Static Stack Depth Profiler

Functional Specification

General Description

The static stack depth profiler will provide information to the user about the maximum stack depth requirements of their application based on the static information available to it in the output file generated by the linker.

The profiler will be implemented as a stand-alone application called ***sdp470***. The profiler will take a linked output file as input and produce a listing that details the stack usage of all of the functions defined in the application. If an application contains indirect calls and/or reentrant procedures, then a configuration file should also be provided as input to the profiler.

The syntax for invoking the static stack depth profiler is as follows:

`sdp470` `[-c config] out-file`

`-c config` Identifies a configuration file to be used by the profiler to supply information about indirectly called functions and reentrant procedures (for more information on this topic, see *Configuration File Specification* section below).

`out-file` Identifies linked output file for an application to be analyzed by the profiler. This file will contain debug information about all functions included in the final link of an application.

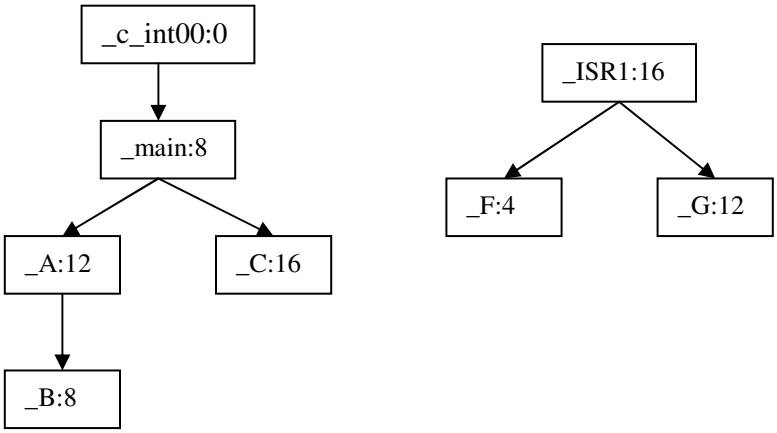
Stack Depth Statistics Listing

Each segment of the listing will detail the stack usage for a particular function in the application. Function segments are listed in order beginning with the function that has the highest total stack need.

The first line in a segment will provide details about a given parent function. The first line in a function segment will provide information on the stack space needed by the function and what its total stack usage estimate (function's stack usage plus the worst stack usage of its callees).

Subsequent lines in a function segment will list the callees of the parent function listed in the first line of the segment. The callees are listed in order from highest to lowest stack usage.

For example, given the following call trees for an application:



The function segment listings will look like this:

```
*******************************************************************
* Static Stack Depth Analysis Profile
*******************************************************************
FCN:cint00              stack usage:  0,  total stack need  :   28
          _main         stack usage:  8,  subtree stack need:   28
===================================================================
FCN:_main               stack usage:  8,  total stack need  :   28
         _A             stack usage: 12,  subtree stack need:   20
         _C             stack usage: 16,  subtree stack need:   16
===================================================================
FCN:_ISR1               stack usage: 16,  total stack need  :   28
         _G             stack usage: 12,  subtree stack need:   12
         _F             stack usage:  4,  subtree stack need:    4
===================================================================
FCN:_A                  stack usage: 12,  total stack need  :   20
         _B             stack usage:  8,  subtree stack need:    8
===================================================================
FCN:_C                  stack usage: 16,  total stack need  :   16
===================================================================
FCN:_G                  stack usage: 12,  total stack need  :   12
===================================================================
FCN:_B                  stack usage:  8,  total stack need  :    8
===================================================================
FCN:_F                  stack usage:  4,  total stack need  :    4
===================================================================
```

A summary of the application's stack depth requirements will be provided at the end of the listing. The summary will state an estimate of the stack depth usage from each function in the application that does not have a parent. The user can derive a worst case estimate of an application's stack depth using the information provided in the listing in combination with their knowledge of which functions are interrupts (and which stack mode those interrupts assume).

For example, a summary of the above example will look like this:

```
================ROOT FUNCTIONS==============================
_c_int00          total stack need:      28 bytes
_ISR1             total stack need:      28 bytes
```

Dependencies and Limitations

The profiler will construct a call tree based on the debug information provided in the linker output file. The tree will be annotated with the stack usage information that the profiler derives from the function's debug information.

