ABSTRACT

RAMACHANDRAN, SHANKAR. Stack Space Analysis for ARM Executables. (Under the direction of Dr. Alexander Dean).

Bounding maximum stack depth for embedded system applications is essential in order to avoid conditions such as stack overflow. Stack usage information is useful for allocating the stack into a memory hierarchy. Prior work in stack space analysis addresses specific issues such as handling interrupts and target ISAs other than ARM.

In this thesis, we propose a methodology for stack space analysis based on euler tour traversal of the call graph to determine the maximum stack depth of the application, good preemption points for tasks and hotspots in stack depth. Our main contribution is Astute (A STack UTilization Estimator), a tool that implements our methodology for stack space analysis of ARM executables. We also study the effects of compiler optimization on maximum stack depth of an embedded system application and benchmark programs.
Stack Space Analysis for ARM Executables

by
Shankar Ramachandran

A thesis submitted to the Graduate Faculty of
North Carolina State University
In partial fulfillment of the
Requirements for the degree of
Master of Science

Computer Engineering

Raleigh, North Carolina

May 2008

APPROVED BY:

Dr. Alexander Dean
Committee Chair

Dr. Yan Solihin

Dr. Xiaosong Ma
DEDICATION

To my parents and my sister.
BIOGRAPHY

Shankar Ramachandran was born in New Delhi, India in 1981. He received his undergraduate degree in Instrumentation Technology from Rashtreeya Vidyalaya College of Engineering, Bangalore, India in 2003. After graduation, he worked in the software industry for about 3 years. He has been a student of the Department of Electrical and Computer Engineering since August 2006.
ACKNOWLEDGEMENTS

I would like to take this opportunity to thank everybody who has contributed to the success of this work. First, I would like to thank my advisor, Dr. Alexander Dean for his insightful advice and patient support. Working with him has been a pleasant learning experience. I would also like to thank Dr Yan Solihin and Dr. Xiaosong Ma, for taking time out of their hectic schedule to serve as members of my thesis committee.

I would like to thank Matt Hazard and Dave Burke of the Aerial Robotics Club (ARC), NCSU, for their support with flightstix. I would also like to thank Martin Sicks of Absint for all the help regarding Stack Analyzer.

Amol Thacker, Deepa Seshadri, Nikhil Goel and Sangyeol Kang deserve a special word of thanks for all the discussions we have had regarding Astute. I would like to thank Sabina Grover and Harsh Thakker for proof reading this thesis. I would also like to thank Jacqueline Kariuki for all the support.

This thesis would not have been possible without the support of my family and friends, who have stood by me through thick and thin of graduate school life.
# TABLE OF CONTENTS

LIST OF TABLES......................................................................................................................vi

LIST OF FIGURES..................................................................................................................vii

Introduction & Motivation.......................................................................................................1

Background & Related Work....................................................................................................7

Tools available & need for a new tool .....................................................................................12
   Need for a new tool..............................................................................................................13

Methodology..........................................................................................................................15

Implementation & Experimental Evaluation.........................................................................19
   Astute.................................................................................................................................19
   Additional tools required..................................................................................................22
   A note on the input to the tool-chain...............................................................................23
   Description of outputs and their interpretation..............................................................26
   Experimental Evaluation.................................................................................................30

Analysis...................................................................................................................................34
   Stack Space Analysis of Embedded System Applications............................................35
   Effects of Compiler Optimizations on stack depth of the application.........................41

Conclusions & Future Work....................................................................................................46

References.............................................................................................................................48
LIST OF TABLES

Table 5.1 Applications used to verify Astute’s results.................................32
Table 5.2 Results of the verification.................................................................33
Table 6.1 Applications from the Mibench suite used for analysis....................35
Table 6.2 Maximum Stack Depth for the applications, determined by Astute......36
Table 6.3 Results of the compiler optimization experiments............................42
LIST OF FIGURES

INTRODUCTION & MOTIVATION

Figure 1.1 Parts of an ARM elf executable .................................................3
Figure 1.2 The MARTS approach .................................................................4
Figure 1.3 Stack Analysis techniques ..............................................................6

BACKGROUND & RELATED WORK

Figure 2.1 Sample call sequence and its corresponding call graph
Representation ................................................................................................8

TOOLS AVAILABLE & NEED FOR A NEW TOOL

Figure 3.1 Classification of existing stack analysis tools and where
the new tool fits in ........................................................................................13
Figure 3.2 The compilers supported by Astute and Stack Analyzer ...............14

METHODOLOGY

Figure 4.1 Euler tour traversal of a tree data structure .................................15
Figure 4.2 Using euler tour traversal for stack usage ..................................16

IMPLEMENTATION & EXPERIMENTAL EVALUATION

Figure 5.1 Components of the Astute tool-chain ...........................................20
Figure 5.2 Components of Astute (the application) ......................................22
Figure 5.3 Flowchart of Astute backend .......................................................24
Figure 5.4 Flowchart of Astute frontend ......................................................25
Figure 5.5 Flowchart of Astute frontend (Contd.) .........................................26
Figure 5.6 Sample Per Function Stack Usage plot ......................................27
Figure 5.7 Sample Call Graph .................................................................27
Figure 5.8 Sample Euler Tour Traversal Stack Usage plot ......................28
Figure 5.9 Sample Static Stack Depth Distribution plot ..........................29
Figure 5.10 Compilers supported by Astute ...........................................32

ANALYSIS

Figure 6.1 Per Function Stack Usage plot for the bitcnts benchmark .........37
Figure 6.2 Euler Traversal Stack Usage plot for the bitcnts benchmark .......38
Figure 6.3 Static Stack Depth Distribution plot for the bitcnts Benchmark .................................................................39
Figure 6.4 Distribution of library functions & user defined functions with recursion .................................................................39
Figure 6.5 Call graph for the bitcnts benchmark executable, compiled with no optimizations. .................................................................40
Figure 6.6 Change in maximum stack depth for the set of benchmarks .......41
Figure 6.7 The stack usage of functions in the bitcnts benchmark ..........43
Figure 6.8 Call graph for the bitcnts executable, compiled with –O3 optimization flag .................................................................45
INTRODUCTION & MOTIVATION

Embedded systems have become ubiquitous. They are used in a simple microwave, a savvy mobile hand held device or even a complex robotic arm. The challenges in the embedded system domain are unique, as the quest for better application performance, more functionality go hand in hand with the need for tighter constraints.

