LARA Tutorial

3. Programming Strategies for Code Transformations and Optimizations

Ricardo Nobre, Tiago Carvalho, Pedro Pinto, João Bispo, Luís Reis, and João M.P. Cardoso

University of Porto, FEUP, Porto, Portugal

April 20th, 2016

U.PORTO FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

ANTAREX

SPeCS Special-Purpose Computing Systems, languages and tools

LARA for Programming Strategies

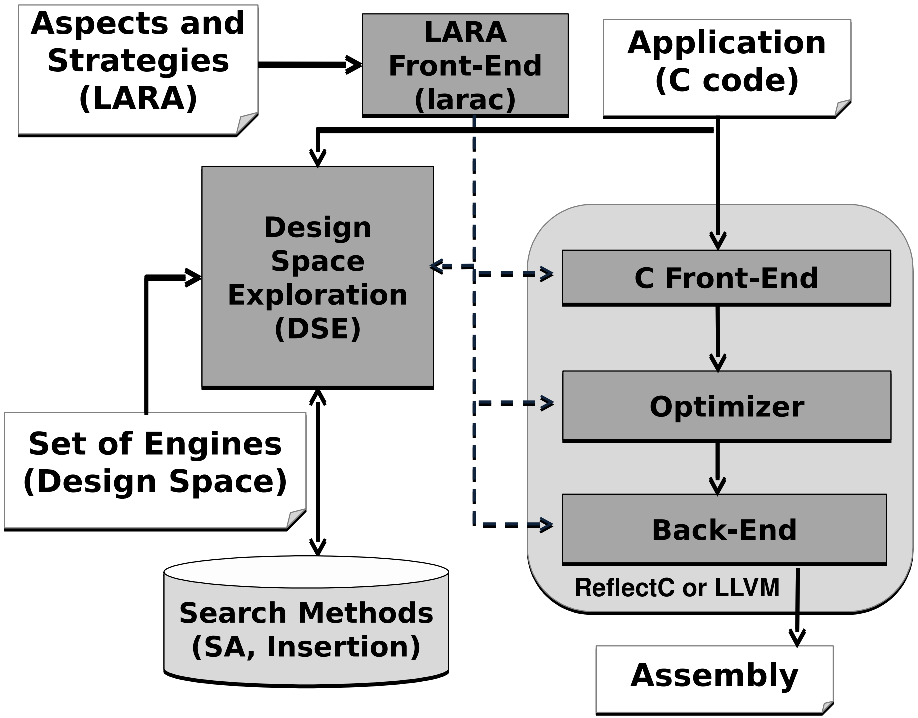
- Specialized compilation strategies can be a way to comply with stringent/strict requirements
- Can be accomplished at two distinct levels using our LARA-based toolchain
- At the level of the IR
  - Individual optimizations can be executed with specific parameters over selected points of interest in the program/function
- At the level of the tools
  - Tools can be called and exchange information in an order instructed by LARA aspects (LARA outer-loop)

LARA IR-level Control Example

```
aspectdef LoopUnroll
    select loop end
    apply
        if($loop.num_iterations <= 32) {
            $loop.exec Unroll(0);
        } else {
            $loop.exec Unroll(2);
        }
    end
    condition
        $loop.is_innermost &&
        $loop.type=="for"
    end
end
```

- Selects every loop in the program
- Loops with less than 32 iterations:
  - Are fully unrolled
  - Uses a factor of 2 otherwise
- Applies transformation if loop:
  - is innermost
  - is a FOR loop
- More sophisticated analyses/strategies are possible:
  - Using attributes
  - JavaScript code

