Compiling for Automatically Generated Instruction Set Extensions

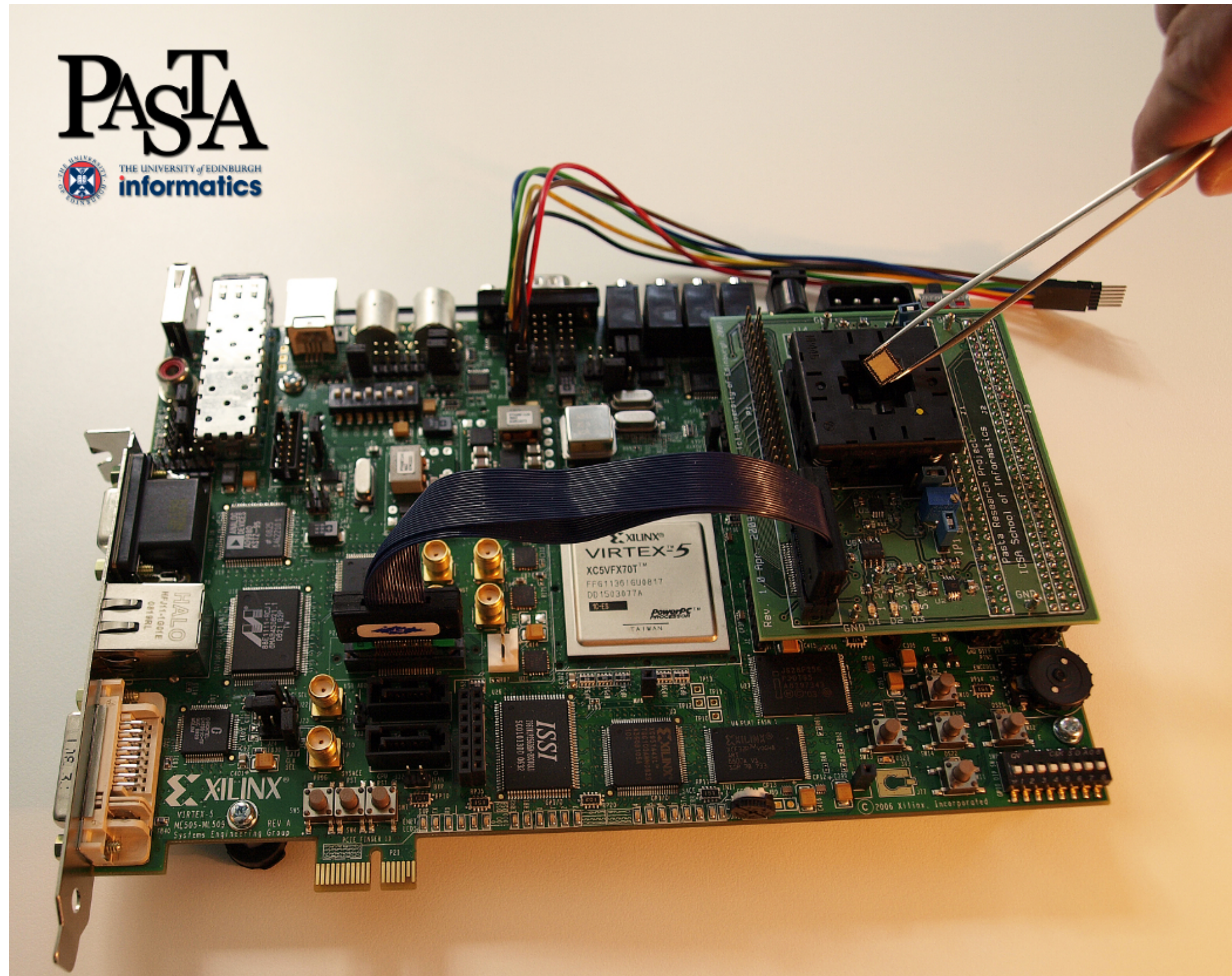
Alastair Murray and Björn Franke
Institute for Computing Systems Architecture,
School of Informatics, University of Edinburgh

