Automatic Thread Stack Management for Resource-Constrained Sensor Operating Systems

Adam Torgerson*

University of Colorado, Boulder

Abstract

As low-power microcontrollers for wireless sensor nodes and other embedded devices continue to be manufactured with more memory, multi-threaded operating systems are becoming available on such devices. Each thread must be associated with an independent stack and part of the system memory must be dedicated to these stacks. This paper proposes a method of bounding worst-case thread stack sizes from assembly code by constructing a limited control flow graph and statically examining all possible execution flow within each thread, accounting for any stack space used by non-reentrant interrupts. This technique eliminates any guesswork from determining a thread's stack requirement and allows thread stack sizes to be managed automatically.

*Electronic address: adam.torgerson@colorado.edu; URL: http://mantis.cs.colorado.edu

I. INTRODUCTION

Current sensor networking platforms use microcontrollers such as Atmel's Atmega128, or Texas Instrument's MSP4301611 as the main processor. This class of microcontrollers typically utilize a Harvard architecture, with 4K-10K SRAM. These processors are fast enough and have enough memory that some of those resources can be spent on an efficient threading framework, with enough left over for typical sensor networking applications. One of the main problems in designing such an efficient threading framework is deciding how to handle thread stacks, pools of memory used as execution stacks, each of which must be independent per thread. These execution stacks are swapped depending on the current thread context. In the case of the MANTIS Operating System[1] (MOS), the onus is currently on the programmer to predetermine the stack space necessary for each thread's execution, in addition to any interrupt handler overhead. The typical recommended starting value for stack space has been 128 bytes; this number is passed as a parameter to the thread creation function, and is increased as deemed necessary by the programmer.

The problem with this approach is many programmers do not completely understand how the stack is used, and perhaps more often, do not understand how their statements in a higher level language such as C will relate to the stack space used during execution. Optimizing compilers only confuse the issue further, as local variable combination and elimination, function inlining and numerous other optimizations occur, the relationship of the C statements to the corresponding assembly generated is greatly loosened. Microcontrollers currently used do not have any kind of memory protection, so when one thread's stack grows beyond its allocated region, it potentially moves into another thread's stack, or corrupts some other portion of memory, either case leading to possible errors. When a programmer estimates a thread's stack size, if their estimate is too low, the stacks will overflow and corrupt memory. If their estimate is too high, scarce memory is wasted in unused stack space. Ideally, the programmer would not have to estimate thread stack sizes. A code analysis tool for determine worst-case stack usage would take the guesswork out of allocating stack space.

II. PROBLEM ANALYSIS

The technique of static code analysis is well-suited for the embedded realm[5]. In MOS, for example, users are discouraged from allocating any memory for fear of fragmenting the severely limited resource. Self-modifying code is not directly possible because of the Harvard architecture. Programs are typically small because of code space limitations, allowing for the analysis to finish in shorter time. This is a concern, because static code analysis is an undecidable problem, and the best algorithm can only be implemented in exponential runtime. In order for a programmer to realistically make use of the stack analysis tool, it must not be much slower than the compiler itself, or they will quickly grow tired of waiting for the analysis, and simply not use the tool.

Using static code analysis, a worst-case bound of stack usage can be determined. The technique involves tracing the execution path, following all jumps and calls and taking both sides of all branches, while keeping a running total of the stack usage. If this analysis can be properly bounded, the running program will not use more stack space than the result of the analysis, unless there is some kind of runtime error. In fact, since the analysis is worst-case, most execution paths will not need as much stack space as the analysis yields. Tightening the worst-case analysis to a real-world analysis is a research problem others are approaching with abstract interpretation tools[5, 8].

There are several reasons that stack usage may not be boundable. If interrupts are reentrant, it is difficult to calculate the order they would be received in, although research in this area has been performed[6, 7]. Recursion can also not be handled properly; there is no general way of knowing how many recursive iterations occur without actually executing the code.

In an embedded operating system, much of the time this analysis can be bounded. Interrupts are often non-reentrant, recursion is not typically used, and execution stacks are usually kept minimal. In these cases, static code analysis can be used to obtain a worst-case bound on stack usage. Expanding this idea further, the stack usage of individual threads can also be bounded, and this information can be used to automatically determine the amount of stack space needed for each thread, at compile time. If this can be computed quickly enough, the analysis can be integrated with the operating system's build system, and stack overflow can almost completely be eliminated in most cases.

