Proceedings of the
GCC Developers' Summit

June 22nd–24th, 2005
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*
Stephanie Donovan, *Linux Symposium*

Review Committee

Eric Christopher, *Red Hat, Inc.*
David Edelsohn, *IBM*
Richard Henderson, *Red Hat, Inc.*
Andrew J. Hutton, *Steamballoon, Inc.*
Janis Johnson, *IBM*
Toshi Morita
Gerald Pfeifer, *Novell*
C. Craig Ross, *Linux Symposium*
Al Stone, *HP*
Zack Weinberg, *Codesourcery*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

Compile-time stack requirements analysis with GCC

Motivation, Development, and Experiments Results

Eric Botcazou, Cyrille Comar, Olivier Hainque
AdaCore
`{botcazou, comar, hainque}@adacore.com`

Abstract

Stack overflows are a major threat to many computer applications and run-time recovery techniques are not always available or appropriate. In such situations, it is of utmost value to prevent overflows from occurring in the first place, which requires evaluating the worst case stack space requirements of an application prior to operational execution time. More generally, as run-time stack areas need memory resources, information on the stack allocation patterns in software components is always of interest.

We believe that specialized compiler outputs can be of great value in a stack usage analysis framework and have developed GCC extensions for this purpose. In this paper, we first expand on the motivations for this work, then describe what it consists of so far, future directions envisioned, and experiments results.

1 Motivation

1.1 Need for stack requirements analysis

Unlike dynamic heap allocations, stack allocations are most often implicit from the programmer's standpoint and there is no standard notion of easy to check stack allocation failures. Besides, stack areas have to be sized beforehand in many cases, and when the amount of stack actually used by an execution thread exceeds an anticipated size, nasty events happen. This is a "stack overflow" condition, the exact consequences of which depend on the operating environment and on the execution context. In the worst cases, random memory is clobbered with arbitrary follow-on effects. Run-time detection techniques a-la GCC -fstack-check limit the most catastrophic effects but leave the program in an uncomfortable state. Even with languages such as Ada offering recovery mechanisms to the programmer for such events, the situation is far from perfect: very little can be done when an execution thread does not have access to the stack space it needs for proper operation.

In many situations, it is of utmost value to prevent overflow conditions from occurring in the first place, which requires a way to evaluate the worst case stack space requirements of an application. In any case, having ways to evaluate those requirements early avoids hitting erratic run-time behaviors with high implied costs because uncovered late in the development process and often hard to characterize. It also allows developers to communicate on the stack consumptions of the interface points they provide, which allows later analysis to proceed even if the module sources are not available.

There are two broad categories of approaches to address the high level issue outlined here : Testing based and Static Analysis based [11, 12], which often complement each other. We believe that GCC can play a major role in a stack analysis framework of the latter category.

We will now describe the two categories of approaches and why we believe that a compiler based solution is sensible.

1.2 Testing based approaches

Testing based approaches consist in measuring the maximum amount of memory actually used in deliberately oversized stack areas while running or simulating the application. One possible technique is to initialize the areas with a repeated pattern and measure the size of the parts clobbered.

A positive aspect is that the measurements provide live data from actual executions or simulations of the code. Every software has to be tested for numerous reasons anyway, and getting additional feedback from the testing campaign is always valuable. Furthermore, the set of testing scenarios is usually augmented for the specific measurement purposes and this benefits the software general robustness.

Testing based approaches suffer a number of weaknesses, however.

To start with, the testing scenarios most often cover only a subset of the possible execution paths, and so do not expose the absolute worst case consumption. This is especially true when interrupt handlers are involved, as they make the system worst case behavior very timing and environment dependent, hence an extremely improbable and hard to produce event.

Then, it is crucial to exercise as many execution paths as possible, and significant costs are entailed because working out and running a proper set of tests requires a lot of resources.

In addition, the technical mechanisms required to perform the measurements may have undesired side effects, such as subtle timing influences twisting the observation or with critical consequences in the case of a real-time system. It may also happen that the target environment is not easily suited to these technical requirements, as in some typical cases of embedded systems where the available resources are really scarce.

Finally, as implied by the previous point, testing based approaches often need to be as non-intrusive as possible to avoid perturbing the observed execution and wasting precious run-time resources. To satisfy that constraint, only restricted feedback is allowable, with very little information on the observed worst case context, and this does not help much the development process.

