P of Ancestor in the call graph)
20.           if (not (can_roll (Curr_Proc, P))) return (false)
21.   return (true)
22. }
```

Figure 4.1: Pseudo-code for top level Rolling Checks Optimization

```
void do_certainty_optimization (Curr_Proc) {
1.   for (each parent P of Curr_Proc)
2.       if (not(certainly_calls (P, Curr_Proc))) return (false)
     /* Each parent certainly calls the child */
3.   for (each parent P of Curr_Proc)
4.       certainty_roll_check (Curr_Proc, P)
}

boolean certainly_calls (Ancestor, Curr_Proc) {
5.   if (Rolled_Size[Ancestor] == 0) /* Ancestor does not have a check */
6.       for (each parent P of Ancestor)
7.           if (not (certainly_calls (P, Curr_Proc))) return (false)
8.   else
9.       if (Ancestor calls Curr_Proc) /* Ancestor directly calls Curr_Proc */
10.          if (Curr_Proc post-dominates Ancestor) return (true)
11.          else return (false)
12.      else
13.          for (each child C of Ancestor such that
                 there is a path from C to Curr_Proc in call graph)
14.              if (C post-dominates Ancestor)
15.                  if (not (certainly_calls(C, Curr_Proc))) return (false)
16.  return (true)
}

void certainty_roll_check (Curr_Proc, Ancestor) {
17.  if (Rolled_Size[Ancestor] == 0)
18.      for (each parent P of Ancestor in the call graph)
19.          certainty_roll_check (Curr_Proc, P)
20.  else
21.      Longest_Path ← Path in call graph from Ancestor to Curr_Proc,
             not including Curr_Proc, with largest sum of stack frame sizes
             among all such paths
22.      Sum_Stack_Size ← Sum of stack sizes along Longest_Path
23.      Rolled_Size[Ancestor] ← max (Rolled_Size[Ancestor],
             Sum_Stack_Size + Rolled_Size[Curr_Proc])
24.      Rolled_Size[Curr_Proc] ← 0
25.  return
}
```

Figure 4.2: Pseudo-code for certainty optimization

```
void do_zero_size_optimization (Curr_Proc) {
1.    if (frame_size[Curr_Proc] == 0)
2.        Rolled_Size[Curr_Proc] ← 0
}
```

Figure 4.3: Pseudo-code for zero size optimization

```
/* K represents the user defined threshold */
void do_lim_size_optimization (Curr_Proc, K) {
1.    if (Frame_Size[Curr_Proc] > K) return (false)
2.        for (each parent P of Curr_Proc)
3.            lim_size_roll_check (Curr_Proc, P, K)
}
void lim_size_roll_check (Curr_Proc, Ancestor, K) {
4.    if (Rolled_Size[Ancestor] == 0)
5.        for (each parent P of Ancestor in the call graph)
6.            lim_size_roll_check (Curr_Proc, P, K)
7.    else
8.        if (Curr_Proc is the first limited size child of Ancestor)
9.            Rolled_Size[Ancestor] ← Rolled_Size[Ancestor] + K
10.       Rolled_Size[Curr_Proc] ← 0
}
```

Figure 4.4: Pseudo-code for limited size optimization

# Chapter 5

## 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 3, 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 5.1(a) shows the normal behavior of the system in which none of the three tasks T1-T3 are out of memory. Figure 5.1(b) shows the snapshot of the system when T1's stack has overflowed its bounds into space in other tasks. Figure 5.1(c) shows a magnified view of the overflow space in Figure 5.1(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

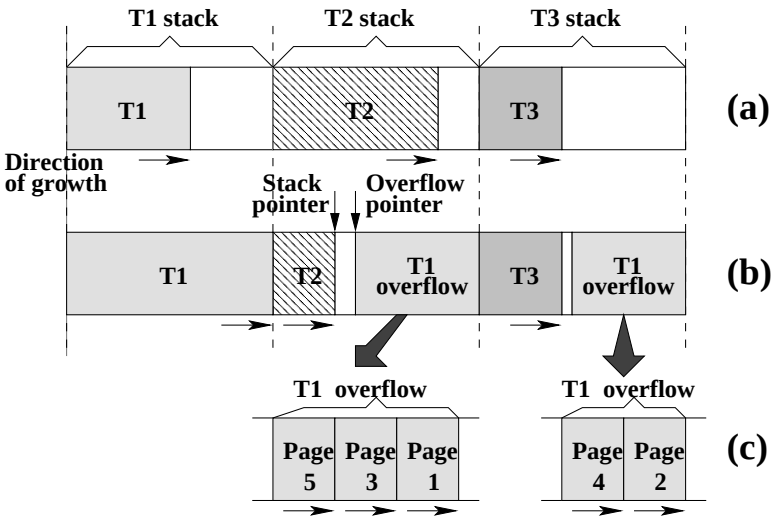


Figure 5.1: Example showing reuse across tasks (a) Normal operation of Cactus Stack (b) Overflow handling in MTSS; and (c) Magnified view of overflow space

overflow pointer of task T2 is shown in Figure 5.1(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 3.1 to that in Figure 5.2. As shown in Figure 5.2 the constant STACK_BOUNDARY in Figure 3.1 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.

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