There are several significant limitations to the profiler's ability to complete this information:

- Hand-coded assembly functions and stack usage information for those functions
- Indirectly called functions and their callers
- Reentrant functions and their depth of recursion

The profiler will rely on user input to supply this information in the application source code and in the configuration file (specified with the –c option). The assembler will process the assembly source code annotations (.asmfunc/.endasmfunc directives) and encode the information into the linked output file in a form that the profiler can understand. The user must also provide a configuration file to identify indirect calls and reentrant procedures to the profiler. The profiler will use this configuration file to annotate the call graph that it constructed from the linked output file for the application.

The accuracy of the stack depth estimate is heavily dependent on the accuracy of the information provided by the user. The user must actively maintain this information throughout the life of a given application.

User Input Mechanisms

The assembler will support assembler directives, ***.asmfunc*** and ***.endasmfunc***, for identifying the beginning and end of an assembly function. The ***.asmfunc*** directive will support an optional parameter for specifying the stack space needed for an assembly function.

For example, you could write a function, $foo(), which uses 12 bytes of stack space, as follows:

```
        .global    $foo
$foo:   .asmfunc   stack_usage(12)
        PUSH       { R4, R5, R6 }
        …
        POP        { R4, R5, R6 }
        MOV        PC, LR
        .endasmfunc
```

The ***.asmfunc*** and ***.endasmfunc*** directives identify the entry and exit points of the function. This information and the fact that $foo uses 12 bytes of stack space are then encoded into the debug information. Assembler functions must be annotated with these directives to be considered for stack depth analysis and profile report.

Configuration File Specification

The stack depth profiler enables users to specify indirect calls and re-entrant procedures using a configuration file. The configuration file is input to the profiler, along with the executable to be analyzed. The profiler applies the configuration information to the stack depth information extracted from the executable and reports the total stack usage.

The configuration file is composed of a series of sections delimited by the section type. Each section contains information for a specific section type. For example, the section specifying indirectly called functions is delimited by **<INDIRECT>** and **</INDIRECT>**, and contains information on the set of procedures called indirectly.

Specifying Indirect Calls

```
<INDIRECT>
func1: callee1[(mode)], callee2[(mode)],…,calleeN[(mode)]
func2: callee1[(mode)],…,calleeN[(mode)]
…
funcm: callee1[(mode)],…,calleeN[(mode)]
</INDIRECT>
```

The keywords **<INDIRECT>** and **</INDIRECT>** delimit an indirect callee section. Each line in the section begins with the name of a calling procedure, *func*, followed by a colon (**:**), which is then followed by a list of functions that are called indirectly from *func*. Each entry in the indirect callee list consists of the callee name, and an optional mode (in parentheses) in which the callee was compiled. The profiler recognizes the specification of one of three modes: ARM, THUMB, and DUAL (default). If a mode is not specified, the profiler will assume DUAL mode. For DUAL mode callee functions, the profiler automatically adds the names of the callees (in ARM and THUMB modes) to the callee list for *func*. In the following example, function *bar* is specified as a callee in DUAL mode, and both *_bar* and *$bar* are added to the callee list for *main*.

```
<INDIRECT>
main:foo(ARM),bar
</INDIRECT>
```

The user may also specify each indirect call on a separate line within an indirect call section:

```
<INDIRECT>
main:foo(ARM)
main: bar
</INDIRECT>
```

The profiler performs no semantic checks on the information input via the configuration file. That is, it does not check to determine if the procedures listed in the callee list or the caller list exists or if they exist in the mode specified. The profiler will check, however, for available debug information for a function and generate a warning if none is found.

The profiler declares the following types of loads of PC (in ARM) mode as indirect call points

LDR{B,H} PC,[reg,#imm]

Where #imm is an immediate value. However, loads of PC with register offsets are not identified as call points since such loads are used generally for switch table jumps.

Specifying Re-entrant Procedures

Reentrant functions present a difficult problem to the profiler's ability to accurately estimate the worst-case space requirement of an application. One difficulty with reentrant functions is that the depth of recursion is often data-dependent. It may be impossible for a user to accurately inform the profiler about the depth of recursion without knowing in advance all of the input data sets that will be used in the application. Another difficulty is the presence of multiple recursive functions in a call-graph, which would make it hard to determine the recursive behavior of each function in the call-graph.