The most significant aspect that differentiates embedded systems from others is their strict timing and space constraints. These constraints arise because of the nature of these applications. For example, in a system that is designed to set off the fire sprinklers in response to an input signal from a sensor, the response time of the system has to be ensured for safety. Typically, these applications run on dedicated hardware that has limited memory. Hence they have stringent space constraints as well.

Embedded processors used in low end embedded systems are different from general purpose processors. Such embedded processors typically have few/ no pipeline stages. They usually have cacheless memory systems. They may have a heterogeneous memory hierarchy, where the onus of allocating data into this hierarchy is on the developer/ compiler. The resource constraints play an important role not only because the processors have access to limited memory, but also because of the nature of the applications that run on embedded processors. The memory utilization of applications that run on general purpose processors is managed by the operating system. These applications have a shorter lifetime (they can be terminated by the user once the objective of the application is accomplished) as compared to those that run on embedded processors. Embedded applications are typically expected to run eternally, unless there is an error condition that the system is not able to handle. The eternal
lifetime of the embedded applications makes their memory utilization analysis a critical component of their design.

Embedded systems can be monitored dynamically (i.e. during execution) or analyzed statically (before execution, pre / post compilation). One of the popular techniques for monitoring these systems is to add instrumentation code to the application, which monitors the state of the system at run time. Static analysis is a suitable approach for embedded systems, since they need to be guaranteed to meet their space and timing constraints before they be used. As mentioned earlier, a part of the reason for the stress on this guarantee is the safety aspect, but for embedded systems that are designed for mass production, the economic aspect plays a significant role as well. Through analysis of the application, the developer can make a sound judgment about the memory allocation policy thus improving the run time efficiency of the system at lower cost.

Significant research has been carried out on embedded systems, primarily in determining the worst-case execution time. This, however, is not the focus of this thesis; the other domain of static analysis – space, is the focus of this thesis. As with timing analysis, space analysis is also motivated by the safety and economic aspects of embedded systems. Insufficient static space analysis can lead to pathological conditions such as stack overflow [1], illegal memory references and uneconomical overestimation of the required memory.

Generally, the variables used by an executable are split into various sections in memory. These sections are logical divisions of the executable’s memory space. The initialized static variables in the program are usually in the data section, while the uninitialized static variables are in the bss section. Automatic variables are stored in the stack area. This can be seen in figure 1.1.
One of the areas of research at the Center for Efficient, Scalable and Reliable Computing (CESR) is Memory Allocation for Real Time Systems (MARTS). The MARTS framework aims at allocating the data in a program into heterogeneous memory in an optimal manner through integer linear programming and preemptive threshold scheduling, such that the real time scheduling constraints are met. Choosing appropriate points of preemption in such a system of tasks can lead to optimizations in the memory usage of the application. The applications used in this research are statically linked and the tasks are statically loaded. The MARTS framework is shown in Figure 1.2
The topic of this thesis is the analysis of stack space usage of embedded systems. More specifically, embedded system applications, which are compiled for ARM architecture using the arm-elf-gcc compiler, are analyzed to estimate stack related information. The motivation behind this work is multifold. Stack usage estimates are essential

- To ensure that the real time application meets its space constraints.
- To improve the efficiency of an embedded system that has a memory hierarchy composed of multiple memory units with variable access latencies. Information regarding stack usage aids in allocation of the stack into the memory hierarchy.
- In relation to the research in Memory Allocation with Real-Time scheduling (MARTS) [2], the stack usage information of the tasks can be used while choosing preemption points.

Figure 1.2: The MARTS approach
To study the effect of compiler optimizations on stack space usage of embedded system applications.

Stack space analysis can be performed analytically or experimentally. High-water marking is a popular experimental technique. The stack is filled with an uncommonly occurring pattern and the application is executed. By observing the point in the stack up to which the painted pattern has changed, the observed maximum stack depth can be calculated. The disadvantage of this approach is that the observed maximum stack usage may not be the worst-case stack usage at run time as seen in figure 1.3. The other approach is to analyze the stack usage statically. The benefit of this approach is that the maximum stack depth can be bound & hence the real time properties of the system can be ensured.

Another way of viewing stack space analysis techniques is classifying them as static or dynamic techniques. Analytical stack space analysis techniques fall under the static category. The stack can be dynamically monitored, with which the changes in the stack depth can be monitored for example, per machine cycle. Dynamic information can used to do optimizations on the fly for stack memory. Static techniques are equally important, especially in embedded systems since bounding the maximum stack depth is usually critical for the safety of the system. The specific contributions of this thesis are:

1. Methodology and implementation for analytical stack space analysis - Astute.
2. Study of the effects of compiler optimizations on maximum stack depth of an application.
Figure 1.3 Stack analysis techniques
BACKGROUND & RELATED WORK

Stack space analysis of programs is primarily based on their call graph representation. A call graph (also known as a call multigraph) is a directed graph that represents calling relationships between subroutines in a computer program. Specifically, each node represents a procedure and each edge \((f, g)\) indicates that procedure \(f\) calls procedure \(g\). [3].

A sample program (only calling sequence shown for simplicity) and the corresponding call graph is shown in figure 2.1. By traversing the edges of the call graph from top to bottom, one can estimate the stack space required. The calling conventions vary from one architecture to the other. [4] and [5] specify the calling convention used in ARM processors. The specific responsibilities of the calling function and that of the called function are as follows:

- The calling function allocates stack space for the called function’s arguments
- The called function is responsible for allocating stack space for its automatic variables.
- The called function is responsible for de-allocating the stack space, which had been allocated for automatic variables, after it has completed execution.
- The calling function is responsible for de-allocating the stack space it had allocated for the called functions arguments.