1.3 Static Analysis based approaches

Static Analysis based approaches consist in using tools to analyze the application stack space consumption patterns and possibly compute worst case bounds prior to execution time. They usually perform some local stack consumption analysis combined with control-flow graph traversals. The bound computation is actually a small subset of a wide category of resource bounding analysis problems, focus of a lot of research activity over the years. See [2] for an example set of publications in this area, or [9] for a specific instance.

Static Analysis schemes operate over a rich variety of technical grounds, the choice of which influences a lot the contexts in which they may be used. Consider for example [13], describing formal transformations of functional programs into resource usage functions returning bounds on the stack or heap usage of the original program. Although presented as adaptable to imperative languages, this approach would require a large amount of work to become applicable to general purpose languages like C, C++, Java or Ada. Actually, all the analysis schemes we know of are able to operate only within a comprehensive set of assumptions on the target environment and on the programs to be analyzed.

When available, static analysis approaches address the whole set of concerns expressed in the previous section about testing based approaches. They can provide precise results without much effort, rapidly and early in the development process. They also have no runtime side effects and are not constrained by the target resources so have more room to offer as much and various feedback as desired.

Regarding hard bounds computation, they all hit a common set of challenges:

- *Cycles in the Control Flow Graph* : In presence of control flow cycles involving stack consumption, the worst case bound is a function of the maximum number of cycle iterations. When bounds on such iteration counts are unknown, the amount of potential stack usage is infinite. Bounding the iteration count is very often difficult or impossible.

- *Indirect jumps or subprogram calls* : Determining where such transfers of control lead is often not achievable, and the corresponding stack usage is then unknown.

- *Unresolved calls* : These are calls to subprograms for which no stack usage information is available, introducing an unknown amount of stack usage in a call chain.

- *Dynamic stack allocations* : For instance from `alloca` with a variable argument in C, or from dynamically sized local objects in Ada. They introduce potentially difficult to bound variable amounts of stack usage on the paths where they appear.

- *Possible interrupt events* : When interrupt (hardware or signal) handlers run on the currently active stack, their consumption has to be accounted for when sizing the various areas in which they may run. When they run on their own dedicated stack, this one has to be properly sized too. In any case, how interrupts may preempt each other greatly affects the associated possible stack usage and is not easy to determine automatically. [7, 10, 8] and [11] are examples of research literature on this matter.

- *Dynamic global behavior* : Analysis tools typically include in their evaluations many paths that can never be taken at run-time, referenced as False Paths in [6] or [5]. Excluding some of such paths on safe grounds may, for instance, allow cutting some control flow cycles or the production of tighter worst-case bounds, saving runtime resources. [11] illustrates the use of such techniques to automatically compute the hardware interrupt preemption graph for an AVR target. This is a hard problem in the general case.

1.4 Compilers as a static stack requirements analysis framework component

Despite the set of challenging issues enumerated in the previous section, static analysis based approaches to stack requirements evaluation remain very appealing for a number of reasons.

As already pointed out, they alleviate a number of weaknesses of the testing based approaches when strong guarantees are required and all the constraints to allow such guarantees are satisfied. Actually, avoiding challenging program constructs like recursive/indirect calls or dynamic stack allocations is often part of already established coding guidelines in environments where preventing stack overflows is a real concern. Moreover, the challenging constructs for a static analyzer most often also are challenging for a testing based approach, and potentially not even identified as such. Finally, even when absolute bounds may not be computed, analysis tools are still able to provide very useful feedback on the analyzed stack usage patterns.

The few existing practical solutions we know of [1, 3, 11, 6, 8] work as binary or assembly level analyzers, which gets them a very precise view of what the code is doing and opens specific opportunities. For example, [11] exposes a binary level "abstract interpretation" scheme for AVR microcontrollers, analyzing the operations on the hardware interrupt control bits to infer an interrupt preemption graph. This allows the detection in the interrupt preemption graph of undesired cycles that could have been introduced by programming mistakes, and minimizes the amount of extra stack space to add for potential interrupts at any point. The scheme is in theory adaptable to other target architectures, provided the development of a comprehensive machine code parser and instruction classifier to distinguish calls, jumps, stack pointer adjustments and so forth. [6] develops similar ideas for Z86 targets.

We suggest an alternate kind of scheme here : develop dedicated compiler extensions to produce specialized outputs for stack requirements analysis purposes. This is what we have implemented in GCC together with a prototype analyzer to process the outputs, as described in section 2.

