Efficient Stack Sizing for Very Large Software Systems

D.T.V. Ramakrishna Rao
(ramakrishnadtv@infosys.com)
Infosys Technologies Ltd.,
Electronics City, Bangalore - 560 100, India.

Abstract—Accurate determination of stack sizes for tasks is essential for software systems whose tasks have predeclared stack sizes (for example, embedded systems). Underestimation of the stack size can lead to stack overflows, a particularly insidious run time error. Overestimation will lead to wastage of precious memory space of these often memory starved systems. In this paper, we describe a tool which efficiently computes the stack sizes for tasks of very large software systems supporting multiple products. This is in contrast to existing tools which are mainly targeted at small software systems supporting single products. The efficiency of the tool is achieved by using two different types of caches.

Index Terms—Stack overflow, tool, static analysis, flow graphs, caches, efficiency, recursion, stack size, memory optimization.

I. INTRODUCTION

The run time components of a task (sometimes referred to as “thread”) include text and stack. One of the design issues associated with a stack is its size. The tasks running on general purpose operating systems like Unix do not have to contend with this issue. The operating system extends their stacks on a need basis. But for other systems, especially embedded systems, no such luxury exists. Programmer has to fix the stack sizes of the tasks.

Finding the stack size of a task is not trivial. In general, programmers fix the stack size of a task in a nonscientific and ad hoc basis; often it is a guess. If the guess underestimates the stack size possible in practice, that generally results in memory corruption. Because memory corruption is a particularly insidious bug to fix, programmers tend to overestimate the stack size. On these systems, memory is not a cheap resource. Overestimation of stack size results in wasted memory space which could have been used for other purposes.

It is clear that an accurate estimation of stack size is important for these systems. It is not only embedded systems which benefit from an accurate estimation of stack size but also a number of other systems whose tasks run under fixed stack sizes (e.g. routers).

This paper makes two contributions related to stack sizing. First, it surveys, compares, and evaluates the existing approaches to stack sizing. Second, it describes a tool that statically finds the stack size used by a task. The tool is primarily targeted for very large software systems.

In this paper, for concreteness, we consider stack sizing for programs written in C/C++ languages. The ideas and approaches, however, are not tied to these languages and can be easily carried over to programs written in other languages.

The rest of the paper is organized as follows. In Section II, we critically analyze the existing approaches for stack sizing. In Section III, we describe our tool in depth. We conclude in Section IV.

II. ANALYSIS OF EXISTING APPROACHES

There are two approaches to do stack sizing: run time approach and static approach. Run time approaches find out the stack size of a task using run time information. Static approaches find the stack size of a task at compile time.

A. Runtime Approaches

There are two ways to find out the stack size of a task using run time information.

1) Fill the Stack with a Pattern: We initially arbitrarily decide the stack size of a task. At run time, before the task starts running, we fill the entire stack with a repeating pattern like “0x55aa” [1]. Then the task is set in motion. After running the system as per its typical usage, we check the stack. By the amount of pattern that remains undisturbed, we can make out the stack usage of the task. After adding a margin, the stack size of the task is decided.

A disadvantage with this approach is it does not allow us to reduce the stack usage. One of the main reasons for high stack usage is the use of big structures or large arrays as local variables in functions [2]. The ideal solution in that case is to move the structures from static to dynamic allocation. But in this approach, we do not have the stack trace that caused us to see the longest stack depth – all that we know is in some execution sequence this much stack is used. So, the only alternative left is to increase the stack of the task up to the maximum stack usage seen in practice.

2) Compiler Stack Overflow Check: After deciding the stack size (by whatever means), we enable the compiler to generate stack overflow checks in the code. Many compilers support options for doing so (for example, fstack-check option of gcc [3]). If the option is enabled, the compiler adds a check in the prologue...
of every function to ensure that stack usage is within the set limit. If stack overflow happens, generally an exception is raised.

One advantage of this approach is when stack overflow happens, we will have the trace of the functions which caused the stack overflow. We can then analyze the trace. If the stack overflow is due to usage of huge local variables on stack, we can convert them to dynamic allocation. It is even possible that the specific trace which caused stack overflow is due to a bug [2] and so is not a valid trace. In that case, we can simply fix the bug rather than increasing the stack size. So, with this approach, unlike the previous approach, we are not forced to increase stack size whenever stack overflow happens.

This method has certain disadvantages. Since compiler adds additional code for each function, both run time overhead and code size increase.

B. Static Approaches

Unlike run time approaches, which estimate stack size based on runtime information, static approaches estimate stack size without running the program. By analyzing the call flow, static approaches find the maximum stack size used by a task.