The grey areas in stack space analysis are:

1. Interrupts

An interrupt is an asynchronous signal from hardware indicating the need for some action or a synchronous event in software indicating the need for a change in execution [6].
Interrupts are widely used in embedded systems. The reason they are problematic in static stack space analysis is because an interrupt can potentially occur at any instant and can occur multiple times. When there is more than one interrupt (which is usually the case), the interrupts have priorities. Also, interrupts may be partially or completely disabled in certain functions. These factors contribute to complications in stack space analysis of programs with interrupts. Significant research has been done in this area by Regehr et al [7]. In [7], through context sensitive abstract interpretation of machine code, Regehr et al determined the worst-case stack bound in the presence of interrupts. In our work, we have not given special attention to interrupts. The focus is on determining the maximum stack

Figure 2.1: Sample call sequence and its corresponding call graph representation. Note that the green dotted arrows indicate indirect recursion and the red dotted arrows indicate direct recursion.
usage for systems with preemptive threshold scheduling, in which the optimal memory allocation is derived. Hence an over estimation of stack space contributed by considering the stack space for interrupts as the sum of the individual stack space usage, would not hamper the choice of preemption points.

2. Recursive function calls

Recursion, though an elegant solution for many problems, is not suitable for embedded systems. Static analysis of recursive functions is non trivial because of the variability in the number of recursion loops – the actual recursion bound can be observed only at runtime. Recursion can be direct (a function calling itself) or indirect (a loop in the call graph). This is depicted in figure 2.1. In [8] Blieberger et al propose that recursion is not necessarily harmful in real time embedded systems as long as they satisfy certain conditions that make them statically analyzable. In [9] Blieberger extends the theory to indirect recursion and provides a theoretical approach of finding the maximum stack space used by a chain of functions in indirect recursion. Although the recursion bound can be determined by following the approach in [7], it would still result in an equation based on a parameter(s) that decides when recursion would cease. This parameter is usually an argument to the recursive function or is derived based on the arguments. Hence, statically recursion bound cannot be determined accurately. Certain simple heuristics may be applied on the recursion bound; for example, in embedded systems, loops in the call graph observed from a regular routine to an error handling routine would most probably cease at the error handling routine. Our approach is to allow the developer to specify recursion bounds, based on design criteria or results of the instrumentation code, and to handle recursion appropriately.
In [10] Engblom studied 5579 functions from 334600 non commented lines of commercial embedded real time C code and found 14 of the 5579 to be recursive. He suggests that recursion can be ignored in early WCET tools. Stacktool [11] ignores recursion in the input. Commercial stack analysis tools such as Stack Analyzer [12] from Absint Ltd. and Bound-t [13] from Tidorum, allow the user (developer) to specify the recursion bounds as an input to the tool.

3. Indirect function calls
Indirect function calls are implemented in C through the use of function pointers. Since indirect function calls rarely occur in commercial embedded system code [10], we have ignored analysis of indirect function calls for this thesis. Global pointer analysis is required to resolve indirect function calls. Stack tool ignores indirect function calls. Stack Analyzer does handle indirect function calls.

4. Self-modifying code
Self-modifying code is defined to be one that modifies its own instructions, intentionally or otherwise, while it is executing [14]. From the definition it is apparent that static analysis of application that contain self-modifying code would be imprecise. Self-modified code is not handled in this study.

Since static analysis is critical for embedded system applications, one direction of research has been to start from scratch, i.e. to design a programming language tailored to meet the needs of embedded systems. Hume [15] is an example of such a language. It is a hybrid language that combines functional programming paradigms with concepts from hardware design [15]. Applications written in Hume are automatically guaranteed of meeting space and timing constraints because of the programming constructs used.
In the domain of embedded systems, currently, the use of procedural languages is more common than the use of functional/hybrid languages. In the beginning, assembly language was used for embedded programming as it gave the developer direct access to hardware and one could conceivably write the most efficient code in assembly language. For ease of programming, code is now written in high-level languages such as C, C++ and Java. In [16] Unnikrishnan et al study automatic stack space analysis of code written in high-level languages. Memory utilization analysis of high-level languages has become increasingly important in order to ensure better quality of software.

Another approach is to analyze the executable. The compiler, if optimizations are enabled, optimizes the source code to produce an executable whose stack space usage could be different from that derived by analyzing the sources before compilation. Source code of kernel code or library code may not be available for stack space analysis; however, if the executable is statically linked, then all code the application uses can be analyzed. Due to these advantages, analyzing the executable is a preferred approach in stack space analysis. In [7] Regehr et al analyze AVR executables. Our stack analysis approach also targets executables.
TOOLS AVAILABLE & NEED FOR A NEW TOOL

The tools available, commercial as well as those developed for research, are reviewed below by categorizing them into groups for easier analysis.

- Based on the type of input (to the tool)

Stack analysis can be performed on the source code or the executable. Tools that work on the source code can be handicapped because of library code, since the sources for libraries may not be available. Stack analysis may be integrated into the compiler, or the compiler may be modified to optionally provide information for stack analysis. Compiler optimizations affect the stack usage of an application. By performing stack analysis as a post compilation step; the analysis can be made agnostic to the effects of compiler optimizations.

GNATstack [17] is an example of a stack analysis tool for Ada, which uses the information generated by the compiler (of the GNATPro tool chain). Stack Checker [18] can process AVR assembly, GAS format assembly and avr-objdump format assembly. Bound-t [19], Stack Analyzer [20] and Stacktool [21] can process executables as input.

- Based on Architecture

Stack space analysis is closely tied to the underlying hardware architecture. GNATstack can work on a range of architectures (those architectures which are supported by GNATPro compiler, including AVR). Since AVR microcontrollers are widely used in embedded systems, most of the tools support AVR architecture. Stacktool and Stack Checker support only AVR architecture. Commercial tools such as Bound-t and Stack Analyzer support common architectures (including AVR).