One limitation is that a compiler cannot provide information on elements it doesn't process, such as COTS operating system services for which sources are not available or very low level routines developed in assembly language. When worst case bounds are a strong concern, not having the sources of some components is rare, however, and the stack usage in assembly routines is usually simple enough to be accounted for manually.

The compilation process may also not be able to grasp the interrupt handling bits necessary to size the worst case amount of interrupt related stack, be it for hardware interrupt or signal handlers. Interrupt handling always requires very careful design and coding, though, so the information could at least also be provided to the framework by the user, or accounted for separately.

Leveraging on a compiler's internals has a number of advantages:

- *Reduced effort for new target architectures*: A compiler typically knows everything about stack allocations for the code it processes. When already structured to support multiple target architectures, this knowledge may be used for each and alleviates the need of an associated comprehensive machine instruction parser, potentially difficult and error-prone for complex processors.

- *User level guards*: A compiler can be tailored with options to raise warnings or errors on user level constructs known to trigger stack allocation patterns that may cause troubles to a stack analysis framework involved later on. This points directly at the user level construct, and so very early in the development process, both being of precious value.

- *Access to high level information*: A compiler has visibility on semantic information that can help tackle some of the challenging issues we have previously identified. Consider an Ada `Integer` subtype with range `1..5` for example. If a variable of this subtype is used to size a local array, the range knowledge may be used to compute a bound on the array size, and so on the corresponding stack allocation. Potential targets of an indirect calls are another example. Based on subprogram profiles and actual references to subprograms, the compiler can provide a limited but exhaustive list of subprograms possibly reached by an indirect call. In both cases, the compiler information at hand is extremely hard, if not impossible, to infer from a machine level representation.

- *Scalability*: Support for stack usage outputs is unlikely to change how a compiler scales up against application sizes, so a compiler-based stack analysis component will scale up as well as the initial compiler did. Besides, a compiler has the opportunity to output no more than what is really relevant for later analysis purposes, which makes this output easier to digest than a huge machine level representation of the code.

2 Compile-time stack requirements analysis with GCC

2.1 Basic principles

We have developed a simple call graph based model from two new GCC command line options. They respectively generate per-function stack usage and per-unit call graph information, from which we build a multi-units call graph informally defined as comprising:

- One node per subprogram definition, valued with the maximum amount of stack the subprogram may ever allocate. For nodes performing dynamic stack allocation not trivially bounded, the value *includes* an unknown part, denoted by a symbolic variable named after the subprogram for later analysis purposes. We call such a node a *dynamic* node.

- One node per subprogram referenced from the set of processed compilation units, without being defined. Since the compiler has not processed the corresponding body, the associated stack usage value is unknown and also denoted by a symbolic variable for later analysis purposes. We call such node a *proxy* node.

- Directed edges to materialize a *may_call* relationship, where the source subprogram *may_call* the destination subprogram. Indirect calls with potentially unknown targets are represented as calls to a dummy proxy node.

We value the worst case stack consumption over any path in this graph as the sum of the values associated with each node on the path. As implicit from the graph informal description, this sum includes symbolic variables for paths with proxy or dynamic nodes.

This is a coarse grained representation, with the advantage of simplicity. As described later in Section 2.5, obtaining tighter worst case values is possible with finer grained representations and we already have tracks for future refinements on that account.

There is no special consideration for potential interrupt events at this point. As previously mentioned, they may either be included into the graph manually or accounted for separately.

2.2 New GCC command line options

We have implemented two new GCC command line options: `-fstack-usage` and `-fcallgraph-info`.

2.2.1 -fstack-usage

Compiling a unit X with `-fstack-usage` produces a text file X.su containing one line of stack allocation information for each function defined in the unit, each with three columns. In column 1 is the function name and source location. In column 2 is an integer bound to the amount of stack allocated by the function, to be interpreted according to column 3. In column 3 is a qualifier for the allocation pattern, with three possible values. `static` means that only constant allocations are made, the sum of which never exceeds the value in column 2. `dynamic,bounded` means that dynamic allocations may occur in addition to constant ones, for a sum still never larger than the value in column 2. This typically occurs for aligning dynamic adjustments from `expand_main_function`. Finally, `dynamic` means that dynamic allocations may occur in addition to constant ones, for a sum possibly greater than the value in column 2 up to an unknown extent.