```
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.}
```

Figure 5.2: 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 > Overflow-Pointer[overflow-task-id]) ||
       (Stack-Ptr < Overflow-Pointer[overflow-task-id] - pagesize)
       /* Stack has receded from the overflow page */
3.    Overflow-Pointer[overflow-task-id] = Overflow-Pointer[overflow-task-id] - pagesize
4.}
```

Figure 5.3: Code inserted at procedure exit for receding the overflow pointer.

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 5.2 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 an extra step because of discontiguous growth. 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. Since this code is executed only in the extremely rare case of overflow, it does not slow down the common case of no overflow.

## 5.1 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 recede the overflow pointer, run-time checks are inserted by the compiler at every procedure's exit. Figure 5.3 shows the check inserted at the return point of every procedure call. To understand Figure 5.3, consider that the stack shrinks only at a procedure return and is incremented by the size of the procedure frame. The code in Figure 5.3 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 code checks if the stack has receded from the overflow page. (The *overflow-task-id* in Figure 5.3 represents the id of the task in which the overflowing stack is being grown) If the check succeeds, then the overflowing stack has receded from this page and the overflow pointer of the task where the overflowing stack was being grown is decremented by the size of the page.

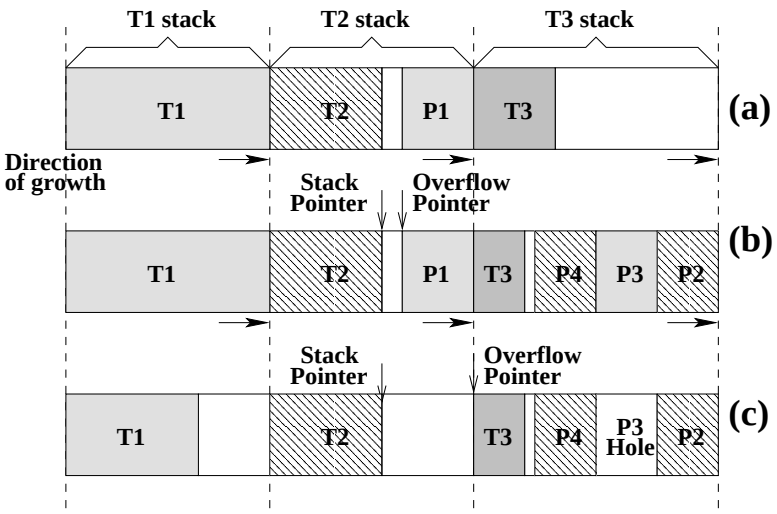


Figure 5.4: Example showing holes in overflow space (a) T1 overflows in T2 (b) T1 and T2 overflow in T3; and (c) T1 recedes leaving holes in overflow space

## 5.2 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 5.4. To understand figure 5.4, 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 5.4(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 5.4(b). Now, if the stack of T1 recedes back to its native space, it vacates pages P1 and P3. This is shown in Figure 5.4(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 5.3 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.

## 5.3 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.

## 5.4 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.

## 5.5 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 [6] 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.

## 5.6 Alloca Function Calls

The *alloca()* library function calls are handled by adding the *alloca()* function's run-time argument, which holds the size of memory in bytes to be allocated on the stack to the calling procedure's frame size to yield the requested frame size for the procedure calling the *alloca()* function. All the other steps of the algorithm are applied to this modified frame size.

## 5.7 Alternate Choice of Page Overflow Heuristic

An alternate heuristic to choose the task for growing the overflowing stack can be to choose the task that has the maximum amount of free space available. This is in contrast to the heuristic proposed in MTSS, which chooses the task with the least number of overflow pages to grow the overflowing stack.