LARA-based DSE Tool Flow

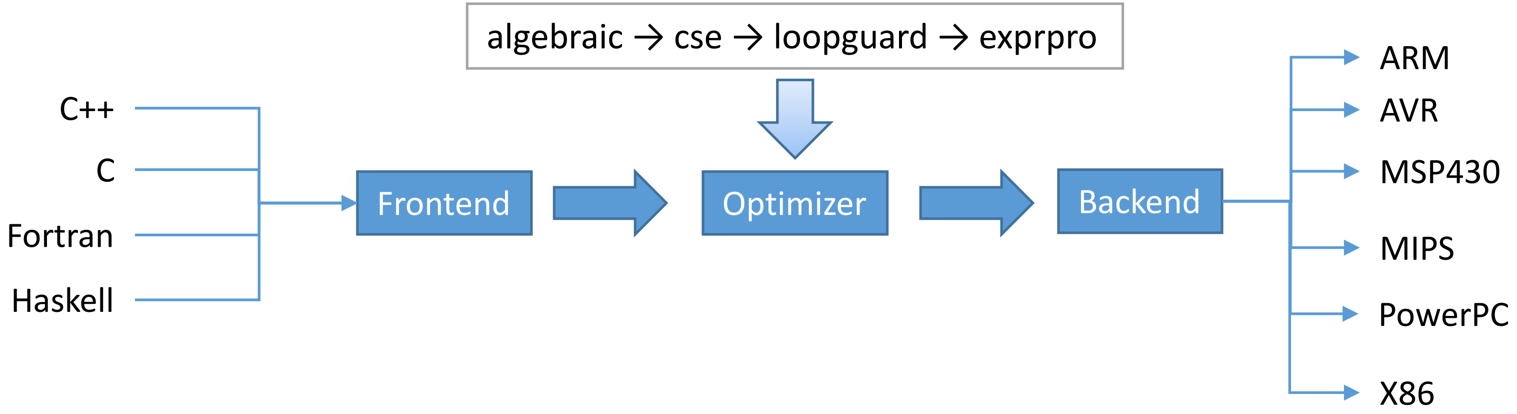


- Multi-metric, multi-target, multi-compiler and multi-algorithm modular compilation flow with support for DSE
- DSE algorithms can be reused (without any modification) between different target architectures, metrics and compilers.

Finding Better Compiler Sequences

- **Phase order:** Set of analysis/optimization/lowering compiler passes executed in a given order between the frontend and the code generator

- Example:



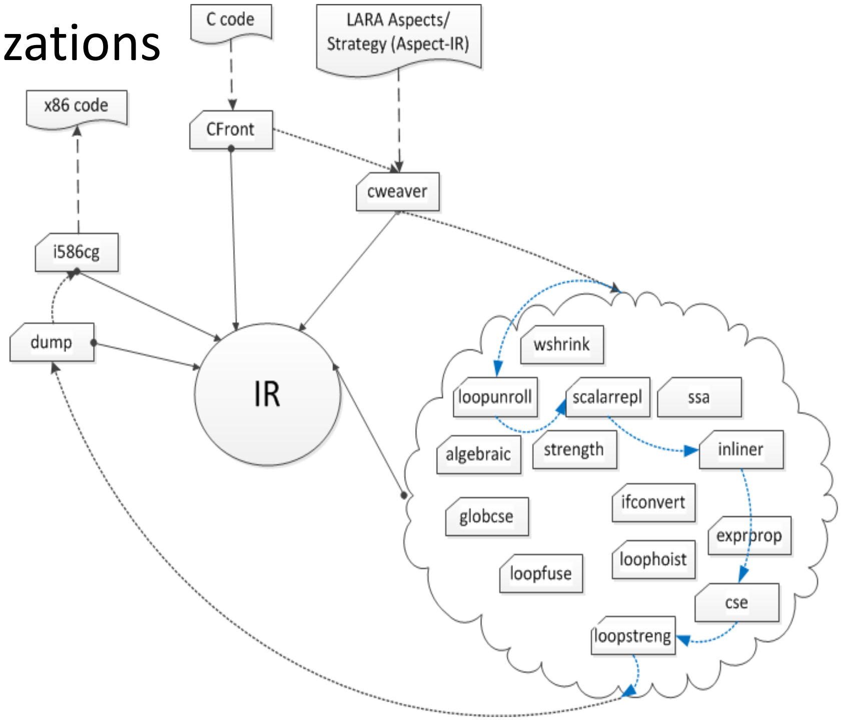
Advantages of Phase Selection/Ordering

- Optimization is typically achieved using available optimization flags
  - E.g., Clang/LLVM –O1, -O2, or –O3
- -OX flags represent fixed compiler sequences
  - No fixed compiler sequence can result in the best possible output
  - Potential for optimization trough specialization
- -Ox flags are typically only tuned for performance
  - Other metrics can be important (e.g., energy, code size)

LARA Actions: Seq. of Compiler Opt.