This can be illustrated from the following C code in, say, `fsu.c`:

```
#include <alloca.h>

static void foo (void)
{ char buffer [1024]; }

static void bar (int n)
{ void * buffer = alloca (n); }

int main (void)
{ return 0; }
```

Compiling `fsu.c` on an x86-linux host with `-fstack-usage` yields `fsu.su` as follows:

```
fsu.c:4:foo     1040  static
fsu.c:7:bar     16    dynamic
fsu.c:10:main   32    dynamic,bounded
```

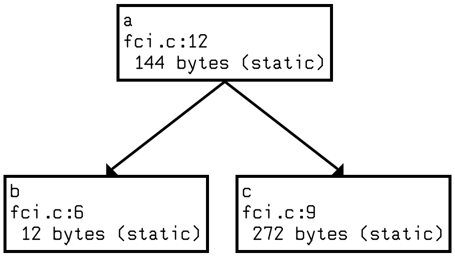
2.2.2 -fcallgraph-info

For a unit X, `-fcallgraph-info` produces a text file X.ci containing the unit call graph in VCG [4] form, with a node for each subprogram defined or called and directed edges from callers to callees. With `-fstack-usage` in addition, the stack usage of the defined functions is merged into the corresponding node descriptions, then qualified as *annotated*.

To illustrate, for the following `fci.c`:

```
typedef struct
{ char data [128]; } block_t;

block_t global_block;

void b (block_t block)
{ int x; }

void c ()
{ block_t local_blocks [2]; }

void a ()
{ int x;

  c ();
  b (global_block);
}
```

We obtain the annotated graph below:



2.3 Processing the outputs

There is a wide panel of possible applications using the new options outputs. Evaluating worst case allocation chains is one, and we have prototyped a command line tool for that purpose.

Our prototype analyzer merges a set of annotated `ci` files and reports the maximum stack usage down a provided set of subprograms, assorted with a call-chain. It is essentially a depth first traversal engine tailored for the kind of graphs we produce. Paths including proxy or dynamic nodes are always reported, as they trigger unknown amounts of stack allocation at run-time. The analysis currently stops at the first cycle and reports it.

For the 'a' entry point in the `fci.c` example, we get:

```
a: total 416 bytes
 +-> a : 144 bytes
 +-> c : 272 bytes
```

That is: "the worst case stack consumption down `a` is 416 bytes, after `a`, which may use up to 144 bytes, calls `c` which may use up to 272 bytes."

Although still experimental, this framework allowed conducting a number of instructive experiments, as we will describe in Section 3.

2.4 Implementation

We are only going to give a sketch of the implementation of the two new command line options. The bulk of the work and experimentation has been conducted on a modified 3.4.x code base, but we think the approach is easily adaptable to a 4.x code base. The options are independent from each other although they produce the most interesting results when used in conjunction.

2.4.1 `-fstack-usage`

The general principle is as follows: we directly leverage the long existing infrastructure for calculating the frame size of functions, made up of both a generic part and a target back-end dependent part, to report the amount of static stack usage for the function being compiled. As the back-end dependent part already needs to gather the final result of the calculation before emitting the assembly code, the actual implementation essentially boils down to modifying every back-end so as to be able to easily retrieve this result by means of a "target hook."

We found that, at least for the most common architectures, the changes to be made are very localized. The "target hook" implementation model proved to be a very efficient device and really minimizes the effort required to add support for a new target. x86, powerpc, sparc, alpha, mips and hppa have been covered up to now. The only challenge is to make sure that every single byte allocated on the stack by the calling sequence, even if it is not formally part of the frame, is taken into account.

In this context, one interesting technical point is of note: the difference in treatment between `ACCUMULATE_OUTGOING_ARGS` and `PUSH_ARGS` targets. In the former case, the arguments of called functions are accounted for in the final frame size, whereas they are not in the latter case; moreover, another subtlety comes into play in the latter case, in the form of the `-fdefer-pop` command line option, which instructs the compiler to not pop the pushed arguments off the stack immediately after the call returns. This may result in increased stack usage and requires a special circuitry to be properly dealt with at compile-time.

The remaining task is then to detect the dynamic stack usage patterns, much like what is implemented to support `-fstack-check`.