However, the alternate heuristic will likely perform worse than the proposed heuristic because of the following observations. First, note that the stack size allocated to each task is based on its maximum observed stack usage across different data sets, and therefore, each task will necessarily use all its allocated stack space at some instant of time. Hence, the presence of more free space in a task at a particular instant of time does not imply that it will continue to have free space at future instants also. Infact, the chosen task can start using more stack space, the next time it gets the CPU, which will increase its own chances of overflow. Second, this heuristic will recover different amount of stack space depending on the instant at which the task overflow occurs. This is because the free space available in any task depends on its stack usage, which will vary as the execution progresses. On the other hand, the proposed heuristic of overflowing in the task with the least number of overflow pages is independent of the stack usage of the task. The proposed heuristic tries to distribute the overflow space equally amongst all the tasks reducing the chances of overflow of the chosen task itself. Further, it will exhibit less variability with respect to the

amount of stack space recovered as compared to the alternate heuristic.

## 5.8 Profile Independence

The compiler used in the MTSS infrastructure is profile-independent since it does not require profile-information to insert run-time checks at the beginning and end of a procedure. Further, the rolling checks optimization described in section 4 does not require profile information either. However, the MTSS infrastructure as a whole needs the initial size estimate of each task as an input, which is obtained via profiling.

## 5.9 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. 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 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.

# Chapter 6

# Real World Considerations

## 6.1 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.

## 6.2 Communicating Tasks

MTSS does not impact the correctness of implementing communicating tasks. To understand this, consider that most of the communicating tasks 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. Since MTSS only modifies the stack layout and has no impact on globals, the correctness of the implementation remains unchanged. Further, since MTSS does not touch the shared memory segment, no additional synchronization problems are introduced.

## 6.3 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 5.2 to detect overflow. Here, the stack pointer, *sp*, 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 occur when *Overflow-Ptr[$T_x$]* has been read by $T_x$, but thereafter $T_x$ is preempted by a task $T_y$ before the comparison (`sp` $<$ `Overflow-Ptr[`$T_x$`]`) is performed. In this case task $T_y$ can overflow into task $T_x$, allocate a page and increment *Overflow-Ptr[$T_x$]*. When $T_x$ gets the CPU again, it will perform the comparison using the old value of *Overflow-Ptr[$T_x$]*. This can potentially lead to incorrect semantics since $T_x$'s latest stack frame could overlap in memory with an overflow page.

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 $T_y$. In the first case, there is no space on the stack of $T_x$ to allocate the pages needed by $T_y$. 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 space for a page in $T_x$ (first case), then $T_y$ will read *Overflow-Ptr[$T_x$]* from memory and realize that there is no space in $T_x$'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 $T_x$ (second case), then $T_y$ will read *Overflow-Ptr[$T_x$]* from memory, find that there is space, and will allocate a page in $T_x$'s stack. However this is not a problem because of a key observation: the check for overflow is inserted *after* a procedure decrements *sp* to allocate space for its frame. Hence, by the time control switches to $T_y$, the procedure currently executing in $T_x$ would have already allocated its stack frame and needs no

more space. Therefore, here too, no incorrect overlap of $T_x$'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*[$T_x$] 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 5.2, if *Overflow-Started* is true, the result of the check `(sp < Overflow-Ptr[`$T_x$`])` 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 5.2.

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 5.2 (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.

## 6.4 Handling Interrupts

The interrupt stack is either separate or part of the task stack in any embedded system. The former configuration is used if the memory constraint is tight, at a cost of few extra machine cycles during the processing of each interrupt. On the other hand, if the latter configuration is used, the overhead of switching tasks while processing interrupts is reduced at the cost of more memory. 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 5. 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.

# Chapter 7

# Applicable Systems

## 7.1 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.

## 7.2 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 [3]. 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.

## 7.3 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 [33]. 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.

# Chapter 8

# 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) [34] scheme is a hardware approach designed to provide fine-grained memory protection for systems requiring data sharing among processes. Another hardware approach [10] 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) [21]. 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. None of these schemes are related to software-managed TLBs [31] and

software address translation [20], 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 [15, 32]. In older schemes, which used heap-based allocation of stacks [7, 17], 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 [32] 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 [32] 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.

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 [19]. 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, 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 in memory.

