SPARKSkein – A Formal and Fast Reference Implementation of Skein

Rod Chapman, Altran Praxis

Agenda

- The big idea…
- What is Skein?
- Coding SPARKSkein
- Results
- The release
- Conclusions and Further Work

The big idea…

- To produce a reference implementation of the Skein hash algorithm in SPARK
  - Make if Formal - Prove at least exception freedom (aka “type safety”).
  - Make it Readable.
  - Make it Portable – identical source code for all platforms, and no dependence on libraries, so suitable for low-level “bare machine” targets.
  - Make it Fast – well…at least as fast of the existing C reference implementation.

The big idea…

- And…Make it *empirical.* What does that mean?
  - From Bertrand Meyer's blog, 31st July 2010:
  - "Has the empirical side of software engineering become a full member of empirical sciences? One component of the experimental method is still not quite there: reproducibility. It is essential to the soundness of natural sciences; when you publish a result there, the expectation is that others will be able to replicate it."
- So…publish all sources, methods, results, and stick to freely available tools.
- Use the C implementation as a control experiment.

What is Skein?

- The US NIST is running a competition to find and standardize a new hash algorithm that will become “SHA-3”.
  - Five candidate algorithms remain in the third and final round of the competition.
  - “Skein” (it rhymes with “rain”) is one of them.

What is SPARK?

- SPARK is…
  - …a programming language – an *unambiguous* subset of Ada, with *contracts* for specification of partial correctness.
  - A *toolset* for static verification, including a VC-Generator and a theorem-prover.
  - A design philosophy for high-assurance software.
  - Overriding design goal: *soundness* of verification shall not be compromised.

Coding SPARKSkein

- Method:
  - Start with the Skein mathematical spec and the existing C reference implementation.
  - Understand both.

  - Re-code in SPARK following the same structure as the C.
    - Why?
      - Good chance of C readers being able to understand it.
      - Good chance of Skein's designers being able to understand it.
      - Good chance of SPARK performance being close to that of the C code to start with.

Coding SPARKSkein

- Observations on the Coding
  - Pretty easy really.
  - Ada's Interfaces package is really useful.
  - Lots of modular types (e.g. mod 2^{64}) and shifting, rotating, and "xor" operations, all of which are *very* efficient in SPARK.
    - For example, Interfaces.Shift_Left_64 is an *intrinsic* function call that emits *one machine instruction* using GCC.

- One tricky bit – making the code endian-ness independent.
  - Skein is designed to be very efficient on little-endian machines – most notably Intel x86 and x86_64.
  - BUT..the code needs to work just the same on a big-endian machine.

- SPARK isolates us from this, since the operations on types are defined *mathematically*, not in terms of the representation.

Results

- Results arise from five activities:
  - Static Analysis and Proof of type safety
  - Testing against reference test vectors
  - Portability testing
  - Structural coverage
  - Performance

Static Analysis and Proof

- All code is 100% SPARK and analyses with SPARK GPL 2011 Edition toolset with no warnings or errors.

- Proof metrics

```
Total VCs by type:
                            -----------Proved By Or Using------------
                     Total  Examiner Simp(U/R)  Checker Review False Undiscgd
Assert or Post:         65      22      35              8      0     0        0
Precondition check:     21       0      12              9      0     0        0
Check statement:        31       0      26              5      0     0        0
Runtime check:         244       0     243(   2)        1      0     0        0
Refinement VCs:          6       2       4(   4)        0      0     0        0
Inheritance VCs:         0       0       0              0      0     0        0
================================================================================
Totals:                367      24     320(   6)       23      0     0        0
% Totals:                       7%     87%(   2%)       6%     0%    0%       0%
===================== End of Semantic Analysis Summary ========================
```

Static Analysis and Proof

- 344 VCs proved automatically (93.7%) – not too bad given significant usage of modular types and arithmetic.

- Remaining 23 proved in the Checker.
  - These were hard…
  - Integer inequalities involving "mod 2^{64}" and integer (truncating) division all over the place.
  - Finding the "just right" loop invariant was very hard for some of the algorithms.

Prover says No – a bug is found!

- During development of the "Finalization" algorithm, something interesting popped up.