For this initial implementation, we mainly operate in the Tree to RTL expander by intercepting the requests for dynamic stack allocation. Moreover, as we are primarily interested in bounding them, we also try to deduce from the IL (here RTL) limits easily evaluated at compile-time. A technical detail that must not be overlooked here is that the compiler may generate code to dynamically align the stack pointer. While the amount of stack usage is easily bounded in that case, it must not be forgotten in the final result.

There is certainly room for improvement in either direction on the axis of compiler passes: by working later down the RTL optimization passes, one should be able to obtain additional constant bounds for dynamic allocation cases that are not inherently bounded; by working closer to the trees handed down by the front-end, one should be able to recognize inherently static allocation patterns that happen to require a dynamic-like treatment for specific reasons, as is the case for big objects when -fstack-check is enabled for example.

2.4.2 `-fcallgraph-info`

We again leverage an existing infrastructure of the compiler to implement this command line option, but a much more recent one, that is the "callgraph" machinery introduced in the 3.4.x series. We only use this machinery to gather information all the way through the compilation. We don't drive the compilation with it or attempt to rely on it for any code generation purpose, so our approach is not tied to one of the compilation modes supported by GCC 3.4.x. In particular, it works with the unit-at-a-time mode (C and C++ compilers at -O2), with the function-at-a-time mode (C and C++ compilers at -O0/-O1) and finally the statement-at-a-time mode (Ada compiler).

The general principle is straightforward: we record every direct function call the compiler has to process, either at the Tree level in unit-at-a-time mode or at the RTL level in non unit-at-a-time modes, and every indirect function call at the RTL level. Of course some of these function calls may end up being optimized away at one stage of the compilation but, as we aim at computing a worst case scenario, this conservative stance is appropriate.

However, an optimization technique relating to function calls is particularly of note since it can bring about huge differences in results for any types of callgraph-based analysis, depending on whether it is accounted for or not, that is function inlining. We therefore do arrange for eliminating or not registering in the first place in the call graph edges that correspond to function calls for which the callee is inlined in the caller in the assembly code emitted by the compiler. For the example of the `-fstack-usage` option, the immediate benefit is that the static stack usage of the callee is guaranteed not to be counted twice in the final calculation.

When additional analysis-oriented command line options are passed to the compiler, nodes in the call graph can be annotated with additional information gathered all the way through the compilation.

The final step is to generate an output suited to the kind of post-processing callgraph-based analyzes are naturally likely to require. In particular, as the process of merging .ci files is somewhat akin to linking object files, the problem of the uniqueness of the identifiers for the merge points has to be addressed. The VCG format for the output was chosen as a simple and natural medium to convey compiler-generated, callgraph-based information. It goes without saying that targeting any other (text-based) format is easily doable and would only command modifications at the very end of the chain.

2.5 Possible enhancements

This is a non exhaustive set of enhancements we are considering at various levels.

There is first large room for improvements in the post-compilation processing tool, which currently stops at the first cycle it sees and is not yet able to provide a rich variety of results.

Then, on the compiler side, the current implementation is low level to have visibility on every detail, and misses high level semantic information which would be useful to better handle a number of challenging constructs.

Finally, the current graph model could be refined to convey finer grained information on the stack allocation within subprograms, to let later analyzers compute tighter bounds. Let us consider the code for 'a' in `fci.c` to illustrate this point. Out of GCC 3.4.4 on x86-linux, with `accumulate-outgoing-args` turned off we see:

```
a: pushl %ebp
   movl  %esp, %ebp
   subl  $8, %esp
   call c
   addl $-128, %esp     <--
   [...]
   call b               <--
   subl $-128, %esp     <--
   [...]
```

Although the reported maximum stack usage, still 144, is correct for the function as a whole, 128 bytes are allocated only around the call to 'b', and the maximum at the call to 'c' is actually 16. Accounting for that would significantly lower the worst case sum, and such effects currently accumulate on deeper call graphs. A possible approach would be to have the graph convey context oriented information on edges, such as a value representing the maximum amount of stack used in the caller when the edge may be taken.

3 Experiments results

3.1 General description

We have first compared the compilation time of a large Ada system with a compiler having the support included and unused against the time with a compiler not having the support included. The difference was hardly noticeable on an unloaded average x86-GNU/Linux PC (a couple of seconds out of a 46+ minutes total time), showing that there is no visible compilation time impact from the support infrastructure when it is not used.