- Specify sequences of compiler optimizations

```
aspectdef optimizationseq
    select function end
    apply
        exec loopinvariant();
        exec loopscalar();
        exec dismemun();
        exec loopstrength();
        exec strength();
        exec looprev();
        exec lowerboolval();
        exec loopbcount();
    end
end
```



How to search phase orders?

- Traditional / Manual approach
  - Requires deep knowledge about correlation between code features, target architecture and compiler passes interdependence
- Automatic approach
  - **ML-based:** Predictive model suggests compiler sequence based on function/program features
    - (+) Fast (-) Less optimization potential
  - **Iterative:** More than one solution is tested in an effort to find the most suitable sequence (e.g., random search, GAs, Simulated Annealing)
    - (+) Higher optimization potential (-) Slower

Phase Selection/Ordering with Clang/LLVM

- Target: **LEON3**
  - SPARC V8 7-stage processor w/FPU
- Compiler: **Clang/LLVM 3.7**
  - 140 compiler passes (up to 128 per sequence)
- Texas Instruments DSP (21) and IMG (21) programs
- DSE algorithms executed for different numbers of iterations:
  - 10; 100; 1,000; 10,000; 100,000
- Target Metric: **Performance**
  - DSE framework supports the definition of other metrics (e.g. energy).

Texas Intruments Kernels

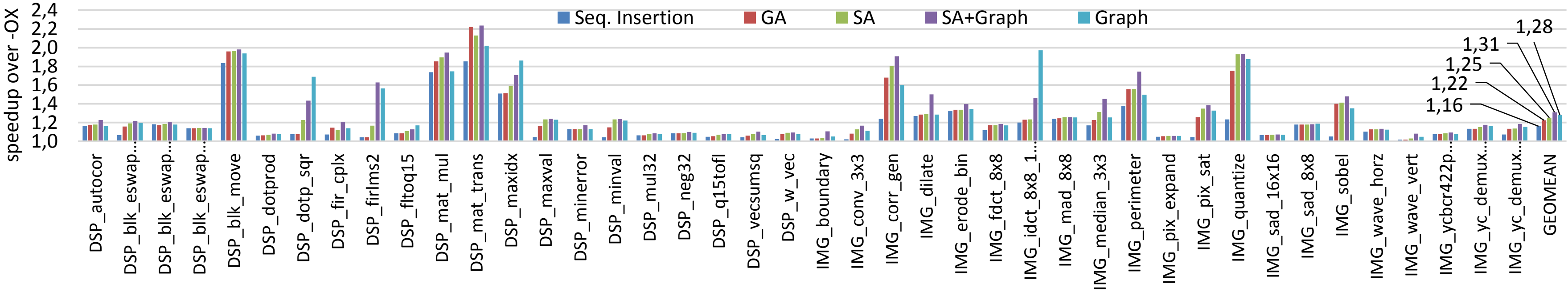
Func. Name	Description	CLOC
DSP_autocor	Autocorrelation of an input vector.	11
DSP_blk_eswap16	Endian-swap a block of 16-bit values.	27
DSP_blk_eswap32	Endian-swap a block of 32-bit values.	31
DSP_blk_eswap64	Endian-swap a block of 64-bit values.	39
DSP_blk_move	Move block of memory.	13
DSP_dotprod	Vector product of two input arrays.	7
DSP_dotp_sqr	Dot product of two arrays.	17
DSP_fir_cplx	Complex FIR.	24
DSP_firlms2	Least Mean Square Adaptive Filter.	17
DSP_fltoq15	Convert IEEE FP into Q.15 format.	16
DSP_mat_mul	Matrix Multiply.	19
DSP_mat_trans	Transposes a matrix of 16-bit values.	8
DSP_maxidx	Finds the largest element in an array.	12
DSP_maxval	Finds the maximum value of a vector.	9
DSP_minerror	Minimum Energy Error Search.	23
DSP_minval	Finds the minimum value of a vector.	9
DSP_mul32	32-bit multiply.	25
DSP_neg32	32-bit vector negate.	11
DSP_q15tofl	Q.15 to IEEE float conversion.	6
DSP_vecsumsq	Sum of squares.	15
DSP_w_vec	Weighted vector sum.	13