It is not possible to come with a stack size that may be used by a task from the high-level source code alone. For precision in estimating stack size, static approaches use disassembled code due to the following reasons:

- The amount of stack space a function uses depends upon the processor that will run the code.
- The amount of stack space a function uses depends upon the compiler.
- Many software systems have embedded assembly code.
- The libraries used by the software may be available only in binary.
- Analysis of assembly code is easier compared to high level source code.

1) Flow Based Analysis: Using the assembly language code, static approaches build the control flow graph (CFG). A CFG models the relationship between assembly statements. A node in the CFG represents an assembly statement. There will be an edge from node \( n_1 \) to node \( n_2 \) if it is possible to execute instruction corresponding to \( n_2 \) immediately after that of \( n_1 \). The weight of a node is the stack size of the instruction associated with that node. By depth-first search of the graph, it is possible to come up with the stack size that may be used by a task [4].

Stlite [5] is a Perl-based tool which does flow based analysis to find out the stack size for embedded systems. It is applied to programs which have up to 1100 lines of disassembled lines of code for Amtel chips. In Java virtual machine [6], an intra procedural stack-depth analysis is done to ensure that the amount of stack space used by the byte code is within stack size limits. StackAnalyzer [7], a commercial product from AbsInt, estimates the stack usage for an application and provides feedback for optimizing the stack usage.

2) Interrupt Support: The flow-based stack depth analysis sketched above did not consider the possibility of interrupts. Apart from tasks, interrupts also involve function calls which occupy stack space. An interrupt service handler invokes a function which in turn may invoke other functions. All these functions occupy stack space. The stack space used by this set of function calls is referred to as “interrupt stack”. If a system has multiple running tasks and if an interrupt is raised, which stack the interrupt uses? There are two answers depending on the type of processor: (1) The interrupt can use the stack of the currently running task; (2) The interrupt can use a separate interrupt stack.

Some processors mandate that only one of the above approaches can be used (for example, i960jx processor uses a dedicated interrupt stack [8]). Others give the flexibility to the system designer (for example, MIPS R5000 [9]).

If a system has multiple tasks and if we use the first approach above then every task should be prepared to host the interrupt stack also. This can considerably increase the memory requirements of the system depending on the number of tasks it has and the interrupt stack depth. So the second approach is better and should be used whenever possible. If a system uses the second approach, the flow-based analysis sketched in Section II-B.1 can be used for the system’s tasks. A separate analysis is needed to find out the stack size of interrupt stack.

Many small embedded systems have a single task. For such systems, where memory space at premium, it is better to use the first approach. But this leaves a difficult question: How do we estimate the stack space needed for such a stack?

Brylow, Damgaard, and Palsberg [4] enhanced the flow analysis to take interrupts into consideration. Unlike the above flow-based analysis, where a node models only a program counter, their analysis models interrupt mask register also. They focused on relatively small programs (around 1000 lines) of assembly code. Regehr, Reid, and Webb [10] extended the stack size analysis of [4]. They tested their tool on embedded applications compiled from up to 30000 lines of C code.

Chatterjee et al. [11] studied two related problems: (1) Is the stack size of a program bounded at all? (2) What is the maximum stack usage of a program for which a function trace exists? They determined the time complexity and provided algorithms for these problems for a restricted class of interrupt-driven programs.

3) Variations on Static Analysis: Avrora [12] research project has a stack checker tool, which performs...
a context sensitive analysis to determine a bound for the stack size of a task. Unnikrishnan, Stoller, and Liu [13] studied the stack usage analysis for first order functional languages. Their analysis works directly on source code rather than assembly code.

C. Comparison of Run time and Static Approaches

Because run time approaches depend on the specific tests we run, we might not have seen the stack trace with maximum stack size during execution. In that case, if the margin added is not enough, stack may overflow during some other execution. On the other hand, if we add a large margin, we might waste precious memory. To improve run time analysis, we need to add more test cases to test as many paths as possible. But there are practical limitations for testing due to the considerable number of resources needed.

Static analyses, in contrast, analyze entire code. They consider all possible paths and all possible inputs. So, they bound the stack size much better. Often there is no need to do add any margin. (We might want to add margin considering the evolution of a program.) Static analyses are also faster compared to run time approaches. Like compiler stack overflow check, static analyses give the longest stack trace which can be used for optimizing stack space. In general, it is better to opt for static analysis compared to run time analysis.

Static analysis, however, is not without its share of limitations:

- **Impossible Traces:** The maximum stack size reported may correspond to a function trace which can not happen in practice. This can happen due to dead code. But this can also happen without dead code – not all function traces which are possible in principle are possible in practice [14].
- **Recursion:** Since static analysis can not find the maximum depth of a recursive function, it can not estimate the stack size contributed by recursive functions.
- **Function pointers:** Since static analysis does not know which function is called, it can not take that part of the flow graph into account for sizing the stack.
- **Indirect Jumps:** This is a jump to an address in a register. Since static analysis does not know the branch target, it can not take that part of the flow graph into account for sizing the stack.
- **Flexible stack size for a frame:** Some features like alloca and local arrays with non-fixed sizes make it difficult for static analysis to estimate the size of stack frame of a function [10].