III. ARCHITECTURE AND ALGORITHM

A. Initialization Required Before Analysis

1. Background

A general tool for static code analysis was not undertaken. The only microcontroller state that is tracked during the analysis is the stack usage at each specific location. The tool was developed in C for speed, and uses the GLib[9] library for portable data structures such as lists and hash tables.

2. Parsing AVR Assembly

An attempt was made at making the implementation portable to other microcontroller assembly languages. Each instruction supported by a particular microcontroller is parsed into an operation data structure, which includes details on what action the instruction is performing. In particular, the types of instructions that are of interest to stack analysis are branches, calls, returns, jumps, loads (from the stack pointer), stores (to the stack pointer), and pushes and pops. Each instruction supported by the Atmega128 is initially inserted into an instruction hash table. As the assembly code of the program being analyzed is parsed, instructions are looked up in this hash table and used to build a list of operation data structures. Potentially, other microcontroller assembly languages could be supported by implementing this instruction hash table for the new assembly language. Comments and assembler directives are completely ignored. Labels are associated with their respective first operation, unless they are strings in which case they are ignored.

The compiler produced two interesting assembly constructs which had to be supported. The first form was quite simple. The target of a branch or jump to .+/-N is the instruction N bytes away from the current location being assembled, either forward (+) or backward (-). The second form was a little more difficult in that it reuses labels, and initially the stack analysis tool relied on labels being unique. Local labels can be defined within a function, and these labels may be reused in other functions. These local labels have the form N:, where N is an integer from 1-9. They are referenced using Nf or Nb. The Nf notation represents the nearest local label N, in the forward direction; Nb in the backward direction. Although none

of the code observed that used these constructs directly modified the stack, functions calls could be made in the execution flow and the stack usage of these calls must be accounted for.

3. Constructing Control Flow

Once the assembly code has been parsed into the abstract operation data structures, a control flow graph can be constructed. The first step in constructing control flow is breaking down the operations into basic blocks. A basic block is a group of linear instructions without any jumps or targets inside. The control flow graph implemented in the stack analysis tool is limited in that back edges are not marked as such and dominator information is not computed. Control flow is established faster by leaving these pieces out, and the analysis tool does not directly analyze loops, so the information is unnecessary.

B. Description of Analysis

1. Stack Analysis Algorithm

The locations of where to begin the analysis are found in one of three ways. Each is found by traversing the list of operations and looking for a set of attributes. First and most obvious, an analysis context starts at the label associated with main(). Each interrupt handler, with the label __vector_N, also begins an analysis context. Lastly, the code preceding a call to mos_thread_new() starts an analysis context as described above.

The stack analysis algorithm is implemented as a recursive routine. Each basic block encountered instantiates a new recursive iteration. First, the basic block currently being analyzed is added to the set of visited basic blocks. The basic block is then analyzed for any instructions which directly modify the stack, and the result of any stack modification found is added to the running total of stack usage. The semantics of how the analysis proceeds at the end of a basic block depends on which kind of instruction is encountered there. For a branch, both the target of the branch and the proceeding instruction are analyzed. For a jump, only the target of the jump is analyzed. For a call, both the target of the call and the next instruction are analyzed, and if the current function frame has not yet encountered a call, the two bytes for the return address are also added to the current stack usage. Any

indirect calls encountered are handled as described above; by analyzing all possible program addresses which have been loaded.

Once each starting point has been established and analyzed, the stack usage of each interrupt handler is checked to find the greatest one. This value is added to the stack usage found for each thread, resulting in the final worst-case stack usage for each thread.

2. Detecting Loops and Recursion

As opposed to performing a full dominator analysis to detect loops, a simple method of keeping track of all previous basic blocks visited in a set, per function scope, is used. Thus, if during the analysis a basic block about to be visited is already in this set, it will skip over this visited basic block and continue on to the next basic block in the execution path instead. This is potentially an issue for hand-written assembly code; it is conceivable that each iteration in a loop could allocate more stack space. Fortunately, the compiler generates code such that all stack operations are in function prologues or epilogues, and these basic blocks are never looped upon.