Func. Name	Description	CLOC
IMG_boundary	Returns coordinates of boundary pixels.	17
IMG_conv_3x3	3x3 convolution.	42
IMG_corr_gen	Generalized correlation with 1xM tap filter.	16
IMG_dilate_bin	3x3 binary dilation.	47
IMG_erode_bin	3x3 binary erosion.	47
IMG_fdct_8x8	8x8 Block FDCT With Rounding.	116
IMG_idct_8x8_12q4	IEEE-1180/1990 Compliant IDCT.	121
IMG_mad_8x8	8x8 block Minimum Absolute Difference.	30
IMG_median_3x3	3x3 median filter on 8-bit unsigned values.	43
IMG_perimeter	Returns the boundary pixels of an image.	31
IMG_pix_expand	8-bit unsigned to 16-bit array.	11
IMG_pix_sat	16 bit signed numbers to 8 bit unsigned.	23
IMG_quantize	Matrix Quantization w/ Rounding.	27
IMG_sad_16x16	16x16 Sum of Absolute Differences.	14
IMG_sad_8x8	8x8 Sum of Absolute Differences.	14
IMG_sobel	Sobel filter.	27
IMG_wave_horz	Orthogonal Wavelet decomposition.	25
IMG_wave_vert	Compute vertical wavelet transform.	27
IMG_ycbcr422p_rgb565	YCbCr 4:2:2/4:2:0 to 16-bit RGB 5:6:5.	61
IMG_yc_demux_be16	De-interleave 4:2:2 BIG ENDIAN stream into LITTLE ENDIAN 16-bit planes.	18
IMG_yc_demux_le16	De-interleave 4:2:2 LITTLE ENDIAN stream into BIG ENDIAN 16-bit planes.	18