We have then performed various experiments with the new options, the prototype analyzer and a couple of "devices" to measure/observe the actual stack allocation at run-time. Those devices are a *fillup* scheme, filling oversized stack areas with repeated patterns then looking for the latest altered, and a GDB based *observation* scheme using watchpoints.

We have experimented this framework in different contexts for different purposes:

- On a large piece of an existing Ada application, to see if the approach scales up.
- On a set of Ada tests initially written for the GNAT High Integrity profiles and expected to fit the constraints for a static bound computation scheme, to confront computed worst cases with run-time observations.
- On a simplified Ada parser, much larger than individual tests in the previous set, to confront computed and measured values again and see how hard it is to adjust the coding style to fit a static analysis scheme.

All the framework experiments were conducted on an average PC running Windows.

3.2 On a large piece of Ada

This large piece of Ada is a significant part of an existing multi-tasking application running on HP-UX hosts and not at all written with the static stack analysis purposes in mind. The code base is approximately 1.280 million lines spread over 4624 units. There are very complex Ada constructs all over, and numerous reasons for a static analysis not to be able to provide a useful upper bound on the stack consumptions for the various Ada tasks.

Still, after having compiled this whole piece with the two new options, we have been able to produce a global annotated call graph in a matter of seconds, for 377905 edges and 99379 nodes. We were also able to evaluate the consumption down specific entry points in a few seconds.

This experiment told us that the approach does scale up very well, both performance and interface wise. We are able to efficiently get very useful text oriented feedback out of a large graph involving thousands of units, while a visual representation was clearly not adequate.

3.3 On a set of Ada tests for the GNAT High Integrity profiles

This experiment was to exercise the framework against a number of tests and compare computed worst case values with run-time observations. The set of tests is a selection among a series initially devised for the GNAT High Integrity profiles and expected to fit a static analysis experiment. In particular:

- They make a very restricted usage of run-time library components, thus avoiding complex constructs which can make static stack analysis difficult.
- They feature no indirect calls and very few call graph cycles or dynamically sized local variables.
- They are inputless, have a very stable behavior over different runs, and so are easy to study.

We ended up with over 10,000 lines of Ada in 14 separate tests together with their support packages.

Table 1 summarizes the first comparison we have been able to make between fillup measured consumptions and statically computed worst case values.

Test	Fillup	Static	Delta
01a	328	328	0.00%
02a	488	240	–50.82%
06a	8112	8288	+2.17%
08a	7932	8092	+2.02%
10a	7868	8032	+2.08%
11a	56	56	0.00%
12a	8040	8208	+2.09%
13a	5280	5452	+3.26%
16a	6732	6896	+2.44%
17a	8	8	0.00%
18a	272	272	0.00%
19a	88	88	0.00%
20a	832	896	+7.69%
21a	1584	400	–74.75%

Table 1: Fillup Measured vs Statically Computed worst case stack amounts (in bytes) on a set of High Integrity Ada tests

For most tests, the statically computed worst case is equal or only slightly greater than the observed maximum usage, as expected. We haven't investigated the detailed reasons for all those differences. A number of factors can come into play:

- The fillup instrumentation code only declares "used" the area up to the last *clobbered* word, possibly not as far as the last *allocated* word.

- All the experiments were performed without `accumulate-outgoing-args`, so overestimates by lack of context may appear as described in section 2.5.

- The reported worst case code path might never actually be taken during the test execution.

One big anomaly shows up from this table, though: for two tests (02a and 21a), the statically computed worst case is lower than the observed actual consumption, which we would expect never to happen. This was triggered by shortcomings in our first version of the prototype analyzer, which silently assimilated the variable amounts for proxy and dynamic nodes to a null value.

Test 21a turned out to involve a couple of dynamic nodes, making the comparison meaningless. Test 02a simply missed to account for a proxy node corresponding to a run-time cosine entry point, and obtaining a better estimate (640 bytes) was easy after recompiling the library units with the new options.

All in all, we obtain a very sensible static worst case evaluation for all the relevant tests, with run-time observations/measurements equal or lower by only small factors.

3.4 On a simplified Ada parser

Here the goal was to evaluate the stack analysis framework on a moderately large single application, written in Ada and not designed with static analysis in mind although believed to be close to match the required criteria. The application is a simplified Ada parser without recursion, used for text colorization purposes in an IDE editor module. Unlike the previous set of testcases, this one is input sensitive and evaluating its worst case stack consumption with a testing based approach is not easy.