Recursion is also tracked in a similar manner. Upon analyzing a new function, the function being analyzed itself is added to this set. If a call is made to itself, the analysis will simply skip over this recursion. In both the loop and the recursion case, the best result static code analysis can achieve is to account for the stack space used in each iteration of the loop, or used in each recursive call. Recursion is one case where thread stacks cannot be managed automatically, if recursion is detected, the analysis tool will error out and will not attempt to manage thread stacks.

3. Handling Branches Jumps and Calls

There are three classes of instructions we are interested in when analyzing the execution path: branches, jumps and calls. These are the instructions that modify the program counter without just incrementing it. Analyzing branches and calls are similar; the analysis continues at both the target of the branch or call and the following instruction. In the case of calls, however, the return address is placed on the stack, resulting in 2 bytes of stack usage which are not present in jumps or calls. If there are multiple calls within the same function, this

stack space will only be applied to the analysis once. The target of a jump is also followed, but in this case the analysis does not proceed to the next instruction because execution will not take that path.

The AVR architecture contains four skip instructions in addition to the typical branch instructions. This class of instructions skips the next instruction if some code in the status register is met. While performing the analysis, they are handled as branches, except that the target of the branch is two instructions forward, instead of at the address of a label.

4. Example

A short example C and assembly listing is given below, and analyzed by hand.

```
/* this is a small C example */
int global;

int function2(void)
{
  return 5;
}
void function1(void)
{
  int i;

  for(i = 0; i < 100; i++) { /* line 9 */
    global += function2();

  }
```

```
;this is the resulting assembly file,
;directives and comments removed for
;brevity.  aditionally, basic blocks
;are noted, each basic block is
;surrounded by brackets, []
__SP_H__ = 0x3e
__SP_L__ = 0x3d
function2:
     [ldi r24,lo8(5)
     ldi r25,hi8(5)
     ret]
function1:
     [push r16
     push r17
     push r28
     push r29
     lds r28,global
     lds r29,(global)+1
     ldi r16,lo8(99)
     ldi r17,hi8(99)]
.L6:
     [call function2] ;line 17
     [add r28,r24 ;line 18
     adc r29,r25
     subi r16,lo8(-(-1))
     sbci r17,hi8(-(-1))
     sbrs r17,7 ;line 22]
     [rjmp .L6]
     [sts (global)+1,r29 ;line 24
     sts global,r28
```

```
                                                    pop r29
                                                    pop r28
                                                    pop r17
                                                    pop r16
}                                                   ret]
int main(int argc, char *argv[])                main:
{                                                   [ldi r28,lo8(__stack - 0)
                                                    ldi r29,hi8(__stack - 0)
                                                    out __SP_H__,r29
                                                    out __SP_L__,r28
  function1();                                      call function1]
                                                    [ldi r24,lo8(0)
                                                    ldi r25,hi8(0)
  return 0;                                         jmp exit]
}
```

Figure 1: Example C code and resulting assembly code

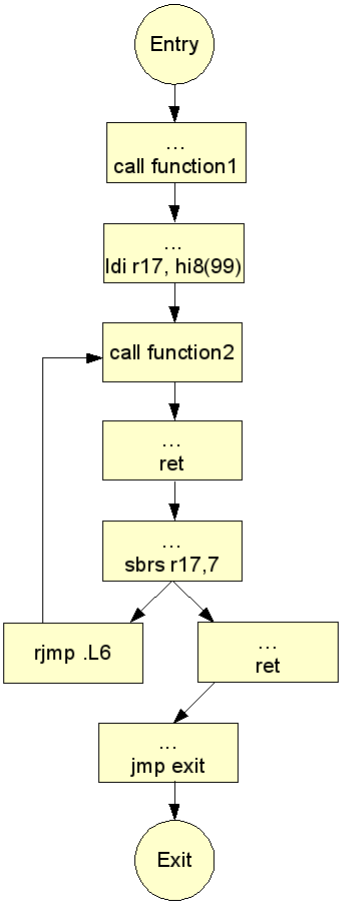

Figure 2: The control flow graph from the code in Figure 1

Although this example does not actually include any threads, they would be handled in a similar fashion, starting a new analysis context at each thread's initial function.