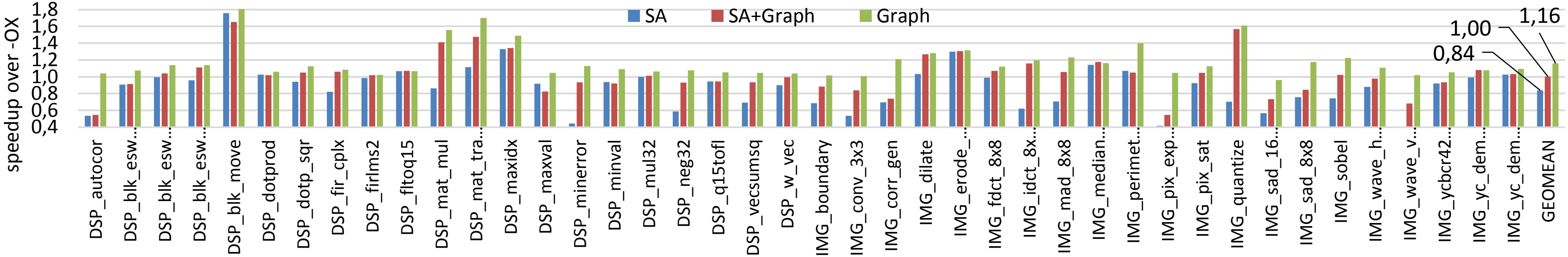
LLVM Passes for Exploration

-aa-eval	-consthoist	-float2int	-lazy-value-info	-loop-vectorize	-objc-arc-expand	-scev-aa
-adce	-constmerge	-functionattrs	-lcssa	-loops	-pa-eval	-scoped-noalias
-add-dis.	-constprop	-globaldce	-libcall-aa	-lower-expect	-part.-inliner	-s.-c.-o.-f.-gep
-alig.-f.-ass.	-correlated-prop.	-globalopt	-licm	-loweratomic	-part.-inl.-libcal.	-simplifycfg
-alloca-hoisting	-cost-model	-globalsmodref-aa	-lint	-lowerbitsets	-pl.-ba.-safe.-im.	-sink
-always-inline	-count-aa	-gvn	-load-combine	-lowerinvoke	-place-safep.	-slp-vectorizer
-argpromotion	-da	-indvars	-loop-accesses	-lowerswitch	-postdomtree	-slsr
-ass.-cache-track.	-dce	-inline	-loop-deletion	-mem2reg	-prune-eh	-spec.-execution
-atomic-expand	-deadargelim	-inline-cost	-loop-distribute	-memcpyopt	-reassociate	-sroa
-barrier	-debug-aa	-instcombine	-loop-extract	-memdep	-reg2mem	-strip
-basicaa	-delinearize	-instcount	-loop-ex.-single	-mergefunc	-regions	-str.-dead-d.-info
-basiccg	-die	-instnamer	-loop-idiom	-mergereturn	-rewr.-sta.-for-gc	-str.-d.-proto.
-bb-vectorize	-divergence	-instrprof	-loop-instsimpl.	-mldst-motion	-rewrite-symbols	-strip-d.-declare
-bdce	-domfrontier	-instsimplify	-loop-interchan.	-mod.-debuginfo	-safe-stack	-strip-nondebug
-block-freq	-domtree	-intervals	-loop-reduce	-nary-reass.	-sancov	-structurizecfg
-bounds-checking	-dse	-ipconstprop	-loop-reroll	-no-aa	-scalar-evolution	-tailcallelim
-branch-prob	-early-cse	-ipsccp	-loop-rotate	-objc-arc	-scalarizer	-targetlibinfo
-break-crit-edg.	-elim-avail-ext.	-irce	-loop-simplify	-objc-arc-aa	-scalarrepl	-tbaa
-cfl-aa	-extract-blocks	-iv-users	-loop-unroll	-objc-arc-apelim	-scalarrepl-ssa	-tti
-codegenprepare	-flattencfg	-jump-threading	-loop-unswitch	-objc-arc-contrac.	-sccp	-verify