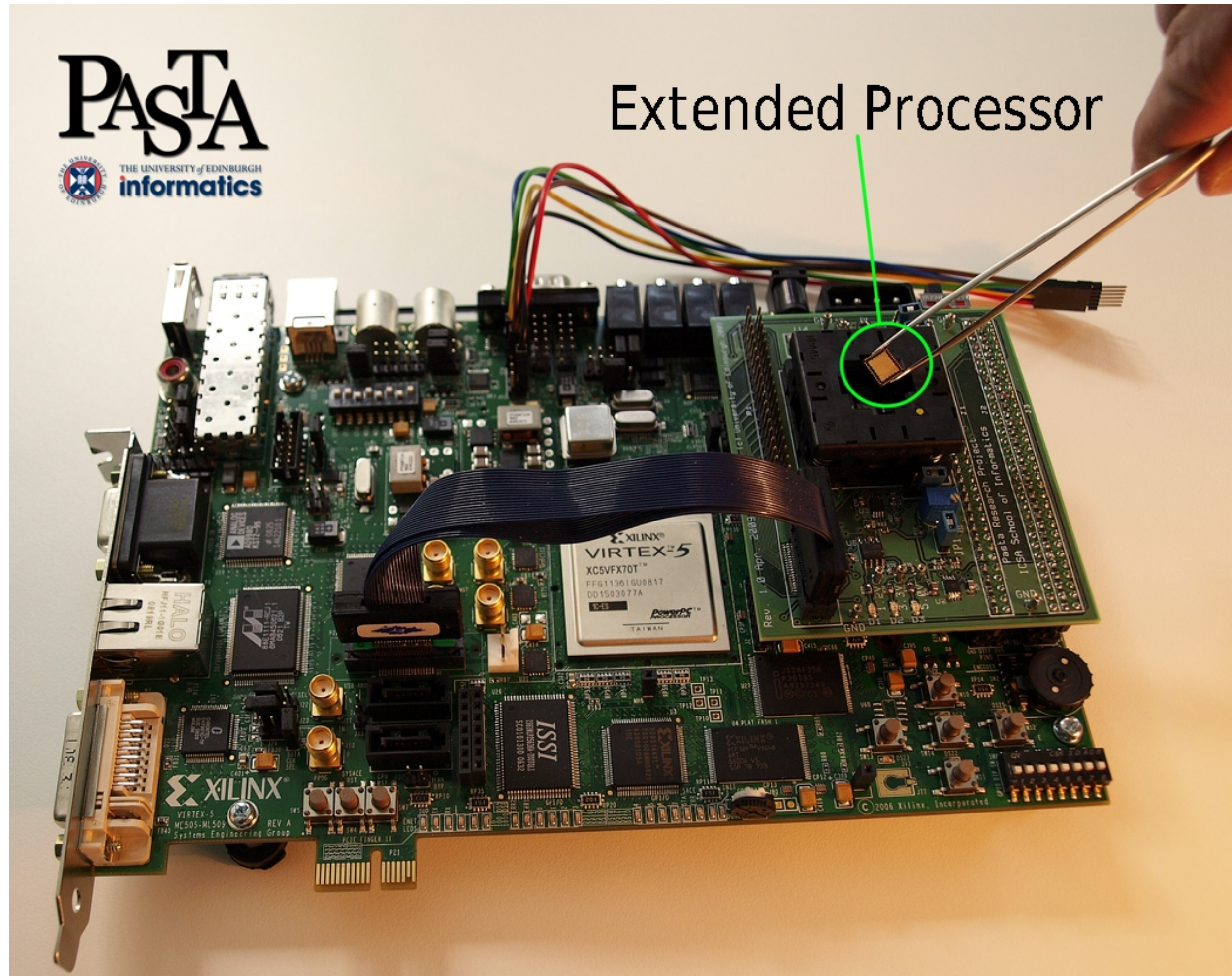
CGO '12: April 2nd, 2012

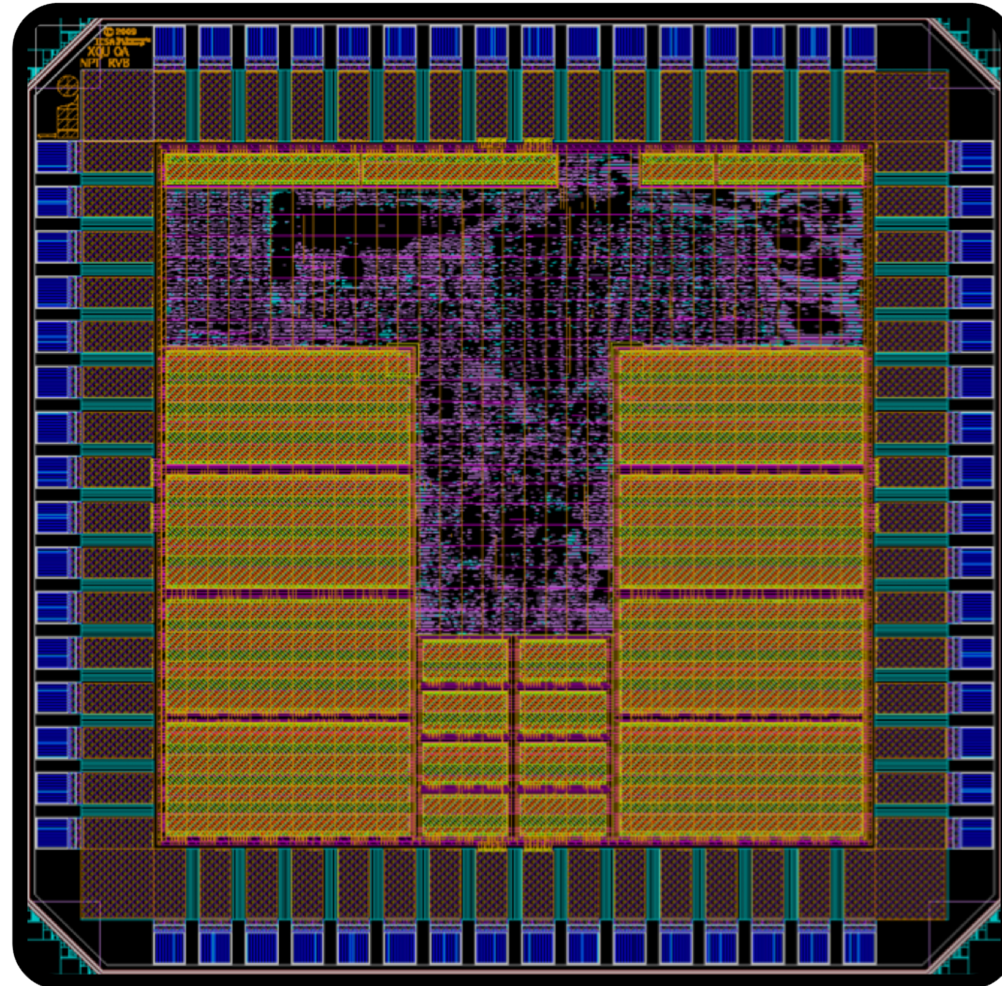
PASTA

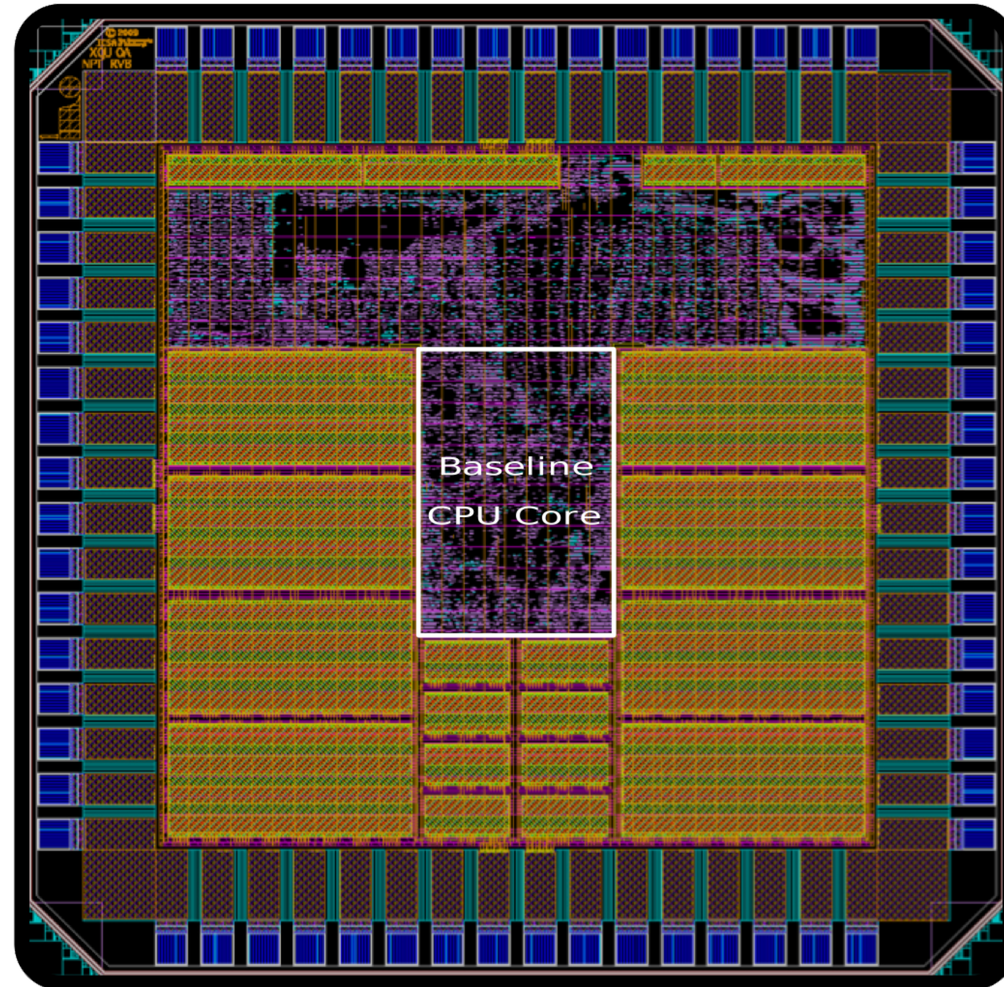