In the other attempt for reusing memory across tasks, each thread shares stacks from a stack pool [35, 26]. In [35], 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.

Real-time and scheduling methods that impact MTSS have been discussed earlier in section 7.

Methods for estimating the maximum depth of the stack [29, 9] 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 [16] 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 [6, 5], 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 [6] 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 [6] 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.

# Chapter 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 [13] targeting the ARM7TDMI [2] 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 $\mu C$`linux` operating system [11], modified as needed by the proposed techniques. $\mu C$`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 targetting 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 [14]. We enhance the simulator to enable it to run $\mu C$`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 $\mu C$`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 used in a real embedded system.

# Chapter 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[1], the multi-tasking workloads that are used for evaluation are constructed by combining together multiple benchmarks from MIBench, PTRDist, Olden and Mediabench embedded benchmark suites. Table 10.1 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 7 workloads, each containing four benchmarks, for a total of 28 benchmarks. The first four workloads are from the MIBench suite; the last four 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 10.1, 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 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.

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

| 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 |
| Olden | Perimeter | Perimeters of regions in images | 503 | 1584 |
| | Health | Columbian health care simulation | 759 | 1592 |
| | Voronoi | Voronoi diagram of points | 1380 | 2076 |
| | Bisort | Forward/backward integer sort | 659 | 1614 |
| 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 10.1: Multi-tasking benchmark programs and characteristics

## 10.1 Overheads of run-time checks

Table 10.2 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 4. 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 14.91% to 3.95%; the reduction is significant and makes MTSS a feasible scheme for embedded systems. Similarly, energy overheads are also reduced considerably and go down from 17.42% to 3.76% 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 5. 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 2.4%, but increases the run-time overhead to 5.3% 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* and *olden* workloads have 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 4. Both these factors increase overhead.

Figure 10.1 shows the execution overhead of the run-time checks from different component of rolling checks optimization. As described in Section 4, the rolling checks optimization is split up into three parts: the *certainty optimization*, that rolls checks out of functions that are certainly called by their parents, the *zero size optimization* that rolls checks out of functions with frame size

| **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 | 2.15 | 0.13 | 25.29 | 6.92 | 14.2 | 0.21 |
| Security | 12.79 | 0.83 | 22.07 | 4.98 | 13.66 | 0.93 |
| Telecomm | 11.44 | 4.07 | 27.04 | 7.42 | 12.46 | 3.33 |
| Network | 8.57 | 2.20 | 30.75 | 8.13 | 8.83 | 2.16 |
| Olden | 8.29 | 6.34 | 32.18 | 9.31 | 8.74 | 6.23 |
| Mediabench | 18.36 | 2.05 | 25.79 | 2.55 | 19.46 | 2.12 |
| Ptrdist | 42.36 | 12.04 | 28.7 | 7.49 | 44.57 | 11.30 |
| ***Average*** | *14.91* | *3.95* | *27.40* | *7.33* | *17.42* | *3.76* |

Table 10.2: Overheads for Safety Checks

of zero, 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 10.1 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 *certainity 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 4, rolls checks out of functions that have a frame size of less than $K$ bytes. Therefore, as $K$ increases, the number of

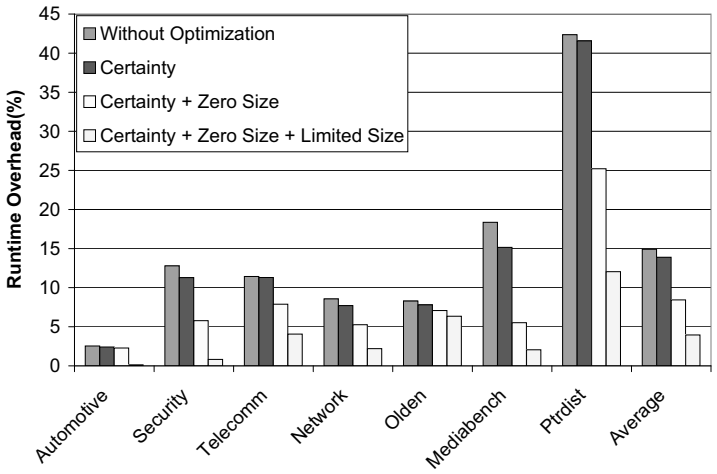


Figure 10.1: Run-time overhead contribution to overflow detection checks from each component of the rolling checks optimization.

functions containing run-time checks will decrease, reducing the run-time overhead. Figure 10.2 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 5.9% to 3.95% 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. We use a limited size threshold value of 128 bytes throughout this paper as a good compromise value.