The existence of these limitations do not mean that the results from static analysis tool will not be useful. The results should be taken with the understanding that the tool has not taken these aspects into consideration. It would be a good idea if the tool reports those aspects which it failed to take into account so that user may take necessary action. These aspects affect the tool in different ways. For example, due to dead code the tool may overestimate the stack size of a task. On the other hand, due to function pointers, alloca, recursion, and indirect jumps, the tool may underestimate the stack size. These limitations of static analysis can be alleviated in a number of ways:

- **Understanding Compiler code generation:** Majority of the assembly code that is analyzed by static analysis tools is produced by compilers. Compilers produce highly stylistic code. They generally do not use the full generality of assembly instruction set. For example, C’s switch statement may use indirect jumps but uses them in a specific form of jump tables [15]. If the tool understands how compiler generated the code, it will be able to analyze such indirect jumps also. So, by understanding how a compiler generates code, some of the limitations of static analysis can be avoided in practice. An even better approach is integrating static analysis with the compiler itself [16][17].

- **Using Compiler Techniques:** Some of the limitations are arising because a static analysis tool does not have the needed information to build the complete annotated flow graph. For example, in an indirect jump it does not know where the jump is going; it does not know how much stack size alloca allocates; it does not know how much depth a recursive function may go. If the tool can extract this information from the code, it can proceed further. Compilers do this all the time during their optimization phase. Techniques like constant propagation, constant folding, etc [15] can be used by the tool to extract the needed information. For example, Blieberger and Lieger [18] provided algorithms to derive the upper bound on the stack space usage of recursive procedures that meet certain conditions.

Although the limitations are alleviated by these techniques, they can not be eliminated in all their generality – certain undecidability results from theoretical computer science are associated with the limitations[10].

Due to the limitations of static analysis, a combination of run time approach and static analysis can play an important role. We find the stack size of a task using static analysis and enable compiler stack overflow checking. If static analysis has underestimated stack size, that will be detected at run time by the compiler generated check. This helps not only in tackling the limitations of static analysis but also in other ways. Software is not static; it gets changed, new features get added, and bugs get fixed. So, stack usage of a task changes over time. Hence, irrespective of the method we use to find out the stack size of a task, it is generally a good idea to enable stack overflow
III. STACK SIZING TOOL

The initial impetus for our tool came from the need for stack sizing on data communication software systems. Each of these software systems, typically written in C/C++, serves as a code base for multiple networking products. They use a diverse range of processors including P960, MPC860, and MIPS R5000. Their code bases are generally huge (tens of thousands of files and millions of lines of code are common).

These systems are structured with many cooperating tasks. Each task has a fixed stack size which is specified by the programmer at the time of task creation. Stack overflow is a concern on these systems because of which they employ the two approaches discussed in Section II-A (filling the stack with a pattern and compiler stack overflow checking). Although these approaches help in stack overflow detection, they do not avoid the possibility of stack overflow happening in the customer setups. Indeed, a considerable number of stack overflows were noticed in customer setups. Clearly, there is a necessity for managing the stack sizing of tasks in a better manner. That necessity provided the motivation for our stack sizing tool. Although it might seem that the tool is tied to data communication software systems, it is not. It is general purpose in nature.

In the rest of the section, we discuss the design and development of the tool. Instead of describing requirements, design and development in sequence, we describe the tool in phases for easier exposition. It also places the requirements of the tool and their motivation in proper context.

A. Phase 1: Basic Tool

When we considered stack sizing for the tasks in data communication software systems, we considered the existing tools described before. On examination, we found that they are not appropriate for our kind of software systems.

Most of the existing tools model each assembly statement as a node in the flow graph. This is fine for the small software systems for which those tools are targeted. But for very large software systems, the number of nodes will be humongous. The time consumed by such a fine-grained analysis will be prohibitive in practice.

Therefore, it is clear that a new tool is required for very large software systems. To keep the size of flow graph moderate, we model a function as a node in flow graph. We represent the task as a call flow graph where the edges represent calling relationships between functions.

Before building the call graph, we need to define the notion of individual stack size:

Definition: Individual Stack Size of a function is the maximum stack size occupied by that function alone considering all possible execution sequences originating from that function.

To build the call-graph, we need two types of information for every function $f$ in the software system: (1) The individual stack size of $f$; (2) The list of functions called by $f$. We extract this information from the assembly code of the software system.