The analysis begins at main(). The initial instructions of main() establish the starting value of the stack pointer, and these initial stores into the stack pointer are ignored by the analysis. Immediately following the stack pointer setup, execution continues to function1(). This adds two bytes of stack usage. The analysis proceeds inside function1(), where it encounters four push instructions, each of which increase stack usage by one byte. Thus, at the basic block starting at label .L6, there are six bytes of stack space used. A call to function2() is encountered at line 17, further increasing stack usage by two bytes. Inside function2(), there are no operations which modify the stack, and the analysis continues back in function1(). The next basic block begins with the add on line 18, and ends with the skip instruction on line 22.

The skip instruction implements the for loop found on line 9 of the C code. Execution continues either at the following rjmp, or one instruction beyond, the sts. Both execution paths are analyzed. Following the jump back to .L6, the call to function2() is once again encountered. However, this time the analysis does not follow the call, because this basic block is a member of the visited set, and has already been analyzed. Finally, this analysis context ends. The other analysis context, still at the basic block starting on line 24, continues, encountering the four pop instructions, and finally, the return back to main(). Once again inside main, the return value is loaded into registers, and a jump to exit is encountered, signifying program termination.

This example has shown a worst-case stack analysis of 8 bytes.

5. Handling Indirect Jumps and Calls

Indirect jumps and calls present somewhat of a problem because the target address is not immediately apparent, it is located in registers whose values are not known until runtime. Instead of analyzing the specific target address in these situations, which would be impossible since it is not known until runtime, a separate hash table of possible target addresses is maintained. Any time an address in program memory is loaded, unless the operation is performed as part of spawning a new thread, the target of that label is inserted into this hash table. When coming across an indirect jump or call, every possible target location is

analyzed.

6. Handling Interrupt Handlers

Interrupt handlers are similar to regular function calls except that the architecture pushes the return address and the status register onto the stack before entering the handler. GCC for the AVR architecture names interrupt handler labels in the form "__vector_N", where N is the associated interrupt vector number, making them easy to locate. Thus, an architecture-specific amount of stack space will be used prior to analyzing the interrupt handler. Since these interrupts are non-reentrant, each interrupt handler can be analyzed in this way and the value from the analysis of the handler with the greatest stack usage requirement is added to the stack requirement of each thread.

7. Finding Threads

In MOS, new threads are created with the function mos_thread_new(). During the analysis, calls to this function are located, and the list of operations is traversed backwards until the function in which the new thread begins execution is found. GCC uses a macro, pm(), to get the address of a label in program memory, and the argument to this macro is the label of the starting function for the new thread. A new analysis context is started for each thread located in the program in this fashion.

8. Handling Compiler Built-ins and Standard Library Functions

There are several functions which a programmer could use which would not be emitted in the final assembly code. These are the C standard library functions, and various compiler built-ins. Some of the standard library functions, such as memset(3) and strlen(3), are completely optimized into simple loops. Others, such as sprintf(3), are not, and must be analyzed. Along with these functions, there are also several compiler built-in functions which are automatically linked into the final executable if they are used. Most of these are math routines which operate on larger datatypes than the 8-bits natively supported by the microcontrollers, or which provide capabilities such as division, for which there is not

a native instruction. To handle this class of functions, each was analyzed once, and the result hard-coded into the stack analysis tool. Thus, if the standard library or compiler is updated, these values will have to be updated as well. Porting the analysis tool to a new microcontroller architecture will also require analyzing the built-ins associated with that architecture.

If the analysis were performed on a complete binary, or the binary disassembled, the exact instructions implementing the standard library and built-in functions would be available, and no hard-coding would be necessary. This, however, would lead to more complex code dealing both with the parsing of binary instructions and establishing the control flow information. The current implementation of the analysis tool deals with only assembly code for reasons of simplicity, and to avoid the tedious and error-prone parsing of binary instructions.

9. Automatic Stack Management

In order to automatically manage thread stacks, each thread is identified and scanned for worst-case stack usage, as described above. Once the worst-case stack usage has been determined for each thread, this value can be inserted as the stack size parameter to the thread creation function. The runtime stack size will not exceed this value, since each possible execution flow has been analyzed.

Once a worst-case stack value has been computed for each thread in the program, the assembly file is searched for those places where a new thread is spawned. At each of these locations, the function which begins the thread is found in the list of basic blocks, and the associated worst-case stack usage of that thread is obtained. The operation list is then traversed backward from the point of the spawned thread, until the instructions loading the stack parameter are found, and the worst-case stack usage value is inserted. This replaces the stack size parameter specified by the programmer,

C. Limitations of the Current Implementation

The assembly parser has only been used on compiler-generated assembly code. Some of the assembly constructs used when hand-writing assembly code may not be properly handled. Most notably, assembly macros and macros generated by the C preprocessor are

not handled. To handle the latter, assembly code may simply be passed through the C preprocessor before beginning the analysis. There is no plan to support assembly macros at this point.