The first analysis attempts stumbled on dynamic allocations for common Ada constructs, easily rewritten in a more efficient manner.

The second difficulty was calls to the system heap memory allocator. It turned out to consume significant and not easily predictable amounts of stack on our platform, so we have replaced it with in-house Ada Storage_Pool.

Eventually, we found that indirect calls were used in numerous places and that properly accounting for them was a hard prospect. For the sake of the experiment, we assigned a null value to the fake indirect call proxy node and were then able to compute a worst case. Of course, the so computed value was expected not to be a reliable upper bound. Running an instrumented version of the parser over 2047 files from the GNAT source tree indeed revealed one case of measured stack consumption greater than the statically computed value. The other tests also provided interesting statistical figures: more than 60% of the tests revealed a measure within 98% of the computed value and the overwhelming majority of the remaining tests got measures above 80% of the computed value.

This experiment confirmed that static analysis of stack depths requires a lot of factors to allow the computation of reliable upper bounds, and that it should be anticipated early in the development process. It also shows that even when reliable upper bounds may not be obtained, performing the analysis is instructive and is a very helpful development instrument.

It is also interesting to notice that in none of our experiments to date, have we found a case of a computed max value unreasonably bigger than the measured values. Admittedly, our experiments have been limited and thus are not fully representative. Nonetheless, it is an encouraging result which would tend to indicate that refinements in the maximum computation are not immediately urgent.

4 Conclusion

Static analysis of stack allocation patterns is of great value in many situations and even a requirement in some specific software development processes. We have explained why a compiler can play a key role as a component in an analysis framework, and described two new GCC command line options implemented for this purpose. We obtained very encouraging results from several experiments with these options and a prototype analyzer to process the outputs, reporting on the worst case paths down provided entry points. As we have shown, the approach scales up well and is adaptable to new target architectures with only little efforts. It is already able to produce very valuable feedback early in the development process and still leaves a lot of room for refinements and extensions in the future.

References

[1] Absint - StackAnalyzer. `http://www.absint.com/stackanalyzer`.

[2] Survey Workshop on Resource Bound Checking. Part of the Research On Program Analysis System (ROPAS) initiative at the Seoul National Universtity— `http://ropas.kaist.ac.kr/survey`.

[3] The AVR Simulation and Analysis Framework. `http://compilers.cs.ucla.edu/avrora`.

[4] Visualization of Compiler Graphs (VCG). `http://rw4.cs.uni-sb.de/users/sander/html/gsvcg1.html`.

[5] Peter Altenbernd. On the False Path Problem in Hard Real-Time Programs. In *Proceedings of the 8th Euromicro Workshop on Real-Time Systems*, June 1996.

[6] Dennis Brylow. *Static Checking of Interrupt Driven Software*. PhD thesis, Purdue University, August 2003.

[7] Dennis Brylow, Niels Damgaard, and Jens Palsberg. Static Checking of Interrupt Driven Software. In IEEE Computer Society Press, editor, *Proceedings of the 23rd International Conference on Software Engineering*, May 2001.

[8] Krishnendu Chatterjee, Di Ma, Rupak Majumdar, Tian Zaho, Thomas Henzinger, and Jens Palsberg. Stack Size Analysis for Interrupt-Driven Programs. In Springer Verlag, editor, *Proceedings of the 10th International Static Analysis Symposium*, volume 2694 of *Lecture Notes in Computer Science*, June 2003.

[9] Karl Crary and Stephanie Weirich. Resource Bound Certification. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, 2000.

[10] Jens Palsberg and Di Ma. A Typed Interrupt Calculus. In Springer Verlag, editor, *Proceedings of the 7th International Symposium on Formal*

Techniques in Real-Time and Fault Tolerant Systems, volume 2469 of *Lecture Notes in Computer Science*, September 2002.

[11] John Reghr. Eliminating stack overflow by abstract interpretation. In Springer Verlag, editor, *Proceedings of the 3rd International Conference on Embedded Software*, volume 2855 of *Lecture Notes in Computer Science*, October 2003.

[12] John Reghr. Say No to Stack Overflow. September 2004. Article #47101892 at `http://www.embedded.com`.

[13] Leena Unnikrishnan, Scott D. Stoller, and Yanhong A. Liu. Automatic Accurate Stack Space and Heap Space Analysis for High-Level Languages. Technical Report 538, Computer Science Dept., Indiana University, April 2000.