Individual Function Speedups

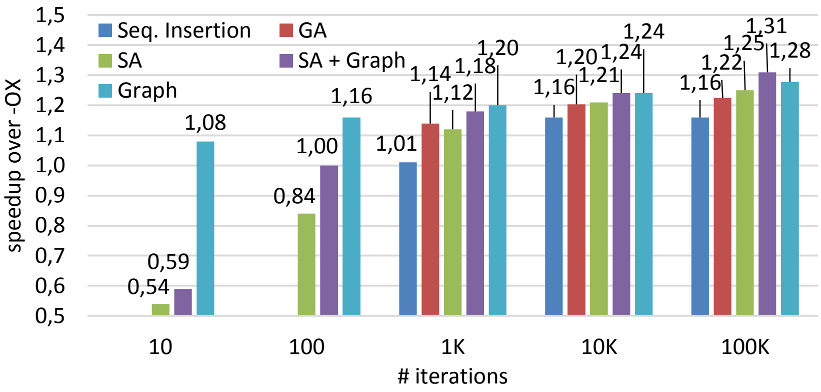


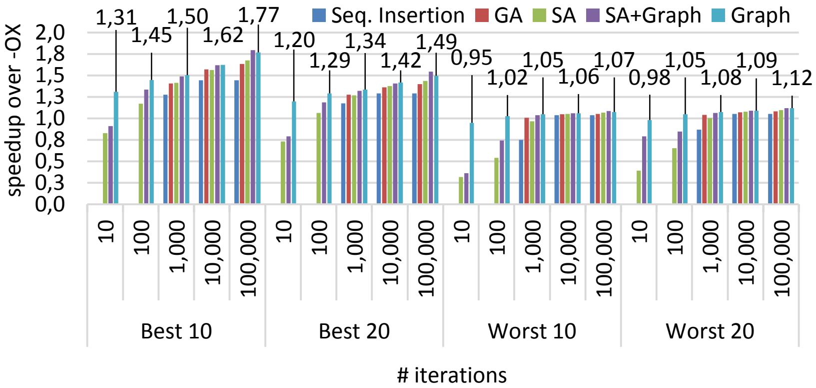
100,000 compilations/simulations



100 compilations/simulations

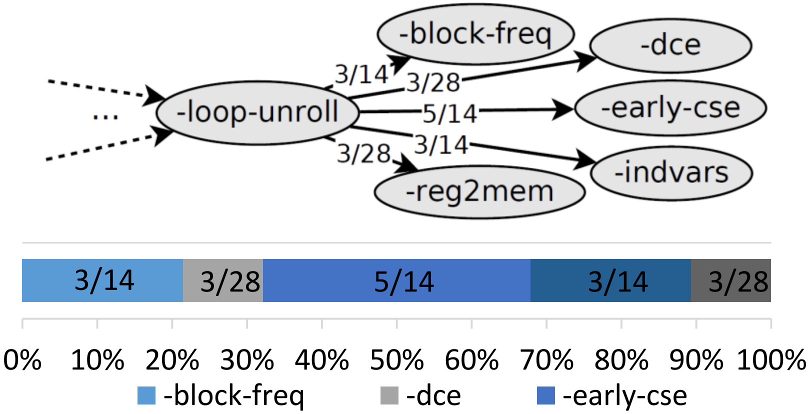
Geomean Speedups



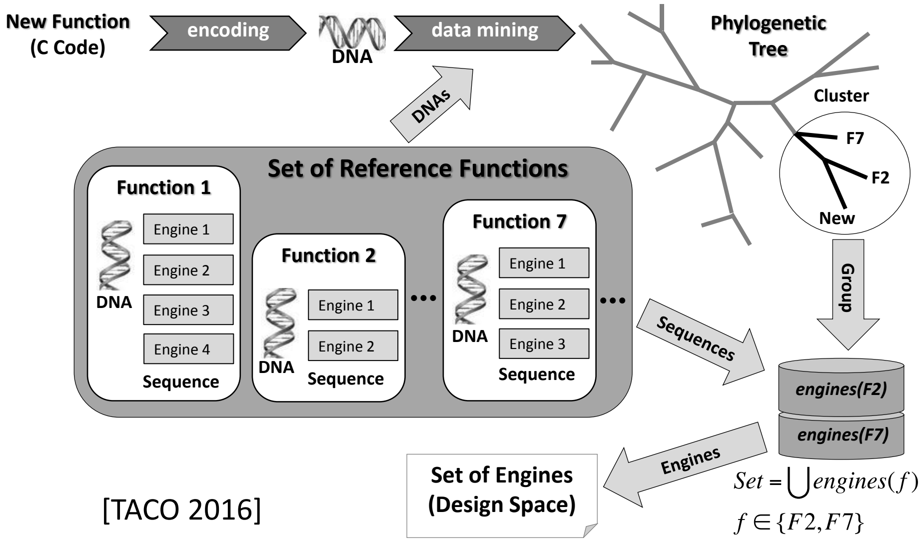


- *Graph*-based method is a good tradeoff between very good performance for high num. iterations and best performance for low num. iterations (1,000 or less)
- *SA+Graph* better for more that 10,000 iterations

Pass Selection and Ordering Approaches



[LCTES 2016]



[TACO 2016]

- Graph previously generated using historically suitable compiler sequences and/or rules, is iterated when generating new sequences.
- Clustering to reduce the number of compiler passes do consider for exploration based on input code.

Ongoing Work

- LLVM IR-level integration with LARA (as with the CoSy-based compiler)
  - Support for fine-tuning compiler pass behavior
    - Where to apply optimization
      - e.g., “select function{name==fname}.loop{is_innermost==true}”
    - Expose existing/new parameters
      - e.g. “exec loopunroll(k:2)”
- Explore metrics in the context of HW synthesis (using LegUp)
  - E.g., area, frequency
- Develop and test new DSE methods
  - E.g., using neural networks, markov models, or others

Takeaway Points

The LARA tool flow provides:

- The specification of strategies for code instrumentation and synthesis/compiler optimizations
- Fully exploration of compiler optimizations (req. integration at IR-level)
- Mechanisms to find and apply the most suitable compiler sequence according to **code**, target **architecture** and **requirements**
  - Current version contributed to research achievements with phase ordering
- Advanced ways to control and customize tool flow

Thank you! Questions?

U.PORTO
FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

ANTAREX

SPeCS Special-Purpose Computing Systems, languages and tools