To eliminate the need to resolve references across multiple assembly files, the C files are concatenated. The current build system makes use of static libraries and links these, along with the application object files, to produce the final binary. Thus, the build system will have to be modified to support this concatenation, or the analysis tool will have to be modified to resolve references across multiple files.

Standard functions that allocate stack space, such as alloca(3), are presently not handled, because the AVR backend of the GCC compiler has not implemented them. Potentially, such functions could be handled in a method similar to the other compiler built-ins, as described above. If the value for the space allocated using such functions is stored in a variable instead of a constant, however, it will be much more difficult to handle.

If the stack pointer is referenced in a non-standard way, such as through defining a new name, or using an offset in combination with its memory-mapped address, the analysis will not properly detect the activity. Currently, only the two ways the compiler references the stack pointer (through __SP_H__ and __SP_L__, or the memory mapped addresses 0x3E and 0x3D) are handled.

The handling of indirect calls does not make for a very tight bound on stack usage, since every indirect operation will show the same amount of stack usage. Although there is an idea of how to handle this with a tighter bound presented in the Future Work section, this idea has not been implemented or tested in practice.

The stack check tool does not take global interrupt enable state into consideration, either. Potentially, a section of code could disable interrupts, then use some stack space. In such a case, the greatest stack using interrupt handler would never fire because interrupts are disabled, and this stack usage value should not be added to the total. Global interrupt state is very difficult to statically track if the interrupt state is modified using a variable instead of writing constant values to the status register, which in practice it often is. This will lead to an even worse-case analysis than would be required, but is unavoidable with the current implementation.

IV. EVALUATION

A. Analyzing Typical MOS Applications

Four applications were tested for stack usage. The first, blink_led, blinks three different LEDs in three different threads. radio_test is a test application for evaluating radio performance, with only a single thread. The bedrest_sense and bedrest_relay applications represent typical sensor networking activity: the sense program operates a sensor and sends the data to the relay. Results are given in the table below:

App. Name	main()	preStart()	Thread 1	Thread 2	Thread 3	check()	vec15	vec8	vec12	vec17	vec21
blink_led	63	62	68	68	68	103	56	56	56	N/A	N/A
radio_test	70	69	125	N/A	N/A	110	56	56	56	63	30
bedrest_sense	70	69	134	125	N/A	110	56	56	56	63	30
bedrest_relay	70	69	114	N/A	N/A	110	56	56	56	63	30

Figure 3: Predicted worst-case stack usage (bytes)

In bootstrapping MOS, main() spawns a thread for initialization, preStart(), in order to provide a thread context and start the scheduler prior to running the initialization routines. The different applications each spawn a different number of threads, with blink_led spawning the most: one for each LED to blink. Also of note, the check() thread is used for computing actual stack usage, described below. Each of these test applications used the check() thread, but a normal application would not.

The three interrupt handlers with the same worst-case stack usage, __vector_8, __vector_12 and __vector_17 are all used by the kernel to provide context switching. Two separate hardware timers, whose interrupt routines are __vector_12 and __vector_17, provide pre-emptive context switching. __vector_8 is a software interrupt routine used internally by the kernel. Each of these routines pushes 33 bytes for the context switch itself, in addition to the other stack space needed for additional computation.

The radio driver's interrupt handler is located at __vector_17. It is quite a large piece of code, performing any necessary MAC backoffs, CRC checking of incoming data and buffering of packets. __vector_21 is the interrupt handler for the analog to digital converter, which is fairly simple.

B. Reducing the Impact of Interrupt Handler Stack Usage

As shown from these results, the interrupt handler with the most stack usage is __vector_17, the radio driver's interrupt handler, having a worst-case stack bounding of 63 bytes, which every thread must have room for. Since MOS does not use reentrant interrupts, this space can be reclaimed from each thread, and put into a single space, possibly the idle thread stack, or a separate space explicitly for this purpose. For simple programs, this space will account for over half of the necessary thread stack space, so the memory savings in implementing this scheme will be substantial.