- Skein has a configurable hash size – you initialize the algorithm with a "hash bit length" – how many bits of output you want.

- The Finalization algorithm converts this bit length into a number of bytes required for output.

Prover says No – a bug is found!

- Here's the offending bit of code:

```
Byte_Count := (Hash_Bit_Len + 7) / 8;
```

- Where the "+" operator is "mod 2^{64}" and the "/" operator is integer division (rounding down toward zero).

- This was basically copied direct from the C code…

- This is followed by a loop that iterates to generate the required numbers of blocks of output.

Prover says No – a bug is found!

- This loop *has* to iterate at least once, otherwise *no* output would be produced. In SPARK, this came out as a later VC that tries to establish:

```
Hash_Bit_Len >= 0 and
Hash_Bit_Len <= 2^64 - 1
 ->
((Hash_Bit_Len + 7) mod 2^64) / 8 > 0 .
```

- Which the Simplifier refused to prove....
- ...mainly because it isn't True.

Prover says No – a bug is found!

- How come?

- If `Hash_Bit_Len` is very large (nearly 2^{64}), then the “+ 7” overflows round to a small number near 0, which divided by 8 *is zero*. Oh dear!

- Result: If you ask for nearly 2^{64} bits of output, the C code returns immediately, and returns a pointer to an *arbitrary* block of memory…Subsequent behavious is *undefined.*

- Of course…. “no one would ask for that much output…” would they?

Prover says No – a bug is found!

- Solution in SPARKSkein…

```
subtype Hash_Bit_Length is U64 range 0 .. U64'Last - 7;
```

- Subtype declarations in SPARK act like simple type-invariants.

Results – Reference Test Vectors

- The Skein spec defined 3 test vectors for the 512-bit block version of the algorithm – known data blocks with knows hashes.

- Initial test failed…

- Why? One mis-typed rotation constant had value "34" instead of "43".
  - After that corrected, all is well…

- Moral: even type-safe code isn't necessarily correct code.

Results – Portability

- Code submitted to AdaCore for inclusion in their mighty GCC testsuite. Runs every night on all the platforms that they support.

- Target architectures and operating systems include
  - 32-bit x86 (Windows, Linux, FreeBSD, and Solaris), x86_64 (Windows, Linux, Darwin), SPARC (32- and 64-bit Solaris), HP-PA (HP Unix), MIPS (Irix), IA64 (HP Unix, Linux), PowerPC (AIX), Alpha (Tru64).

- Result: it works.

Results – Coverage

- I wrote a single test program to exercise various scenarios – short data blocks, medium blocks, long blocks, sequences thereof etc. etc.

- Result: 99.7% statement coverage, with ONE uncovered line of code that turned out to be a type declaration that has no object code associated with it.

- Conclusion: false alarm in gcov. No worries.

Results – Performance

- Now the real fun started…

- Could it possibly be as fast as the C?

- Conjecture:
  - “Proven type-safe” SPARK code ought to be fast.
  - No aliasing, no function side-effects, aggressive inlining, turn off all run-time checks…optimizers should be able to do *better* with SPARK than C.

  - Is this True?

Results – Performance

- Method
  - The C reference implementation comes with a performance testing program.
  - Therefore – write exactly the same program in SPARK to test the performance of the SPARK code in the same way, running the same test.
  - Test machine: Intel Core i7 860 @ 2.8 GHz, running 64-bit GNU/Linux.
  - Use the *same* compiler for both languages. Initially, we used:
    - GNAT Pro 6.3.2 (GCC 4.3.5)

      and
    - GNAT Pro 6.4.0w (GCC 4.5) for same platform

      To see if GCC 4.5 makes any difference.

Results – Performance

- Method
  - Experiment with different GCC- and SPARK-specific compiler options to see what happens.
  - -O[0|1|2|3] – optimization level.
  - -gnato – *enable* full Ada runtime checks including overflow check.
  - -gnatp – *disable all* Ada runtime checks (like default in C).
  - -gnatn – enable inlining at –O1 and above.