Table 10.3 shows the comparison of the profile-independent rolling checks optimization with the profile-dependent rolling-checks optimization presented in [6]. 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 4. 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

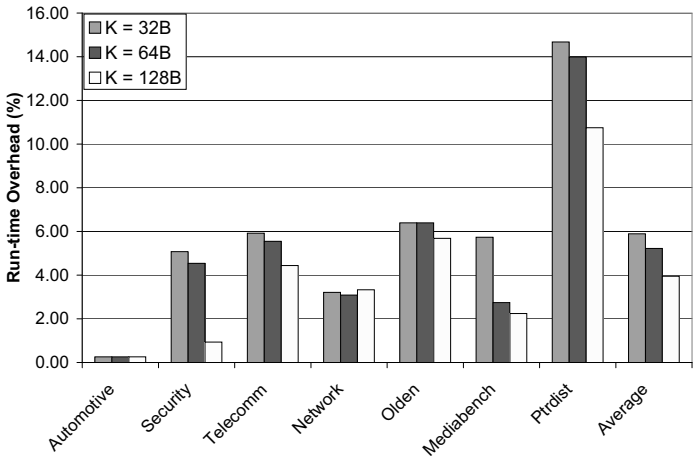


Figure 10.2: Impact of varying limited size threshold on run-time overhead.

penalty of pre-mature overflow declaration. Thus, the profile-dependent scheme has an equivalent *limited-size* optimization proposed in section 4 except that the limited-size parameter $K$ is variable in this case. Further, this scheme also does not insert checks in functions that have a frame size of zero.

Table 10.3 shows that the average run-time overhead of MTSS is 3.9% Vs 5.4% 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. Indeed, a deeper analysis of the workloads reveals that the workloads on which the profile-dependent scheme performs worse than MTSS is when it is unable to roll the check from a function that is certainly called by its parents.

| **Workload** | **Run-time overheads with** | |
|---|---|---|
| | **(MTSS)** | **(Biswas et al.)** |
| Automotive | 0.13 | 6.02 |
| Security | 0.83 | 4.16 |
| Telecomm | 4.07 | 1.30 |
| Network | 2.20 | 5.23 |
| Olden | 6.34 | 5.66 |
| Mediabench | 2.05 | 2.79 |
| Ptrdist | 12.04 | 12.39 |
| ***Average*** | *3.95* | *5.36* |

Table 10.3: Comparison of profile-independent and profile-dependent rolling checks optimization

## 10.2 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 10.3 shows the maximum satisfiable overflow for each task in different workloads. In figure 10.3, 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.

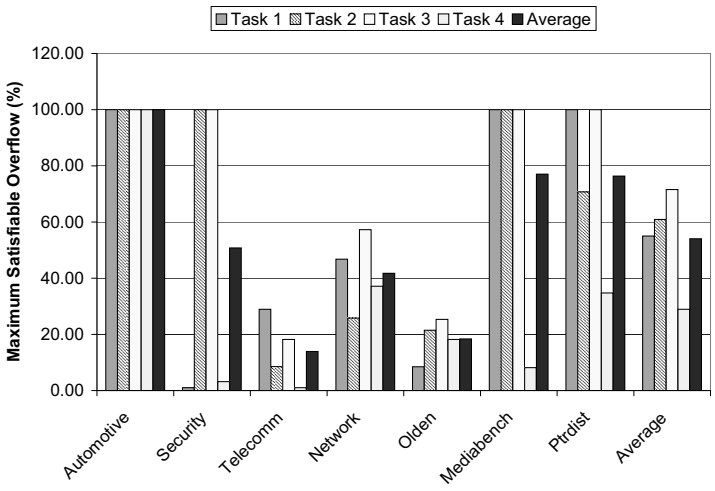


Figure 10.3: Maximum Satisfiable Overflow for different tasks in different workloads.

The numbers in Figure 10.3 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 10.3, 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.

## 10.3 Effect of Page Size

Figure 10.4 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 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.