Preemptive multithreaded systems typically use a hardware timer as a timeslice for context switching. MOS actually uses two hardware timers, one for normal operation and another to provide a low-power sleep mode. Context switches occur in both of these interrupt handlers. All registers and the status register are pushed onto the current thread's stack, alone using 33 bytes of stack space. Each thread will need this additional space to support saving their state.

C. Comparing Worst-Case Analysis to Actual Use

Using the same set of MOS tests on which the stack analysis was performed, the actual stack usage of the running applications was determined. The MOS memory allocator inserts a flag into all allocated blocks with the byte 0xEF. The stack of each thread is checked for this byte from the lowest memory address to the highest, opposite the order in which the stack grows, giving the number of bytes in the stack which have not been written with a different value. While admittedly not a perfect solution, we can be reasonably certain the stack did not progress beyond this bound of free space.

The following table compares observed stack usage to the worst-case bound calculated in the analysis:

Application Name	Thread 1	Thread 2	Thread 3	check()
blink_led observed	57	57	57	76
blink_ led calculated	*68*	*68*	*68*	*103*
radio_test observed	94	N/A	N/A	79
radio_ test calculated	*125*	*N/A*	*N/A*	*110*
bedrest_sense observed	98	86	N/A	79
bedrest_ sense_ calculated	*134*	*125*	*N/A*	*110*
bedrest_relay observed	89	N/A	N/A	79
bedrest_ relay calculated	*114*	*N/A*	*N/A*	*110*

Figure 4: Observed stack usage (bytes)

D. Execution Time Analysis

The analysis tool is extremely fast. Compiling the radio_test application took 0.4s. The table below gives execution time of the analysis tool for each analyzed program:

Application Name	Execution Time
blink_led	0.014
radio_test	0.030
bedrest_sense	0.064
bedrest_relay	0.029

Figure 5: Execution time for code analysis (seconds)

Clearly this tool is fast enough to be integrated into MOS' build system without causing inconvenience to programmers.

V. RELATED WORK

Several tools currently exist for analyzing stack usage, although none of the tools investigated supported analyzing stack usage of specific threads, nor did they attempt to automatically manage stacks. One commercial tool for stack analysis is StackAnalyzer[13],

which works across several architectures and displays visual control flow graphs. Such an application would not be possible to modify, and it would not be feasible to integrate into the build system of an Open Source project such as MOS. Avrora[14] offers a complete simulation and analysis framework for AVR microcontrollers. Unfortunately it is a research-grade piece of software, and all of the versions available simply crashed when trying to analyze the stack usage of even the most simple programs.

The tool closest to addressing the problem is stacktool[15], which drives the analysis with an abstract interpretation framework. Unfortunately, abstract interpretation is slow, as portions of the analysis are simulated by analyzing the results of instructions in the bitwise domain. This results in unacceptable runtimes for automatic thread stack management, upwards of 8 seconds per program analyzed. This is significantly slower than the compiler, and while the results give a tighter bound on stack usage, developers would not consistently use a tool so slow. Such a tool is better suited for pre-deployment code optimization rather than general purpose development.

VI. FUTURE WORK

The worst-case analysis is not ideal. Although the guesswork in determining necessary thread stack space has been eliminated, memory is still wasted, as shown above, because some of the cases analyzed will simply not be possible to reach during program execution, because they deal with error conditions never reached, or the event which triggers that code never happens. Even more often, running one piece of code often excludes running another piece; either one or the other is executed and the analysis described here will account for the stack usage of both. The stack space determined to be needed by the analysis of such impossible runtime conditions never actually get used during program execution, and this portion of stack space will be unused, yet still allocated, memory. The maximum stack usage needs to be tightly bounded in order to conserve the most memory. Abstract interpretation is a good way to tighten this bound, but unfortunately it is currently too slow to be practically applied to the problem.

Other sensor operating systems do exist, such as TinyOS[10] and Contiki[11], and it would be interesting to analyze and compare their stack requirements to that of MOS'. TinyOS is written in the nesC language, which is compiled to C, making the analysis potentially easy.

Contiki uses coroutines, implemented in a manner similar to Duff's Device[12], which may make detecting and handling context switches more difficult.