But we have a problem here. As indicated earlier, data communications software systems use different types of processors for different products. As these systems evolve, it is not unusual to use new processors for the new products which come along. It is important that the stack sizing tool evolves smoothly given that. If the tool directly processes assembly code then it needs considerable rework while porting to a different processor (for instance, the existing tools suffer from this issue).

Since stack sizing is done in the context of a specific product, which uses a specific processor, naturally the tool will be tied to that specific processor. But we intend to do stack sizing for all the products. Instead of aiming at a single tool for all products, we aim at easier porting of the tool from one processor to another. Accordingly, the tool is structured as a combination of front end and back end as shown in Fig. 1.

The entire processing of assembly code resides in the front end. The results of front end are updated in the flow database. The database has the function stack size information and call relationships for the entire system in processor independent form. The back-end uses the flow database to find the stack size of a user supplied task. When the tool needs to be ported to a new processor, a new front end for that processor needs to be written; back end remains unchanged. We are basically using the same approach that is used by compilers to generate code for a variety of processors; intermediate code [15] serves as the link between a compiler’s front end and back end.

1) Front end: Our initial front end is targeted for MIPS [9] assembly based processors. To use the front end for a software system, its Makefile should be modified to generate the assembly file for each source file compiled. Each assembly file will be processed by an AWK [19] script to generate the flow database for that file. The flow databases of all the files are concatenated to form the flow database for the system.

The AWK script understands the MIPS instruction set [9] related to changing stack pointer (e.g., “subu $sp,$sp,24”) and calling functions (e.g., “jal stcpy”).
The script analyzes these instructions for each function in a file. By analyzing the stack pointer manipulation instructions, the script calculates the individual stack sizes of the functions. By analyzing the jump instructions, the script finds out the functions called.

Compilers follow one of two approaches with regard to stack manipulation, partly determined by Application Binary interfaces (ABI) [16]. In the first approach (incremental approach), the compiler generates the stack size manipulation instructions on need basis. For example, if a function has two local variables, the compiler may generate stack changing code for these local variables. If subsequently the function has an inner block with some local variables, compiler generates stack changing code for them again. So, there can be multiple instructions in a function which increment stack pointer.

In the second approach (lumped approach), the compiler does not generate multiple instructions for incrementing stack pointer for a function. The compiler analyzes the entire function and finds out the maximum stack size used by the function in all its possible execution paths. Then it generates a single instruction in the function prologue which decides the individual stack size for the function.

In the lumped approach, a function upfront uses the maximum possible stack frame it might ever need. In the incremental approach, a function’s stack frame grows or shrinks on a need basis. So, the incremental approach generally results in lesser overall stack usage of tasks. The recommendation of programming “Declare variables near their usage”, an aid to improve code readability, is especially valuable in this regard in reducing stack usage when compilers that adopt incremental approach are used. But the incremental approach achieves reduction in stack usage at the expense of increased performance overhead and increased code size (as it generally requires multiple stack incrementing instructions compared to the lumped approach).

The front end processes the assembly code generated by GCC version 2.91.22 [3], which uses lumped approach. This results in simplification of the AWK script. The AWK script looks for the stack incrementing instruction in a function prologue to find out the individual stack size of the function. If we need to enhance the AWK script for compilers of incremental approach, we need to use the same algorithm used by the lumped approach compilers to find out the individual stack size of a function.

A snippet of flow database generated by the AWK script is shown in Fig. 2. The snippet is for the function `tmclear()` (function names are in mangled form), which has an individual stack size of 32 bytes. It calls 3 functions: `signal()`, `freemem()`, and `printf()`.

While the script is going through the assembly code, it may encounter two situations which it can not handle: (1) Stack manipulating instructions where the stack pointer adjustment is not statically known. This is generally due to `alloca()` or local array variables whose size is known at run time only; (2) Function calling instructions involving indirect calls; this is generally due to function pointers. The script informs these events to user through a log file meant for this purpose. The user, based on his knowledge, may update the flow database for the events noted by the tool. For example, if function pointers are involved and if user knows the functions which are called through that function pointer, he may directly update the flow database. Similarly, he can update the individual stack size of functions for `alloca()` calls. This approach is similar to annotations provided by the users for some of the existing stack sizing tools.

2) Back end: By using the flow database, back end finds the maximum stack usage of user input task. Back end does not refer to source code at all. In phase 1 of the tool, we restrict our attention to tasks which do not use recursion. We relax this restriction in phase 3.

Although we are interested in the stack sizes of tasks only, it helps if we generalize the concept. A task basically represents execution of a specific function. We design our tool so that we can find out the maximum stack usage of any function. Then the maximum stack size of a task is nothing but the stack size of the function associated with the task.

We need to formalize the notion of maximum stack usage of a function:

**Definition: Accumulated Stack Size** of a function \( f \) is the maximum size of space occupied by the stack with \( f \) as the first function and considering all possible execution sequences originating from \( f \). The function trace which corresponds to accumulated stack size of a function is referred to as its Accumulated Stack Trace.