Automated Processor Synthesis:

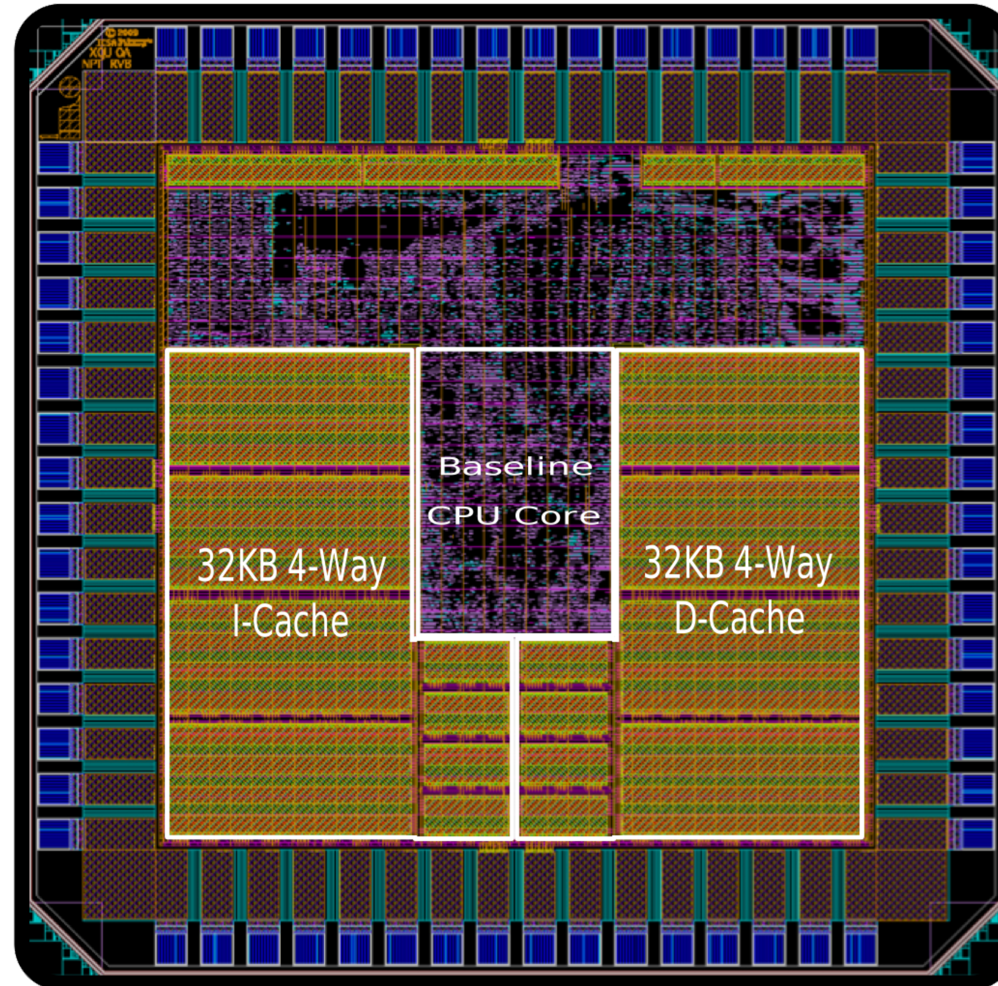
- Targeted at the embedded domain.
- Automatically generate an application specific processor.
  - Extend a baseline processor.
- Automatically find specialised extension instructions.
- Automatically generate a simulator and compiler for this processor.