One idea to tighten the worst-case stack usage bound is to use a configuration file to allow the programmer to specify certain situations which are difficult to statically analyze. In the case of indirect calls, the programmer could associate the possible target locations of each indirect call in this file, and the analysis tool would only follow those locations the programmer associated with the call. In a similar manner, the programmer could specify an upper-bound on the number of iterations in a recursive call. This would allow the analysis tool to apply the amount of stack usage found in a single iteration over the total number of iterations specified in the file, allowing the tool to more properly handle recursion. Although application developers may not want to specify such a configuration file for each application, operating system developers could easily do this once for common system calls, enabling a tighter bound.

VII. CONCLUSIONS

This paper has presented a tool for computing worst-case stack usage through static code analysis. The results of this analysis are applied to spawned threads in an application, alleviating the programmer from needing to haphazardly guess necessary thread stack sizes. As shown, the tool is quite fast; fast enough to integrate with an operating system's build system in order to provide automatic thread stack management across multiple threads. Applied to real sensor networking programs, the results of the analysis never exceeded the observed stack usage. The tool has identified interrupt handler stack usage as a large contributor to overall stack usage and proposed a method of eliminating the need for each thread stack to maintain stack space for interrupt handlers. Programmers using a tool such as the one described here for automatic stack management no longer need to worry about exceeding allocated stack space in embedded systems such as MOS.

Acknowledgments

The author would like to thank Professor Han for his contributions as advisor, and Brian Shucker, for the idea of using the analysis to automatically manage thread stacks.

---

[1] H. Abrach, S. Bhatti, J. Carlson, H. Dai, J. Rose, A. Sheth, B. Shucker, J. Deng, R. Han. *MANTIS: System Support for MultimodAl NeTworks of In-situ Sensors.* 2nd ACM International Workshop on Wireless Sensor Networks and Applications (WSNA), pp. 50-59. 2003.

[2] Cullen Linn, Saumya Debray, Gregory Andrews, Benjamin Schwarz. *Stack Analysis of x86 Executables.* Manuscript. April, 2004.

[3] Leena Unnikrishnan, Scott D. Stoller, Yanhong A. Lui. *Automatic Accurate Stack Space and Heap Space Analysis for High-Level Languages.* Lecture Notes in Computer Science Volume 1474, pp. 31-40. 1998.

[4] Dirk Grunwald, Richard Neves. *Whole-Program Optimization for Time and Space Efficient Threads.* Architectural Support for Programming Languages and Operating Systems, pp. 50-59. Cambridge, MA, October 1996.

[5] John Regehr, Alastair Reid, Kirk Webb. *Eliminating stack overflow by abstract interpretation.* Proceedings of the Third International Conference on Embedded Software, pp. 306-322. Philadelphia, PA, October 2003.

[6] Krishnendu Chatterjee, Di Ma, Rupak Majumdar, Tian Zhao, Thomas A. Henzinger, Jens Palsberg. *Stack Size Analysis for Interrupt-Driven Programs.* Proceedings of the Tenth International Static Analysis Symposium, Lecture Notes in Computer Science Volume 2694, pp109-126. Springer-Verlag, 2003.

[7] Dennis Brylow, Niels Damgaard, Jens Palsperg. *Static Checking of Interrupt-driven Software.* Proceedings of the 23rd International Conference on Software Engineering, pp. 47-56. Ontario, Canada, 2001.

[8] Patrick Cousot, Radhia Cousot. *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximations of Fixpoints.* Conference Record of the 4th ACM Symposium on Principles of Programming Languages. Los Angeles, CA, January 1977.

[9] GLib, http://www.gtk.org

[10] TinyOS. http://www.tinyos.net

[11] Adam Dunkels, Björn Grönvall, and Thiemo Voigt. *Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors*. In Proceedings of the First IEEE Workshop on Embedded Networked Sensors 2004 (IEEE EmNetS-I), Tampa, Florida, USA, November 2004.

[12] "Tom Duff on Duff's Device". http://www.lysator.liu.se/c/duffs-device.html, Accessed in March, 2005.

[13] "StackAnalyzer - Stack Usage Analysis". http://www.absint.com/stackanalyzer

[14] "Avrora - The AVR Simulation and Analysis Framework". http://compilers.cs.ucla.edu/avrora

[15] "Using Static Analysis to Bound Stack Depth". http://www.cs.utah.edu/~regehr/stacktool