Suppose function \( f \) calls functions \( g_1, g_2, \ldots, g_n \), then

\[
\text{Acc}(f) = \text{Ind}(f) + \max(\text{Acc}(g_1), \text{Acc}(g_2), \ldots, \text{Acc}(g_n))
\]

where \( \text{Acc}(f) \) stands for accumulated stack size of \( f \), \( \text{Ind}(f) \) stands for individual stack size of \( f \), and \( \max \) gives the maximum of a list of numbers. Equation (1) serves as the basis for our stack sizing tool. Although the primary intention of the tool is to find out the

---

Fig. 2. A snippet of flow database
(stackSize, stackTrace) findStackSize(file, fn)
{
    (definedFile, fnSize, calledFnList) ←
    findFuncInfo(file, fn)
    if (definedFile, fn) exists in cache
        return (stackSize, stackTrace) from cache
    currentSize = 0; currentTrace = {}
    for each function in the calledFnList
        (calledFnSize, tempTrace) ←
        findStackSize(definedFile, calledFn)
        if (calledFnSize > currentSize)
            currentSize = calledFnSize
            currentTrace = tempTrace
        prepend (definedFile, fn) to currentTrace
        store ((definedFile, fn), fnSize + currentSize, stackTrace) in cache
    return (fnSize + currentSize, stackTrace)
}

Fig. 3. Pseudo code for Phase 1 and Phase 2 of the tool

accumulated stack usage of a task, in practice it helps to provide the accumulated stack trace as well. Apart from helping in potentially reducing stack usage, it also serves as a debugging aid during the development of the tool. The tool takes as input a function name and the file in which it is referenced. Using flow of the tool, the tool outputs the accumulated stack size and the file in which it is referenced. Using flow database, the tool outputs the accumulated stack usage and accumulated stack trace of the function.

The pseudo code for the heart of the tool appears in Fig. 3 (ignore the code in italics²). The code is fairly self-explanatory. findStackSize() computes accumulated stack size (first return value) and accumulated stack trace (second return value) for the task corresponding to fn which is referred in file. It might be surprising why the file gives the file in which fn is referenced and not the file in which fn is defined (there are two ways a file may reference a function: either the function is defined in the file or the function is called in the file). After all, when users of the tool are giving the name of the function associated with the task they are interested in, they can as well give the name of the file in which fn is defined rather than referenced. This is a perfectly valid scenario and is the normal way in which the tool is used. But we make the interface as it is to take care of the recursive nature of findStackSize(). While recursively calling findStackSize(), we call it with called functions of fn. Suppose g is a called function of fn. Then when we want to call findStackSize() on g, we know only the file in which g is called, we do not know the file in which g is defined.

Given (file, fn), findFuncInfo() returns definedFile, the file in which fn is defined; fnSize, the individual stack size of fn; calledFnList, the list of functions called by fn.

An example accumulated stack trace generated by the tool for a task is shown in Fig. 4 (not shown fully). The first line gives the accumulated stack size of _diffTask as 1256. The trace, which shows the order in which functions are called, gives the split-up of the accumulated stack size in terms of individual stack sizes. For each function in the trace, name, file name, and the individual stack size are included. The trace is useful in finding out if there are any unusual hoggers of stack space. For example, user might want to check the functions _timeoutHandler() and printf().

3) Evaluation: We ran the tool, written in C, on a set of tasks chosen from a data communications software system (let us refer it as “DC”). The tool ran on a Linux (Redhat Version 9.0) based system with Pentium P4 processor running at 2.6 MHz. The results are shown in Table I. Second column gives the time taken by the tool for finding the accumulated stack size of the tasks. The time reported is the real time elapsed for the tool from start to finish.

The time for the tool to find out the stack size is in the order of minutes. We ran the tool on those tasks which are declared with a “small” stack size (2 kilo bytes). It can be easily understood that if we run the tool on tasks with “very large” stack (20 kilo bytes) (there are many such tasks in the system), the time would be very high.

We have chosen to model the flow graph with a node represented by a function rather than an assembly statement to reduce the time consumption of the tool. The results show that even with this modelling, the
time consumption is high and there is a need to improve the tool further.

B. Phase 2: Cache for Nonrecursive Functions

To make the tool faster, we analyzed the tool in action. We applied the tool to find the stack size of a lot of tasks and analyzed it by tracing all the functions whose stack size it was computing. We made two observations.

First, the tool was repeatedly computing the stack sizes of certain functions. For example, it was computing the stack size of printf function a large number of times. That is because printf is called in almost all the functions. When we realize that printf is a complex function involving a large call graph, we can understand how the tool is spending its time. This observation is not specific to printf but to most of the functions in libraries like logging functions, string library, pools library (malloc and free), semaphore library, container library (queues, lists, etc), and timer library.