In light of these difficulties, we do not perform any automated analysis to determine the presence of reentrant functions or their nature. Instead, we rely on the user to input the maximum recursive depth of a function. The profiler assumes that the user input is correct and reliable, and warns the user if there is no apparent recursion in the call graph for a function and the user has specified a recursion depth. Beyond the required analysis to issue such a warning, the profiler does not perform any analysis on reentrant calls.

The configuration file enables the user to specify the list of reentrant procedures in the application. The format is as follows:

```
<REENTRANT>
func1(depth)
func2(depth)
</REENTRANT>
```

The re-entrant section is delimited by the **<REENTRANT>** and **</REENTRANT>** keywords as shown above. Each line in the section consists of a function name followed by the recursion `depth` (in parentheses) to be assumed for the function. The `depth` value should be a valid, non-negative integer. The profiler multiplies the stack size of the function by its specified `depth` to obtain the final stack usage requirement for the function. For example,

<REENTRANT>
foo(5)
</REENTRANT>

If the profiler determines that the static stack usage requirement of *`foo`* is 100, the maximum dynamic requirement of *`foo`* would be 100*5 = 500. The depth of a function defaults to 1 in the absence of a `depth` field

The profiler does not make any effort to determine if the procedure specified in the re-entrant section is indeed recursive. It assumes that the user input is correct and reliable.

Specifying Interrupt service routines

Interrupt service routines (ISRs), as the name indicates, are used to handle interrupts. An interrupt service routine is a special root function that is usually not directly called from the application. In ARM, an interrupt request can be service by an ISR in any of the six modes. Each mode typically uses a different stack area for the interrupt handler. The SDA can output the stack requirements of an ISR if the user can identify ISRs and their modes of operations. The syntax for specifying an ISR and its mode is given below

<INTERRUPT>
routine_name(mode)
</INTERRUPT>

Where routine_name is the name of the ISR and mode can be one of

1. *irq* (general purpose interrupt),
2. *fiq* (data transfer or channel process)
3. *svc* (operating system protected mode)
4. *abt* (data or instruction prefetch abort)
5. *sys* (privileged user mode for the operating system)
6. *und* (undefined instruction).

The stack space requirement for ISRs are output after the stack space requirements for routines directly called by the application.

Other Features

The configuration file supports all pre-processing directives supported by cl470. The configuration file should be processed using cl470 pre-processor before being input to the profiler. For example, if file "config1" contains:

<INDIRECT>
main:foo
</INDIRECT>
#include "config2"

and file "config2" contains:

<REENTRANT>
foo(5)
</REENTRANT>

the cl470 pre-processor can be used to combine the 2 config files. For example:

cl470 –ppo config1

The pre-processor will add or replace the file suffix with .pp. In the above example, the combined config files will be found in the file "config1.pp".

The profiler supports C++ style comments. All characters following the // symbol to the end of the line (new line character) are ignored. The // can appear anywhere in a configuration file. For example

```
<INDIRECT>
// This is a comment
main:foo
// End of indirect callee list
</INDIRECT>
```

The configuration file does ***not*** permit nesting of sections. For example, the following specification is illegal

```
<INDIRECT>
foo:bar
<REENTRANT>
foo(5)
</REENTRANT>
</INDIRECT>
```

It is worth re-iterating that the profiler performs ***no*** semantic checks on the information specified in the configuration file and assumes that the information provided is correct and valid. It is the user's responsibility to ensure the validity of information specified in the configuration file.

Tail Calls

The SDA handles tail calls (branching to a routine *foo*() at the end of routine *bar*() so that control returns directly to *bar*'s caller at the end of *foo*()) by identifying tail call points as both call point and return points. The pseudo assembly instruction CRET is used for representing tail calls.