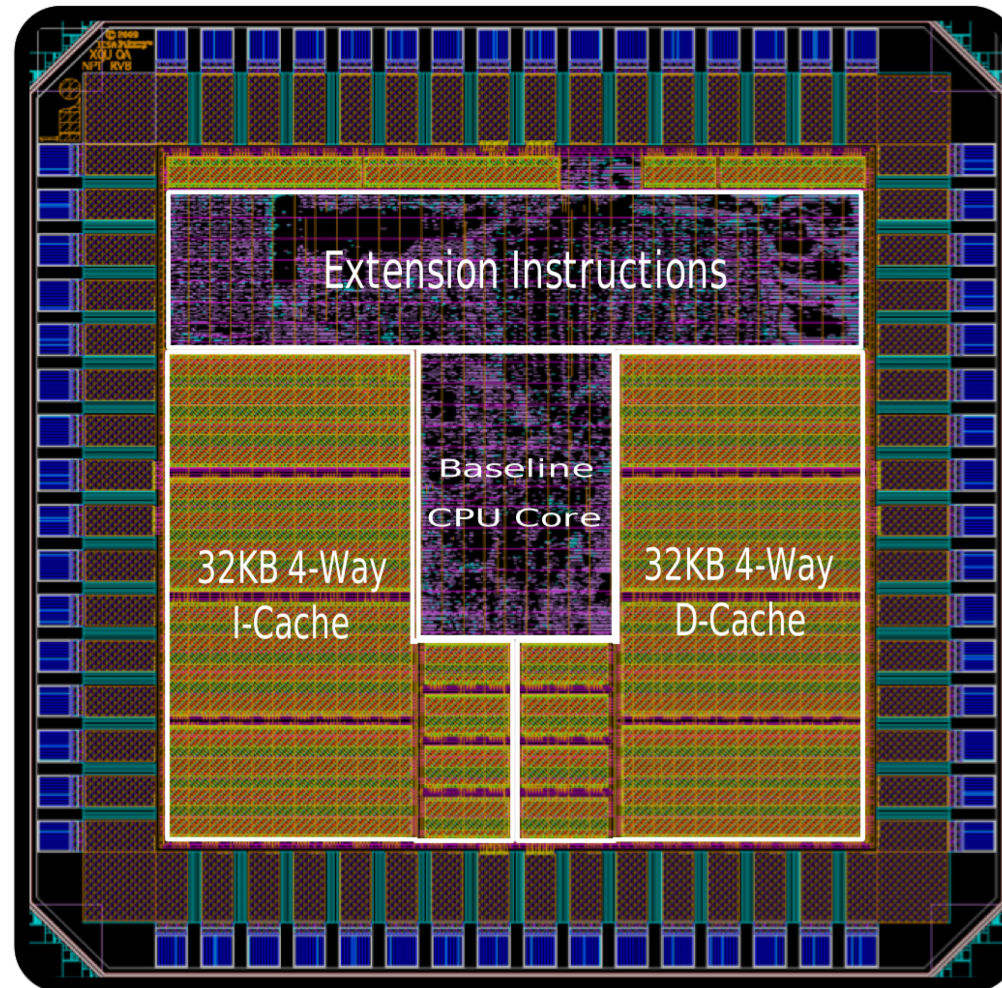

PASTA
THE UNIVERSITY of EDINBURGH
informatics
XILINX
VIRTEX-5
XC5VFX70T
ISSI
XILINX

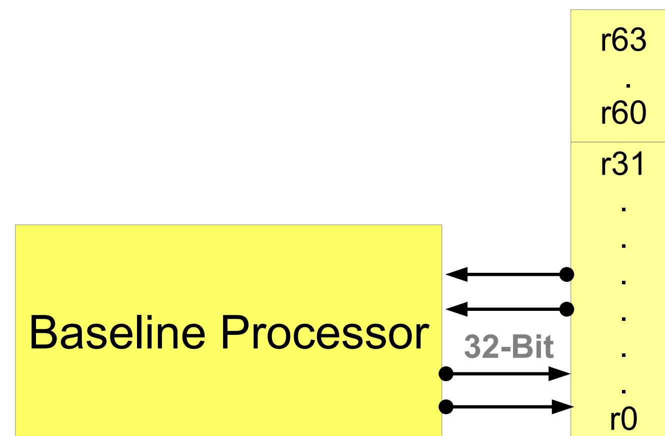