Second, this phenomenon of repeated stack size computation is not specific to library functions. We noticed that some tasks are sharing significant amount of call graphs. For example, multiple tasks are receiving packets and processing them. It is unusual to see multiple tasks doing similar jobs when one of the principles in structuring tasks is that each task should do a specific and single job. One of the reasons for the discrepancy is the need to attain better performance; different tasks sometimes end up doing similar processing to avoid context switching overhead.

Based on these observations, we have decided to create a cache to keep track of accumulated stack size and accumulated stack trace of functions. The caching design is based on (1). With respect to cache design, we are interested in two main aspects: (1) How and when a cache entry is created? (2) How and when a cache entry is referenced?

We augmented the Phase-1 tool with caching (shown in italics) in the pseudo-code (see Fig. 3). When we intent to find the accumulated stack size of a function, before we analyze the called functions, we first check whether the function already exists in cache. If so, we return the accumulated stack size and accumulated stack trace from the cache. Otherwise, we proceed further with the regular approach. Once we have found the accumulated stack size and accumulated stack trace of a function, we store that information in the cache.

1) A Statistical Analysis of Function Call References: Cache for functions will be useful only when there are multiple references to same functions in the flow database; if functions are rarely used more than once then cache does not help. Based on the experimental analysis described before, we found that cache will be useful. We intend to verify this claim by statistical analysis of function call references in the flow database.

If a function \( f \) is called by function \( g \), that is a reference to \( f \). Using the flow database, for each function we computed how many times it is referred. Then we sorted these functions in non-ascending order (with the function which is referred most occupying first slot). To get a better appreciation of the function reference distribution, Table II is created from the function reference data. The first row indicates that first 10% of the functions account for 71% of total references and so on. Clearly, function call references are following Pareto’s law. Based on these observations, it should be clear that having a cache will be very helpful.

2) Evaluation: To compare the effectiveness of caching in reducing the time to run the tool, we ran the tool with cache on the same set of tasks we ran before on Phase 1 of the tool. The results appear in Table I. We can see that the tool works much faster with caching. We ran the tool on the tasks in the same order as shown in the table. For the first task the cache is cold (no cache entries). That’s why it takes considerably more time compared to other tasks. The subsequent tasks benefit from the cache entries created by the previous tasks. So, the times depend on the order in which we run the tool. This is not the case with respect to cacheless approach – it always takes the same time irrespective of the order. The tasks shown in the table are taken from different modules. The times indicate the considerable commonality of function usage among tasks.

C. Phase 3: Coping with Recursion

Till now, we have assumed that the task for which we want to find the accumulated stack size does not have recursion. Now, we lift that restriction. A function is recursive if starting from that function we can reach the same function in the call flow graph. Note that if a function \( f \) calls a recursive function \( g \), that itself does not make \( f \) a recursive function.

By their very nature, recursive functions take more stack space and do not lend themselves easily to come up with a bound on the stack size. So, the use of recursion in embedded systems is less [20]. Data communication systems are no exception. Therefore, for our tool, we have decided not to take recursion depth into account for stack sizing. But we still need to handle recursion; otherwise, the tool will go into in-
(stackSize, stackTrace, rec) findStackSize(file, fn, ft) 

{ 
    (definedFile, fnSize, calledFnList) ← findFuncInfo(file, fn) 
    if (funcRepeats(definedFile, fn, ft))
        Note all the functions in the loop as recursive 
        Store recursion loop trace 
        return (0, NULL, (definedFile, fn)) 
    if (definedFile, fn) exists in nonrecursive cache 
        return (stackSize from cache, 
            stackTrace from cache, {}) 
    if (definedFile, fn) exists in recursive cache 
        Retrieve stSize and rTrace from cache 
        if (no duplicates between ft and rTrace ) 
            return (stSize, rTrace, {}) 
        currentSize = 0; currentRec = {}
    Append (definedFile, fn) to ft 
    for each function in the calledFnList 
        (calledFnSize, tempTrace, tempRec) ← findStackSize(definedFile, calledFn, ft) 
        currentRec = currentRec U tempRec 
        if (calledFnSize > currentSize ) 
            currentSize = calledFnSize; 
            currentTrace = tempTrace 
    Prepend (definedFile, fn) to currentTrace to get stackTrace 
    currentRec = currentRec - {(definedFile, fn)} 
    if (definedFile, fn) is a nonrecursive fn 
        Store the nonrecursive cache entry: 
        ((definedFile, fn), fnSize + currentSize, cacheTrace) 
    else if (currentRec is an emptySet) 
        Store the recursive cache entry 
        ((definedFile, fn), fnSize + currentSize, cacheTrace) 
    return (fnSize + currentSize, stackTrace, currentRec) 
}

Fig. 5. Pseudo code for Phase 3 and Phase 4 of the tool

finite loop. Accordingly, the definition of accumulated stack size is updated:

**Definition:** The accumulated stack size of a function is the maximum stack size of all traces starting from the function in which no function repeats itself. The corresponding trace is referred to as accumulated stack trace.