Profiler Generated Warnings

The static stack depth profiler will detect the following situations and issue a warning in each case:

- If a function contains an indirect branch (branch to the contents of a register) and there are no <INDIRECT> entries for the function in the configuration file, the profiler will issue a warning that a potential indirect call has been observed and the user has ***not*** specified this caller/callee pair in the configuration file. Not all indirect branches are function calls; e.g., in 32-BIS mode, long branches are achieved through indirect branches.
- If the user specifies a set of indirect callees for a function and the profiler observes no indirect branches in the function, it issues a warning to the user indicating the absence of indirect calls.
- If a function's call graph has a link back to the function indicating potential recursion and the user has specified no recursion depth, the profiler issues a warning to the user indicating that the function is reentrant. The reason for issuing a warning and not an error is that a function could have an apparent recursion and not a real one.
- Similarly, if the profiler discovers no links back to the function in a call-graph and the user has specified a recursion depth, it issues a warning to the user.
- If no stack usage information for a function is found.
- If no debug information for a function is found.
- The profiler will generate potential indirect callee warning for each routine compiled in dual mode. This is because, in dual mode, each call to an external routine is performed by jumping to the RTS routine IND_CALL or IND$CALL (depending on the mode of the caller)
- The profiler will generate potential indirect callee warnings for routines in the RTS library.

Run-time Support for Stack Depth Profiling

The latest version of the ARM compiler provides a debug option to enable the user to determine the maximum stack usage of an application. If the application is compiled with –debug:sdp command line option, the compiler places a call to a stack depth bookkeeping routine (C_SDPBK for 32-BIS and C$SDPBK for 16-BIS), immediately after the frame has been allocated for a function. This bookkeeping routine tracks the maximum stack usage for each function. We have also made available a profiled version of the run-time support that can be linked when –debug:sdp is turned on. This would enable the user to determine the complete stack usage of an application that contains run-time support utilities (such as I/O utilities).

There is an overhead associated with the –debug:sdp option. The link register (LR) is always saved when this option is turned on, since all functions in the application call the bookkeeping routine.

Frequently Asked Questions

1. *Does the Stack Depth Analyzer has to determine the greatest possible stack demand for User-, FIQ-, IRQ-, Supervisor-, Abort-, and Undefined Stack [modes]?*

   We do not feel that it is necessary for the profiler to be aware of which stack mode is assumed for any given function. The profiler will provide stack usage information about every function in the application, including interrupt functions. The user can then combine their knowledge of the application with the profile data that is generated to determine what their worst case stack depth estimate is.

2. *Is stack depth has to be reliable? Is stack depth is correctly analyzed in all cases?*

   There are three situations that may arise in an application that will cause the profiler to rely on additional user input to complete the details of the call graph: assembly functions, the presence of indirect calls and/or recursion. The assembler supports .asmfunc and .endasmfunc directives to help the user annotate assembler functions with the debug information needed by the profiler to include the function in the call graph. Indirect calls and re-entrant procedures are identified by the user in a configuration file that is consumed by the profiler and used to annotate the call graph. These input mechanisms are discussed in more detail above.

3. *How fast is stack depth analysis? Can results be obtained in a reasonable amount of time (i.e. 5 to 10 minutes or less)."*

   This is not an issue. The current profiler prototype runs quite fast (a few seconds).

4. *Does stack depth analysis increase code size or execution time of the application?*

   The profiler relies on the profiling debug information generated by the compiler and assembler to build a call graph that it can analyze. The compiler and assembler will produce default debug information to allow the profiler to analyze the stack depth requirements of an application without impacting the size or performance of the application.

5. *Does stack depth analysis work on Win98, Win2000, and WinNT?*

   The source code for the profiler will be ported to all of the host operating systems that are currently supported.

6. *Does stack depth analyzer work with stack manipulations in the application code?*

   The profiler will make not detect changes in stack mode during a function. Stack usage statistics will be reported on a function by function basis, for every function that is included in the final link of the user's application.

7. *Are there any restrictions on C-compiler or linker optionsallowed?*

   The application cannot be compiled with STABS type debug enabled (-gt option). The profiler relies on DWARF type debug information to be produced by the compiler and assembler.

8. *How are indirect function calls and recursive functions handled?*

   Please see above section describing the *Configuration File Specification*.

9. *How are entry points identified?*

   The entry point, or root function, is identified in the linked output file. The profiler uses the same policy to determine the entry point as the linker:

   1. A global symbol identified with –e linker command option
   2. The value of the *_c_int00* symbol (if present)
   3. The value of the *_main* or *$main* symbol (if present)
   4. Address 0x0 (the default