Results – Performance

Compiler: GNAT Pro 6.3.2 (GCC 4.3.5) Clocks per byte hashed (Lower numbers are better)		
Options	**SPARK**	**C**
-O0 -gnato	213.9	N/A
-O0 -gnatp	207.9	172.3
-O1 -gnatp	27.6	37.7
-O1 -gnatp -gnatn	26.8	37.7
-O2 -gnatp -gnatn	25.5	24.7
-O3 -gnatp -gnatn	20.4	20.1

Results – Performance

Compiler: GNAT Pro 6.4.0w, built 28th July 2010		
Options	**SPARK**	**C**
-O0 -gnato	71.1	N/A
-O0 -gnatp	69.9	96.5
-O1 -gnatp	22.2	37.0
-O1 -gnatp -gnatn	20.7	37.0
-O2 -gnatp -gnatn	20.2	19.7
-O3 -gnatp -gnatn	13.4	12.3

Results – Performance

- Bottom line – GCC 4.3.5
  - At –O0 both languages are awful with SPARK trailing C owing to full runtime checking. This is expected – GCC at –O0 is “deliberately bad”.
  - At –O1, SPARK is much better than C. Better (and earlier) inlining mostly responsible for this.
  - At –O2, C leads by a little.
  - At –O3, auto loop unrolling gives another performance boost to both languages, with C still leading by a little, owing to slightly better optimization of partial redundancies, dead-store elimination, and other nerdy optimizer stuff.
  - The difference lies in the relative “optimizer friendliness” of the intermediate language generated by the Ada and C front-ends.

Results – Performance

- Bottom line – GCC 4.5.0
  - Big improvement across the board for both languages.
  - Same pattern, except at –O0 where SPARK leads now.

Results – Performance

- Improving GCC 4.5
- Based on this analysis, Eric Botcazou of AdaCore improved the Ada “middle-end” in GCC to produce more “optimizer-friendly” intermediate language.
- These improvements are included in GNAT Pro 6.4.1 and GCC 4.5.2 and beyond.

Results – Performance

Compiler: GNAT Pro 6.4.1 (GCC 4.5.2)		
Options	**SPARK**	**C**
-O0 -gnato	70.6	N/A
-O0 -gnatp	69.7	96.4
-O1 -gnatp	22.2	37.0
-O1 -gnatp -gnatn	20.5	37.0
-O2 -gnatp -gnatn	20.0	19.7
-O3 -gnatp -gnatn	12.3	12.3

Results – Performance

- With GNAT Pro 6.4.1:
  - At –O0 – SPARK is better
  - At –O1 – SPARK is better
  - At –O2 – C is (slightly) better
  - At –O3 – identical performance

- This trend has been observed many times before: GCC development tends to be driven by “the masses” (i.e. C users!). Ada and SPARK performance catch up one or two generations later.

The Release

- Check out www.skein-hash.info
- Download the whole thing – sources, test cases, proofs – the lot.
- All results are reproducible using the GPL 2011 Editions of GNAT and SPARK Toolsets.

Conclusions and Further Work

- Well…it worked.

- Formal – Yes…
- Readable – Well…I think so…
- Portable – Yes…
- Fast – As good as we could have expected…
- Empirical – Yes…

Conclusions and Further Work

- Further work - SPARK:
  - One procedure takes *an hour* to prove on the test machine. Definite Simplifier problem here. Work on-going to fix this.
  - Several other Simplifier improvements identified.
  - Several Proof Checker improvements identified.

Conclusions and Further Work

- Further work - Proof:
  - Re-prove all VCs using SMT-based provers, such as Z3 or Yices. Initial results look good.
  - Z3 can prove *all* 23 VCs where we had to use the Checker.
  - BUT..this only works *after* you've toiled to find the "just right" loop invariants, so not a free lunch.
  - Automated help in finding (non-linear) loop-invariants is sorely missing in SPARK right now. Help please!

Conclusions and Further Work

- Further work – SHA-3:
  - Those with C tools – please verify the C reference implementations…
  - Other SHA-3 candidates
    - Repeat the experiment for the other "final five" SHA-3 candidate algorithms.
    - How many bugs will we find?
      - (Student project anyone?)

Altran Praxis Limited
20 Manvers Street
Bath BA1 1PX
United Kingdom
Telephone: +44 (0) 1225 466991
Facsimile: +44 (0) 1225 469006
Website: www.altran-praxis.com

Email: rod.chapman@altran-praxis.com