PASTA
THE UNIVERSITY of EDINBURGH
informatics
Extended Processor
XILINX
VIRTEX-5



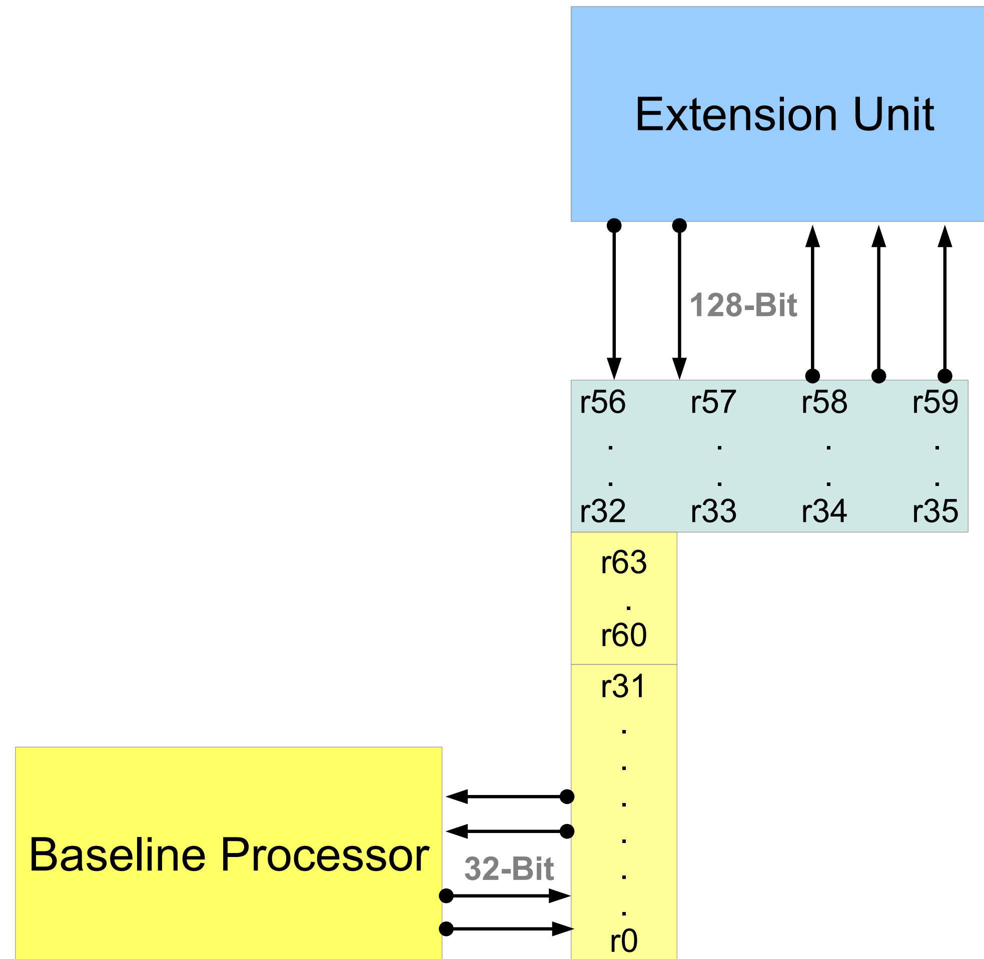

Baseline
CPU Core