Since we are not taking recursion depth into account, it is important to notify the user whenever recursion is encountered. This informs the user the presence of recursion which is useful as the underlying theme is that there is not much recursion in the system; if there are lot of recursive functions, user may decide an appropriate course of action.

The pseudo-code shown in Fig. 5 (ignore the code in italics) extends the Phase 2 tool to handle recursion. findStackSize() traverses the entire call flow graph. To detect the presence of recursion, we have added an additional argument to findStackSize(). The argument ft represents the “forward trace” in the call flow tree. Whenever findStackSize() is visiting a particular node of a tree, ft gives the entire flow trace starting from the root of the tree to the parent of that node. Using that trace it will be trivial to detect the presence of recursion, which is done by funcRepeats(). It checks whether the current node is same as any of the nodes in the forward trace. If so, we have encountered recursion and we record the fact. We find the recursive loop of functions and note all of them as recursive. We do not explore the tree downward anymore (based on our design decision).

Because we are maintaining cache only for non-recursive functions, before storing the cache entry we need to find whether the function under consideration is nonrecursive or not (we note whether a function is recursive or not while checking in funcRepeats()).

1) Evaluation: We ran the tool on a set of tasks chosen from DC software. These tasks are known to have recursive functions. The timing results appear in the second column of Table III. The tool was run on tasks declared with “medium” stack size (3 kilo bytes). It can be seen that the time taken by the tool is in the order of minutes. If we run the tool on “very large” stacks, the time would be very high. Note that this phase of the tool already has caching for nonrecursive functions.

The timing analysis revealed that most of the time is spent in traversing the recursive function traces. Some of the recursive traces are involved with widely used functions (e.g., syslog). That most of these traces are long and occur often explain the considerable time taken by the tool. It follows that if we can use caching for recursive functions as well, we may realize savings in time.

D. Phase 4: Cache for Recursive Functions

Caching for recursive functions is complicated compared to caching for nonrecursive functions. The first complication relates to creating a cache entry for a
recursive function. There are two ways we can create a cache entry. One approach is to create a cache entry for a function \( f \) when we are sizing the stack for \( f \). The second approach is to create the cache entry not only for \( f \) but also for all the nodes involved in the call flow tree of \( f \) when we are sizing the stack for \( f \). The second approach is better for efficiency and is what we followed for nonrecursive functions. We intend to follow suit for recursive functions. But there is an issue here. An example may make it clear.

Suppose, we are finding the stack size for recursive function \( f, f \) calls \( g \), another recursive function. From \( g \), we build its flow tree. If we intend to cache the stack size for \( g \), we need to completely explore the flow tree anchored by \( g \). But if \( f \) is encountered while exploring the flow tree of \( g \), we stop exploring there (as per the definition of accumulated stack size, we need to visit a node only once). Since, we have not fully explored the call flow tree for \( g \), we can not come up with the accumulated stack size for \( g \). We can come up with the accumulated stack size for \( g \) only when we explore the complete call flow tree for \( g \). We can explore fully the call flow tree for \( g \) only when we do not encounter \( f \) during exploration.

We can generalize the above situation. Suppose, \( g \) is an internal node in the flow tree of \( f \). \( f \) is the list of nodes from \( f \) to \( g \). We can come up with accumulated stack size for \( g \) only when none of the nodes in \( f \) are encountered while exploring the call flow tree of \( g \).

The second complication relates to using a cache entry for a recursive function. The caching for nonrecursive functions is fairly simpler because nonrecursive functions obey (1). Recursive functions, however, do not obey that equation in general. Again, an example may make it clear.

Consider two mutually recursive functions \( f \) and \( g \) each with an individual stack size of 10, which call each other but no others. Hence, \( f \) and \( g \) have accumulated stack sizes of 20 and 20 respectively and accumulated stack traces of \( f \rightarrow g \) and \( g \rightarrow f \) respectively. Note that neither \( f \) nor \( g \) obey (1).

It is not difficult to see why recursive functions might not obey (1). When \( f \) is a recursive function, it is possible that accumulated stack trace of its called functions might include \( f \). In that case, \( f \) will not be able to utilize such accumulated stack traces (as per definition of accumulated stack size).