Figure 3.1 gives a summary of the tools and their classification.
Figure 3.1: Classification of existing stack analysis tools and where the new tool fits in.

3.1 Need for a new tool

The specific requirement for our research was a tool that would take an ARM executable as input and provide information regarding stack usage per task. Stack Analyzer is the only available tool that is known to support ARM architecture, however, the two compilers that it is known to support are the ADS [22] and TI compiler. For our research, the compiler used is arm-elf-gcc [23]. Figure 3.2 shows the compilers supported by Stack Analyzer and the new tool (Astute). Figure 3.1 shows where the tool fits in along with the other available tools.
Thus, there is need for the new tool for stack analysis of elf executables (generated by arm-elf-gcc compiler). Since the GNU ARM tool-chain is a widely used in research & in the open source industry projects, support for stack analysis of executables generated by arm-elf-gcc compiler would be beneficial.
METHODOLOGY

This section explains our methodology for stack space analysis. Our methodology is influenced by the needs of the research pertaining to MARTS and our observations regarding real time embedded software applications. The salient features are discussed below:

- The call graph can be viewed as abstract tree representation of the program, the nodes of the tree representing individual functions and edges representing function calls.

- With the information about the children of each node (subroutine calls of each function) the tree can be traversed. The approach is to store this information in separate structures (a structure per function) and conceptually traverse the tree without actually constructing it.

- Traditionally, Euler tour traversal [24] is discussed in the context of a binary tree; however, in this case, the tree can be non-binary (a node may have more than 2 children). Euler tour traversal can be visualized as walking around the tree as if the tree was a walled fort and we are walking along the outer walls of the fort.

![Euler tour traversal of a tree data structure. The numbers in the arrows indicate the order of traversal.](image-url)

Figure 4.1: Euler tour traversal of a tree data structure. The numbers in the arrows indicate the order of traversal.
starting from the root of the tree, visiting all the nodes and returning back the root.

Figure 4.1 shows the euler tour traversal of a tree starting at root A. The numbers within the arrows indicate the order of traversal. In this case the traversal sequence would be A, B, D, B, E, B, A, C, F, C, G, C and A.

- The euler tour traversal of the call graph is most suitable for stack analysis since it models the behavior of the stack upon function calls in the program. The behavior of the stack is different when a node is first visited (it increases, i.e. for a stack that grows upwards the stack pointer reduces) and when it is last visited (it decreases, i.e. for a stack that grows upward the stack pointer increases). Intermediary visits of the node may affect the stack (if a subroutine call uses the stack to pass arguments, then the stack increases). For example in Figure 4.2, if the euler traversal begins at node “a” (the root of the tree), the stack usage in the traversal increases now by 5 (the stack usage of function “a”). By euler tour traversal, nodes b, c and d are visited the first time, thereby increasing the stack usage in the traversal to 26.

![Sample sequence of calls](image)

Figure 4.2: Using euler tour traversal for stack usage. The stack usage of the function is indicated beside the node.
After the path d, c, e is traversed, node c is visited for the last time and the stack usage of “c” (7) is reduced from the stack usage in the euler traversal.

- The algorithmic complexity of such stack analysis is dependant on the algorithmic complexity of the euler tour traversal and the complexity of the actions performed when the nodes are visited. The actions basically involve searching for elements in a map data structure and searching, adding or deleting elements in a vector data structure.

The algorithmic complexity of the euler tour traversal of a tree is $O(n)$, where $n$ is the number of nodes in the tree. Searching for elements in a map data structure costs $O(1)$, while searching for elements in a vector has an algorithmic complexity of $\log(n)$. Adding an element in a vector can be done in $O(1)$ while deletion of an element involves searching for the element too. Hence the algorithmic complexity for deleting of an element in a vector is $\log(n) +1$.

The overall worst case algorithmic complexity for stack space analysis is given by:

$$O(n) = n + (\log(n) + \log(n) +1 +1 ) n$$

This expression reduces to $n\log(n)$.

- Although the use of recursion (direct or indirect) is not recommended for embedded systems applications, we observed recursion in many of the benchmarks used [25]. Hence, the stack analysis methods are designed to recognize recursion in the call graph and account for it appropriately based on the specified recursion bounds. The functions in the recursion loop are identified and their stack usage is summed. This sum is multiplied by the specified recursion bound.
Recursion, theoretically, does not hamper euler tour tree traversal, since it can modeled as duplication of a branch of the tree. A simpler way to handle recursion during euler tour traversal is to account for it (in this case, in terms of stack usage) when it is detected instead of duplicating the branch.

To summarize, our methodology for stack space analysis is to perform an euler tour traversal of the call graph and perform different actions depending on whether it is the first, last or an intermediary visit of a node. Recursion is detected and accounted for in stack space analysis.
IMPLEMENTATION & EXPERIMENTAL EVALUATION

5.1 Astute

To implement our methodology for stack space analysis we have developed a tool-chain called Astute, A STack UTilization Estimator. Astute works on the elf executable that is generated when an application is compiled with arm-elf-gcc compiler of the GNU ARM tool-chain [26]. Performing stack analysis as a post compilation step has benefits as mentioned in the chapter “Background & Related Work”.

Astute tool-chain is composed of two GNU utilities and a C++ application. The development environment is Linux (fedora core 6). The components are shown in figure 5.1. The components are briefly described below:

- arm-elf-objdump
  This utility is a part of the GNU ARM tool-chain available at [26]. arm-elf-objdump is a disassembler for executables that are in elf format (generated by the arm-elf-gcc compiler). Since the compiler used is arm-elf-gcc (which is a part of the GNU ARM tool chain), the corresponding disassembler – arm-elf-objdump is used to disassemble the executable. The executable code in an arm-elf binary is in the text section. Hence, using the options –d –j. text with arm-elf-objdump, the text section of the executable is disassembled. Further information about arm-elf-objdump can be found at [27].