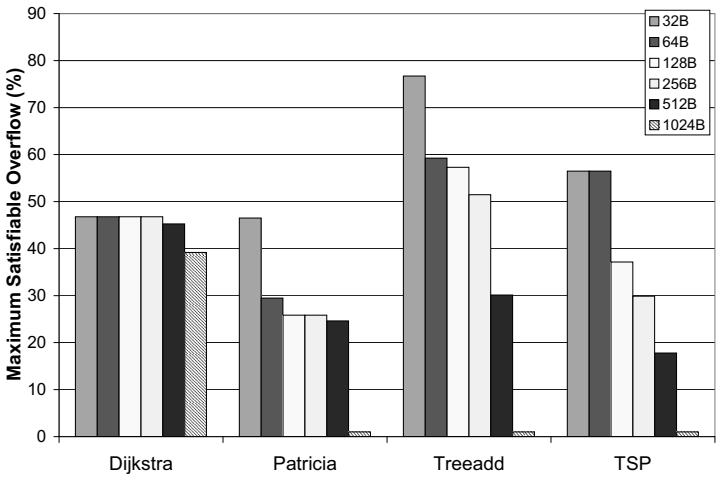


Figure 10.4: Effect of page size on MSO for the network workload

## 10.4 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 10.5 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.

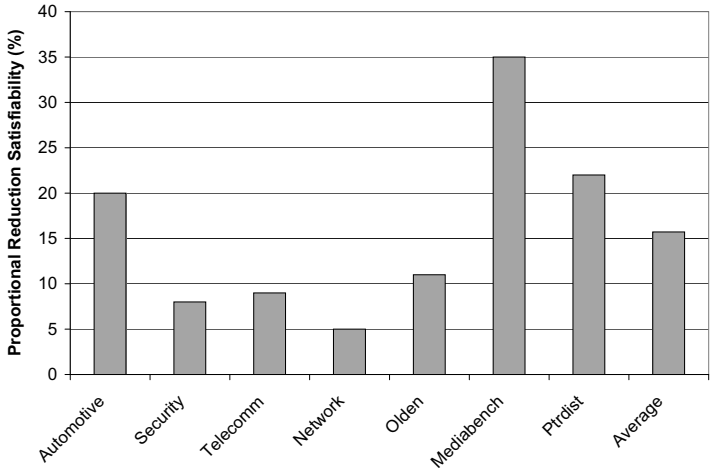


Figure 10.5: Proportional Reduction Satisfiability for different workloads

## 10.5 Comparison with non-contiguous stack allocation

The Capriccio scheme [32] described in Section 8 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 10.4 compares the run-time overheads of our scheme with Capriccio[2]. The table shows that the average overhead from MTSS is 3.9% versus 10.7% from Capriccio. Capriccio

[2]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 [32]. 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.

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, pre-compiling 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 10.4 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 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 | 0.13 | 7.72 | 87 | 2135 |
| Security | 0.83 | 8.18 | 90 | 2138 |
| Telecomm | 4.07 | 7.38 | 6 | 2054 |
| Network | 2.20 | 14.52 | 7 | 2054 |
| Olden | 6.34 | 17.96 | 8 | 2055 |
| Mediabench | 2.05 | 1.77 | 16 | 2064 |
| Ptrdist | 12.04 | 17.27 | 8 | 2056 |
| ***Average*** | *3.95* | *10.69* | *32* | *2080* |

Table 10.4: Comparison of run-time overheads of MTSS and Capriccio.

## 10.6 Real time bounds

Figure 10.6 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 across all workloads in the system is 37.5%. These numbers represent a theoretical and provable upper bound on the run-time overhead of our scheme; the actual run-time increase is usually much lower (it averages 3.9% 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

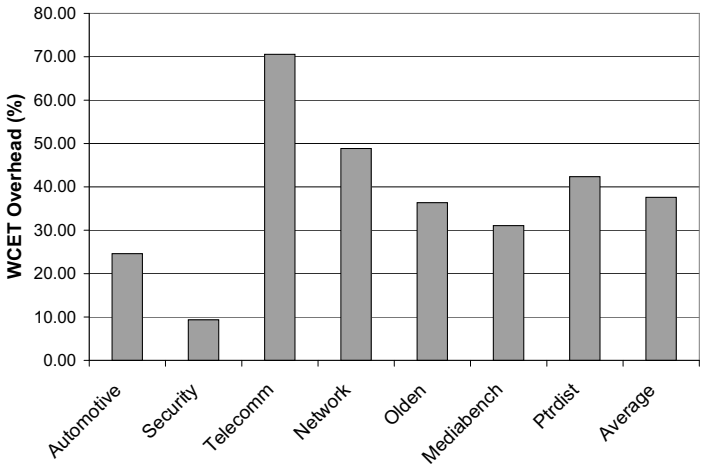


Figure 10.6: Worst case run-time overhead for different workloads

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. We believe this analysis can be refined to lead to a better WCET number; at the time of writing we are attempting to do this.