Based on these observations, we generalize (1) to all types of functions. Suppose function \( f \) calls functions \( g_1, g_2, ..., g_n \). Out of this, without loss of generality, accumulated stack traces of functions \( g_1, g_2, ..., g_m \) do not include \( f \). Let \( T \) be the call flow tree of \( f \) excluding the subtrees anchored on \( g_1, g_2, ..., g_m \) nodes. Then

\[
\text{Acc}(f) = \text{Ind}(f) + \max(\text{Acc}(g_1), ..., \text{Acc}(g_m), \text{Acc}(f \text{ in } T))
\]

(2)

Note that in the case of a nonrecursive function \( f \), (2) becomes (1) as none of the called functions’ accumulated stack traces will include \( f \) and the remaining subtree \( T \) will be empty.

1) Pseudo-code: The pseudo-code shown in Fig. 5 embodies the above ideas to come with accumulated stack size using cache for recursive functions. The code is fairly complicated by the presence of both nonrecursive and recursive caches. Unlike the previous phases of the tool, it is not obvious that this phase correctly computes the accumulated stack size, computes the accumulated stack trace, and forms the caches. The proof that the code does so is involved and lengthy [21]. In the discussion of the code below, we give intuitive reasoning to show that the code works correctly.

The main change in this phase is the third return value of \texttt{findStackSize}(). It represents the set of recursive functions that belong to forward trace \( f_t \) that are encountered while exploring \((\text{file}, \text{fn})\). This is useful for taking care of the first complication explained above. While going through the code, we shall explain how this is computed.

\texttt{funcRepeats()} checks whether we are encountering a function again; if we encounter a function again, we return the encountered function. Next we check for the function in both the caches, nonrecursive and recursive. Looking up in cache for nonrecursive functions is same as before. When we look up in cache for recursive functions, we retrieve the accumulated stack trace for the function (something we do not do for nonrecursive functions). We use this trace to check whether there are any common functions between \( f_t \) and the retrieved trace. If there are any common functions, we can not utilize the cache entry. Instead, we have to freshly explore the flow tree. This is related to second complication explained above. On the other hand, if there are no common functions, then we can utilize the cache entry; we return from the function then (note that we return empty set for the list of recursive functions encountered in that case).

If there is no cache hit, we resort to regular processing to find out the stack size, as done by the first phase of the tool. The only additional detail is computation of \texttt{currentRec}, which is the union of \texttt{rec} sets of the called functions. Before going through the called functions, we have added \((\text{defineFile}, \text{fn})\) to the forward trace. So it could potentially be part of \texttt{currentRec}. After we complete the recursive tracing of called functions, we remove \((\text{defineFile}, \text{fn})\) from \texttt{currentRec}. That gives the set of recursively called functions which belong to \( f_t \) as originally passed to \texttt{findStackSize()} for \((\text{file}, \text{fn})\). If \texttt{currentRec} is empty it means that we did not prematurely terminate the tracing of \( f_n \) due to encountering any functions in the forward trace. Then only we can store the stack size and trace in the cache. Note that if \( f_n \) is a nonrecursive
are outlined in Section II-C.

context sensitive analysis, which models an assembly

A.1). But for compilers that use incremental approach,

self-modifying code, and functions with dynamic stack

E. Limitations and Future Work

We found the tool to be effective in practice in
determining the stack sizes of tasks, but it has some
limitations. Since the tool is static analysis based, it
shares most of the limitations of such tools (see Section II-C). These limitations include the inability to handle: dead code, function pointers, indirect jumps,
self-modifying code, and functions with dynamic stack frames.

Our modelling the flow graph with a function as a
node leads to context insensitive analysis, which
accurately finds the stack size for code generated by
compilers that use lumped approach (see Section III-A.1). But for compilers that use incremental approach, the tool will overestimate the stack size. For accurate stack sizing for code generated by such compilers, context sensitive analysis, which models an assembly statement as a node in the flow graph, is required.

Tackling the limitations of the tool is a primary focus for future work. Some ideas toward that aim are outlined in Section II-C.

IV. CONCLUSIONS

Accurate stack sizing gets rid of insidious run time
stack overflows and conserves memory. Run time approaches and static analyses are the two methods for stack sizing. Where as static analyses tend to be faster and give better bound on stack size compared to run time approaches, static analyses suffer from limitations in handling function pointers, recursion, etc. Since static analyses and run time approaches suffer from different limitations, it is better to use both the approaches in practice.

Unlike the existing tools, which are targeted at single product small software systems, our tool is meant for multi-product large software systems. Our tool is designed using a front-end and back-end approach to easily port it across different processors. The existing tools will be very expensive in time when applied to large software systems. To build an efficient tool, we employ two strategies. The first strategy is to reduce the size of the flow graph by modelling a function as a node rather than an assembly statement as a node. The second strategy is to use caches. Statistical analysis of function references, which showed that function references follow Pareto’s law, indicated that caches may be useful; experimental analysis confirmed the effectiveness of caches in making the tool efficient.

REFERENCES