- Awk scripts
  Awk [28] is a popular data processing utility in UNIX based systems. It is most suitable and has powerful constructs for building text processing based scripts. Two awk scripts are used to parse the disassembly for instructions pertaining to function headers (for
Figure 5.1: Components of the Astute Tool-chain. The dotted arrows indicate temporary outputs.

example <function name >:), function calls (for example bl <function name>) and stack related information (for example push <register>). The process of parsing the
disassembly for information is split between Astute (the application) and the awk scripts, where the scripts initially filter out unrelated instructions and more rigorous information processing is done by the Astute backend.

- Astute

Astute is an application developed in C++. It has three components: backend, command-line and frontend. These components are shown in figure 5.2. Astute creates the following output files:

- Per function stack usage plot for the application.
- Call graph per task
- Euler tour traversal stack usage per task
- Static stack depth distribution plot per task
- Maximum stack depth per task

These output files and their implications are explained in the section 5.4.

The command-line component parses the command line parameters and invokes appropriate functions in the backend and frontend. For the command line options supported please see the detailed documentation of Astute.

Figures 5.3, 5.4 and 5.5 explain the working of Astute backend and front-end (5.4 and 5.5) respectively. Astute backend is responsible for parsing the information from the disassembly (the partially parsed output of the awk scripts) and storing this information in data structures. The backend is conceptually the first stage in stack analysis.
Figure 5.2 Components of Astute (the application). The dotted line indicates that it’s not strictly a component of Astute; it represents the data structures used by Astute.

Astute front-end traverses the call graph based on the information in the data structures. Astute command-line calls functions from Astute front end after the backend has completed parsing the disassembly. Apart from gathering stack related information, the front-end also detects recursion in the call graph (direct as well as indirect). The user can specify recursion bounds; if Astute encounters a case of recursion that is not specified by the user, then it would assume that no recursion occurs there and report those recursion details to the user. Finally it writes all the information onto output files. For further information on the working of Astute please refer to the detailed documentation of the application.

5.2 Additional tools required
To view the graphical outputs generated by Astute additional tools are required. Though these plots could have been generated by the Astute tool-chain, by invoking the appropriate
tools through command line; this part of the process is deliberately separated so that the user could use any graph visualization tool of her/his choice.

The tools used to generate plots for this thesis are gnuplot [29] and graphviz [30]. Gnuplot is a command-line driven graph plotting utility, which works on multiple platforms including UNIX and Windows. Astute generates scripts that can be loaded in gnuplot to generate the graphs. The graph can be saved in popular file formats including postscript. To use any other graph visualization tool, the data file corresponding to every graph (.dat file) generated by Astute can be used. Graphviz is open source graph visualization software. It interprets input in DOT language to form a hierarchical directed graph using the dot engine. Graphviz has a GUI where the raw .dot files can be edited and the graphs can be viewed. It can also be invoked through command line. It supports popular file formats including postscript.

5.3 A note on the input to the tool-chain
As mentioned earlier, the Astute tool-chain primarily works with elf executables generated by the arm-elf-gcc compiler. It can also be used with elf executables generated by the ARM Development Suite (ADS) [31] compiler.

To summarize, the tool-chain Astute is composed of a disassembler, text parser scripts and a C++ application (Astute). Astute in turn contains three components- the backend (parses the information and stores it in data structures), the front-end (traverses the call graph to do stack space analysis) and the command-line (parses the command line options and calls appropriate functions in the backend and front-end components). Additional tools (gnuplot and graphviz) are required to view the graphs and plots generated by Astute.
Start

Parse task file and store the information in a map accessed through the task name

Parse recursion file and store the information in a map accessed through the function name

Parse the disassembly for function names and subroutines and store each function name and the name of its subroutines in a separate structure. Link the structure to a map accessed by the function name

Parse the disassembly for interrupts and account for them in stack usage of other functions

Parse the disassembly for stack related instructions and store the stack usage information in the structure associated with each function

Stop

Figure 5.3: Flowchart of Astute backend.
Figure 5.4 Flowchart of Astute front end
5.4 Description of outputs and their interpretation

The output of the Astute tool-chain can be broadly classified into two categories:

- Human Oriented
- Machine Oriented

The human oriented outputs are discussed below:

- Per function stack usage plot

  This is a plot of the stack usage against the function names, in descending order of stack usage. One per function stack usage plot is generated for the application. The plot aids the developer in choosing functions to optimize, so as to reduce their stack usage. Figure 5.6 shows a sample per function stack usage plot generated by Astute.
Figure 5.6: Sample Per function stack usage plot

• Call graph

A call graph is generated per task. It is a hierarchical representation of the call sequence in the program. The call graph is annotated with stack information for convenience. The recursion bounds are depicted along the edges. The call graph gives a graphical view of calling sequence the program. A sample call graph is shown in figure 5.7.

Figure 5.7: Sample Call graph plot.
• Euler tour traversal stack usage plot

This plot is also generated per task. Euler tour traversal is described in the section “Methodology”. This is a plot of the stack usage after every step/hop in the euler tour traversal of the call graph. The plot aids in choosing preemption points for tasks in systems with preemptive threshold scheduling. By choosing preemption points where the stack usage is low, we open the door to other possible optimizations to be determined through future work. Figure 5.8 shows a sample euler tour traversal stack usage plot. Here, for example if this was a system with two tasks- task1 and task2. Consider the solid line to be representative of the euler traversal stack usage plot for taks1. If it is known that the maximum stack depth for task2 is 250 bytes, then by choosing preemption points at hop #2 or hop# 5 rather than hop #8 or hop # 9 in the euler tour traversal of task1, task1 and task2 can safely share the same stack memory.

![Sample Euler tour traversal stack usage plot](image)

**Figure 5.8:** Sample Euler tour traversal stack usage plot. The dotted lines are for the traversal without interrupts while the solid line is for the traversal with an interrupt whose stack usage is 50 bytes.
Static stack depth distribution plot

The static stack depth distribution plot can be described using an example. Figure 5.9 shows the stack depth distribution (determined statically) of a sample application. On the X-axis is stack space in bytes and on the Y-axis is statically determined frequency. It is a frequency distribution plot. For example the vertical bar of height 4 units between 152 -171 bytes of stack depth indicates that during the euler tour traversal of the call graph for this particular task, the stack depth was determined to be between 152 bytes to 171 bytes four times.