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 hard-

ware 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 [4].

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.

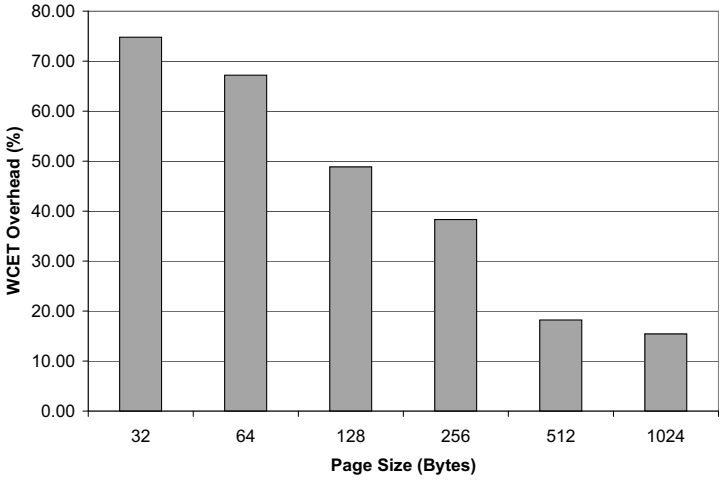


Figure 10.7: Variation of page size on real-time guarantees

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 10.7 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 10.4. 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.

## 10.7 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.

# Chapter 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 3.9% and 3.8%, 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.

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

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

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

[4] M.D. Bennett and N.C. Audsley. 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*, pages 183 – 190. IEEE Computer Society, June 2001.

[5] Surupa Biswas, Matthew Simpson, and Rajeev 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.

[6] Surupa Biswas, Matthew Simpson, Thomas Carley, Bhuvan Middha, and Rajeev Barua. Memory Overflow Protection for Embedded Systems using Run-time Checks, Reuse and Compression. *ACM Transactions in Embedded Computing Systems, To Appear*, 2006.

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

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

[9] Dennis Brylow, Niels Damgaard, and Jens Palsberg. Static checking of interrupt-driven software. In *Proceedings of the 23rd international conference on software engineering*, pages 47–56, May 2001.

[10] John Carbone. Efficient memory protection for embedded systems. *RTC Magazine*, September 2004.

[11] D. Jeff Dionne. uClinux – Embedded Linux Microcontroller Project. 1998.

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

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

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

[15] Dirk Grunwald and Richard 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.

[16] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE 4th Annual Workshop on Workload Characterization*, December 2001.

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

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

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

[20] Bruce L. Jacob and Trevor N. Mudge. Uniprocessor Virtual Memory Without TLBs. *IEEE Transactions on Computers*, 50(5):482–499, May 2001.

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

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

[23] Bill Lamie. A multitasking revolution, 2000.

[24] Bhuvan Middha. MTSS: Multi Task Stack Sharing for Embedded Systems. Master's thesis, University of Maryland, College Park, MD, May 2006.

[25] James Montanaro et al. A 160MHz, 32b, 0.5W CMOS RISC microprocessor. *IEEE Journal of Solid State Circuit*, 31(11):1703–1714, 1996.

[26] Ralph Moore. Unbound stacks and stoppable tasks, 2001. http://www.programmersheaven.com/ articles/smx/article3.htm.

[27] 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.

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

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

[30] Donald McLaughlin Shantanu Sardesai and Partha 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.

[31] Richard Uhlig, David Nagle, Tim Stanley, Trevor Mudge, Stuart Sechrest, and Richard Brown. Design tradeoffs for software-managed tlbs. *ACM Transactions on Computer Systems*, 12(3):175–205, 1994.

[32] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. Capriccio: Scalable threads for internet services. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 268–281. ACM Press, 2003.

[33] Yun Wang and Manas Saksena. Scheduling fixed priority tasks using preemption threshold. In *Proceedings of the Sixth International Conference on Real Time Computer Systems and Applications*, 1999.

[34] Emmett Witchel, Josh Cates, and Krste Asanović;. 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.

[35] Kam-Fai Wong and Benoit 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.