The stack depth can be monitored dynamically (at run time) to produce a plot of the stack depth v/s cycles (or instruction count). This is different from the static

![Static stack depth distribution plot](image)

Figure 5.9: Sample Static stack depth distribution plot.
stack depth distribution plot in that it gives a time perspective of the stack depth and is at a finer granularity (cycles/instructions as against function calls in the static stack depth distribution plot).

This plot can be used to make a sound decision regarding the memory allocation. Typically, embedded systems (cache-less embedded systems) have fast scratch pad memory. Knowledge of the stack depth hot spots can help in allocating the stack into the memory hierarchy to get better run time performance.

The machine-oriented output is the maximum stack depth per task. This is displayed on the screen (for the benefit of the user) and also stored in an output file. This output file can used as input to other tool-chains developed as a part of the research in preemptive threshold scheduling, such as ARMSAT [32] and OMMA [33]. The maximum stack depth is critical for various reasons, some of which are mentioned below:

- Maximum stack depth bounds the stack growth and hence if the stack is appropriately allocated, pathological conditions such as stack overflow can be prevented.
- Since memory is a costly & a limited resource in real time embedded systems, a task’s maximum stack usage information may be factored into the scheduling algorithm.

5.5 Experimental Evaluation

Since Astute is an analytical stack analysis tool, it could be verified by comparing its results against the results from a reference stack analysis tool or against results from actual runs of the executables (using high water marking). High water marking is not suitable for verifying the results since Astute calculates the analytical maximum stack space used by the
executable, while high water marking would only provide the maximum stack usage for a certain input data set.

Stack Analyzer is an appropriate tool to compare with Astute, since it provides the analytical bound on the maximum stack space for the executable and is widely used in the industry for stack analysis. The test environment is as follows:

- **Target Architecture**: ARM. (ARM7TDMI [34] i.e. ARMv4 core [35]).

- **Applications examined:**

  The applications used for verifying Astute’s results (with that of Stack Analyzer’s) are a collection of sample programs that verify specific aspects of Astute such as its ability to handle recursive procedures. They include two benchmark programs- Dhrystone [36] and EDN [37]. These applications are a part of the Stack Analyzer installation package. Please see table 5.1 for description of these applications. These applications have been compiled with the ADS compiler.

- **Compiler**

  The compiler used is the ADS compiler.

Though Astute was originally designed to handle only elf files produced by the arm-elf-gcc compiler, it has been upgraded to support elf files produced by the ADS compiler too. This is shown in figure 5.10.
Fig 5.10: Compilers supported by Astute.

The set of applications in table 5.1 were analyzed using Stack Analyzer and Astute. The verification criterion is the maximum stack depth obtained from the two tools. The results are as shown in table 5.2. The results show that Astute’s results match that obtained from Stack Analyzer except for the application where functions pointers are used (pa.elf). This is because function pointers are not handled in Astute, while this is handled in Stack Analyzer (through global pointer analysis).

Table 5.1: Applications used to verify Astute’s results.

<table>
<thead>
<tr>
<th>Name of executable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prime.elf</td>
<td>Simple program which determines if a number is prime</td>
</tr>
<tr>
<td>Fibfac.elf</td>
<td>Use of recursion - Calculates the nth factorial and the nth Fibonacci number.</td>
</tr>
<tr>
<td>Pa.elf</td>
<td>Program demonstrates the use of function pointers.</td>
</tr>
<tr>
<td>Edn.elf</td>
<td>A sample program from the EDN benchmark</td>
</tr>
<tr>
<td>Dry2_1.elf</td>
<td>A sample program from the Dhrystone benchmark</td>
</tr>
</tbody>
</table>
Table 5.2: Results of the verification.

<table>
<thead>
<tr>
<th>Name of the executable</th>
<th>Maximum Stack depth from Stack Analyzer (bytes)</th>
<th>Maximum Stack Depth from Astute (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prime.elf</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>Fibfac.elf</td>
<td>104</td>
<td>104</td>
</tr>
<tr>
<td>Pa.elf</td>
<td>40</td>
<td>0</td>
</tr>
<tr>
<td>Edn.elf</td>
<td>1340</td>
<td>1340</td>
</tr>
<tr>
<td>Dry2_1.elf</td>
<td>136</td>
<td>136</td>
</tr>
</tbody>
</table>

Hence, the maximum stack depth determined by Astute has been verified for the applications in table 5.1.
ANALYSIS

This section describes the stack space analysis of real time embedded system applications and the effect of compiler optimization on stack depth, using Astute.

The test environment is described below:

- **Target Architecture:** ARM. (ARM7TDMI, Xscale [38], i.e. ARMv4 core and ARMv5 core [39] respectively).

- **Applications examined:**
  
  - Six applications from Mibench suite (One from each category) were used. Please see table 6.1 for description of these applications. Precompiled application binaries (compiled using arm-elf-gcc compiler), available at [40] were used. The applications are all single threaded.

  - Flightstix, an autopilot software developed at the Aerial Robotics Club (ARC) [41], North Carolina State University is used. Since the application is still in development phase, the sensor module of flightstix was used. The sensor module has three tasks/threads – the main thread, the magnetometer thread and the GPS thread. The application was compiled using the arm-elf-gcc compiler.

  - For the compiler optimization related experiments, primarily, the bitcnts benchmark of the Mibench suite was used, as its stack depth showed high sensitivity to compiler optimizations.

- **Compiler**

  The compiler used is arm-elf-gcc (version 3.4.3) of the gnu arm tool-chain available at [42].
Table 6.1: Applications from the Mibench suite used for analysis.

<table>
<thead>
<tr>
<th>Name of Executable</th>
<th>Mibench Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Susan.arm</td>
<td>Automotive</td>
<td>An image processing application</td>
</tr>
<tr>
<td>Dijkstra_small.arm</td>
<td>Network</td>
<td>Simulates Dijkstra's shortest path algorithm</td>
</tr>
<tr>
<td>Toast.arm</td>
<td>Telecomm.</td>
<td>The global standard for mobile communications is a standard for voice encoding/decoding.</td>
</tr>
<tr>
<td>Say.arm</td>
<td>Office</td>
<td>Text speech synthesis application</td>
</tr>
<tr>
<td>Sha.arm</td>
<td>Security</td>
<td>Secure hash algorithm that produces a 160 bit message digest for a given input.</td>
</tr>
<tr>
<td>Madplay.arm</td>
<td>Consumer</td>
<td>MPEG audio decoder.</td>
</tr>
<tr>
<td>Bitcnts</td>
<td>Automotive</td>
<td>Tests the bit manipulation ability of the processor</td>
</tr>
</tbody>
</table>

6.1 Stack space analysis of Embedded System Applications

The maximum stack depth for the Mibench applications and flightstix is given in table 6.2. The application that stands out in the table is Susan. The function main in Susan has a stack usage of 360224 bytes, which primarily contributes to the high maximum stack depth of the application. Also, as expected the static stack usage distribution plot shows a high concentration between 352305 bytes to 391450 bytes.
Table 6.2: Maximum stack depth for the applications, determined by Astute.

<table>
<thead>
<tr>
<th>Name of executable</th>
<th>Maximum Stack depth from Astute (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Susan.arm</td>
<td>391452</td>
</tr>
<tr>
<td>Dijkstra_small.arm</td>
<td>21956</td>
</tr>
<tr>
<td>toast.arm</td>
<td>27176</td>
</tr>
<tr>
<td>Say.arm</td>
<td>22444</td>
</tr>
<tr>
<td>Sha.arm</td>
<td>21236</td>
</tr>
<tr>
<td>Madplay.arm</td>
<td>81708</td>
</tr>
<tr>
<td>Flightstix (Task1 - main thread)</td>
<td>4576</td>
</tr>
<tr>
<td>Flightstix (Task2 – gps thread)</td>
<td>3148</td>
</tr>
<tr>
<td>Flightstix (Task3 – magnetometer thread)</td>
<td>3304</td>
</tr>
</tbody>
</table>

Analysis of the disassembly shows that a single instruction is responsible the bulk for the stack growth:

Sub sp,sp,#356352.

This instruction alone increases the stack usage of main by 356352 bytes. This allocated stack memory for a local variable, which is an array of structures (the variable corner_list in the source code susan.c). The size of the data section and the bss section for susan.arm are 4240 bytes and 3748 bytes respectively.

Fig 6.1, 6.2 and 6.3 show the plots produced by Astute while analyzing bitcnts benchmark.

The “Per function stack usage plot” (figure 6.1) shows the stack usage of the functions arranged in descending order. The function _dtoa_r has the maximum stack space usage (152 bytes). If the developer wants to target specific functions for stack optimizations, _dtoa_r
Figure 6.1: Per function stack usage plot for the bitcnts benchmark.

would give a better result than applying optimization to a function such as ntbl_bitcnts, which has a stack usage of only 25 bytes.
The euler traversal stack usage plot (figure 6.2) can be used to choose the preemption points. For example, choosing a preemption point such as the 18th hop/call (see figure 6.2) on the euler traversal would be more costly in terms of memory than choosing hop/call 6.

The Static stack depth distribution plot (figure 6.3) shows that the stack depth is most frequently between 95 and 114 bytes. In embedded systems that have scratch pad memory, the optimal stack allocation to fast memory (scratch pad memory) can be based on the Static stack depth distribution plot; in this case it would be optimal to allocate 95 – 114 of scratch pad memory to stack.

Figure 6.2: Euler traversal stack usage plot for the bitcnts benchmark
Since Astute can detect direct and indirect recursion in the call graph, it was observed that many applications from the MiBench suite as well as flightstix used recursion – both direct and indirect.

Figure 6.4 shows the distribution of recursive calls between library functions and user defined functions for the applications from the MiBench suite. Both direct and indirect recursion is considered. The figure shows that most of the recursion in the applications is due to library functions. Hence, even though the developer avoids recursion in user-defined functions, recursion creeps in the application through library modules.

Figure 6.4: Distribution of library functions and user defined functions with recursion.
The call graph is shown in figure 6.5. It is analyzed in section 6.2

Figure 6.5 Call graph for bitcnts executable, compiled with no optimization.
6.2 Effects of compiler optimizations on stack depth of the application.

In this experiment five benchmarks from the Mibench suite are used – bitcnts, dijkstra, sha, stringsearch and crc32. The stringsearch benchmark searches for given words in phrases using a case insensitive comparison [43]. The crc32 benchmark performs a 32 bit CRC check on a file [43]. The other benchmarks are described in table 6.1. The sources are compiled with no optimization and with optimization level O3. The change in the maximum stack depth is given by the equation:

\[
\text{Change in maximum stack depth} = \frac{\text{Maximum stack depth with } -\text{O3}}{\text{Maximum stack depth with no optimizations}} 
\]

The change in maximum stack depth for the 5 benchmarks is shown in figure 6.6. It can be observed that in case of bitcnts, dijkstra and stringsearch benchmarks, the maximum stack depth for the application increases when optimization –O3 is used. However, in case of sha and crc32 benchmarks, the maximum stack depth reduces when optimization –O3 is used. Optimizations on bitcnts benchmark are explored to a greater extent in the next experiment.

![Change in Maximum stack depth when optimizations -O3 is used](image)

Figure 6.6: Change in maximum stack depth for the set of benchmarks.
The Bitcnts benchmark was compiled with various compiler optimization flags. The maximum stack depth of the resultant executable is analyzed using Astute. The results are as shown in table 6.3.

Table 6.3: Results of the Compiler optimization experiments.

<table>
<thead>
<tr>
<th>Optimization options used</th>
<th>Maximum stack depth from Astute (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>-static (No optimization options)</td>
<td>192</td>
</tr>
<tr>
<td>-static -O0</td>
<td>192</td>
</tr>
<tr>
<td>-static –O1</td>
<td>452</td>
</tr>
<tr>
<td>-static –O2</td>
<td>452</td>
</tr>
<tr>
<td>-static –O3</td>
<td>452</td>
</tr>
<tr>
<td>-static -mapcs-frame</td>
<td>192</td>
</tr>
<tr>
<td>-static -mno-apcs-frame</td>
<td>192</td>
</tr>
<tr>
<td>-static -finline-functions</td>
<td>192</td>
</tr>
<tr>
<td>-static -O0 -fomit-frame-pointer</td>
<td>192</td>
</tr>
<tr>
<td>-static –O1 -fomit-frame-pointer</td>
<td>192</td>
</tr>
<tr>
<td>-static –O2 -fomit-frame-pointer</td>
<td>192</td>
</tr>
<tr>
<td>-static –O3 -fomit-frame-pointer</td>
<td>192</td>
</tr>
<tr>
<td>-static -fomit-frame-pointer</td>
<td>192</td>
</tr>
<tr>
<td>-static -fforce-mem</td>
<td>200</td>
</tr>
<tr>
<td>-static -fexpensive-optimizations</td>
<td>196</td>
</tr>
<tr>
<td>-static &lt;all the optimizations controlled through –f flag&gt;</td>
<td>220</td>
</tr>
<tr>
<td>-static -Os</td>
<td>440</td>
</tr>
</tbody>
</table>
The un-optimized executable has a maximum stack depth of 192 bytes in case of the bitcnts benchmark, while even with optimization O1 the maximum stack depth increases to 452 bytes.

Optimizations that can be controlled through the command line using the –f flag do not contribute heavily to increase in maximum stack depth. For example, -fexpensive-optimizations contributes to increase of only 4 bytes. The combined use of all the optimizations controlled using –f do not contribute heavily to increase in maximum stack depth. In the case of bitcnts they contribute to 28 bytes increase in stack usage.

Analysis of stack usage of individual functions shows that compiler optimizations result in reduced stack usage for some of the function. This can be seen in figure 6.7. This could be because of optimizations such as function inlining which could reduce the stack usage per function.

![Stack Usage Graph](image)

**Figure 6.7:** The stack usage of functions (whose stack usage changes with optimizations) in bitcnts benchmark. The stack usages of the other functions remain the same.
However, the maximum stack depth for the application increases. This could be because of change in calling sequence or change in the functions called.

The call graph for the un-optimized and optimized version of the executables (figure 6.5 and 6.8 respectively) show that in the optimized function fwrite is used instead of fprintf. The relevant source code is shown below:

```c
fprintf(stderr, "Usage:bitcnts <iterations>\n");
```

The compiler recognizes that the argument “Usage:bitcnts <iterations>\n” is a string literal that does not change at run time, hence the optimized function fwrite is used instead of fprintf in the optimal executable. Since the maximum stack usage of fwrite (determined through euler traversal) is more than that of fprintf, the maximum stack depth of the application increases.

It should be noted that not all compiler optimizations can be controlled through the –f flag; many get turned on by using the –O flag in the command line. Since the optimizations controlled via the –f flag do not result in increasing the stack depth heavily, the other optimizations, which can only be turned on through the –O flag, must be responsible for increase in maximum stack depth of the application. For bitcnts, the increase in maximum stack depth is because of the use of fwrite in place of fprintf. In this case the more “optimal” function (fwrite) results in increased maximum stack depth for the application.

To summarize this section, the stack space analysis of real time embedded system applications reveal interesting results (such as that of recursion) and information which can be used in memory allocation and choosing preemption points for tasks. Compiler optimizations for run time may affect the stack space utilization of an application adversely (as seen in case of bitcnts benchmark).
Figure 6.8: Call graph for the bitcnts executable, compiled with –O3 optimization flag
CONCLUSIONS & FUTURE WORK

The following conclusions can be drawn based on the experiments in this thesis:

- Our methodology and implementation for stack space analysis have been verified to give correct results in run time comparable to that of Stack Analyzer.
- Analysis of maximum stack depth of embedded applications shows that many applications may have large stack space requirements (for example Susan).
- Local minima of stack usage of an embedded application can be determined using the euler tour traversal plot. The local minima points in the plot could be used as preemption points for the tasks in order to optimize the system in terms of stack usage.
- The hotspots in stack usage provide a starting point for allocating the stack usage into a memory hierarchy.
- The embedded application executable should be analyzed for recursion as it is observed that in many cases the recursion in the executable is due to library functions and not user defined functions.
- Compiler optimizations for run-time may affect the stack space utilization of an application adversely as seen in the embedded benchmarks such as bitcnts.

Future work

Improving the run time efficiency of Astute is one of the areas of future work. One of the approaches in this direction could be making Astute a multithreaded application. A thread can be invoked per root node child to compute the maximum stack depth in parallel. The stack information of the individual threads can be accumulated to get the euler traversal plot and static stack depth distribution plot.
Integrating Astute with the rest of the MARTS tools (ARMSAT, OMMA and PTS scheduler) would be beneficial in this area of research. The local minima of stack usage generated from Astute can be used in future work in choosing preemption points. The hot spots in stack usage can be used in future work in allocating the stack into the memory hierarchy to get better run time performance. As a stand-alone tool, a GUI front end for Astute (like that of stack analyzer) would make it easier for the user to use the tool.

Evaluating stack behavior per state and analyzing the dynamic behavior of the stack is another area of future research.

Further research in compiler optimization and their effect on stack utilization is warranted. For systems where memory utilization is of concern while compiler optimizations are still needed, the compiler could be provided feedback from Astute to selectively enable / disable optimizations.
REFERENCES


[4] Procedure Call Standard for ARM architecture, Document number: ARM IHI 0042A


[27] The GNU binary utilities, version 2.14


[29] http://www.gnuplot.info/


[34] http://www.arm.com/products/CPUs/ARM7TDMI.html


[37] Weicker, R.P., An overview of common benchmarks,

[38] http://en.wikipedia.org/wiki/XScale