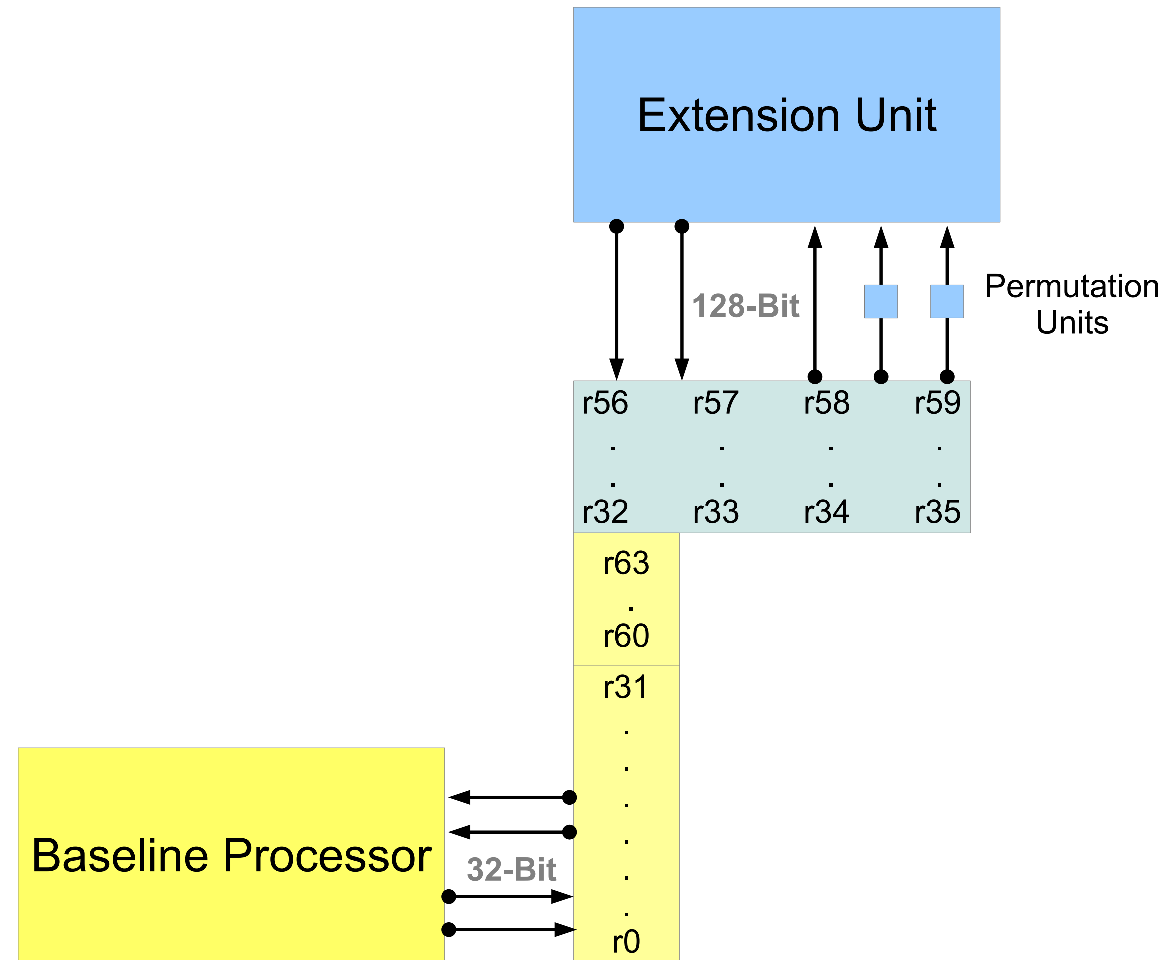

Baseline
CPU Core
32KB 4-Way
I-Cache
32KB 4-Way
D-Cache


Extension Instructions
Baseline
CPU Core
32KB 4-Way
I-Cache
32KB 4-Way
D-Cache

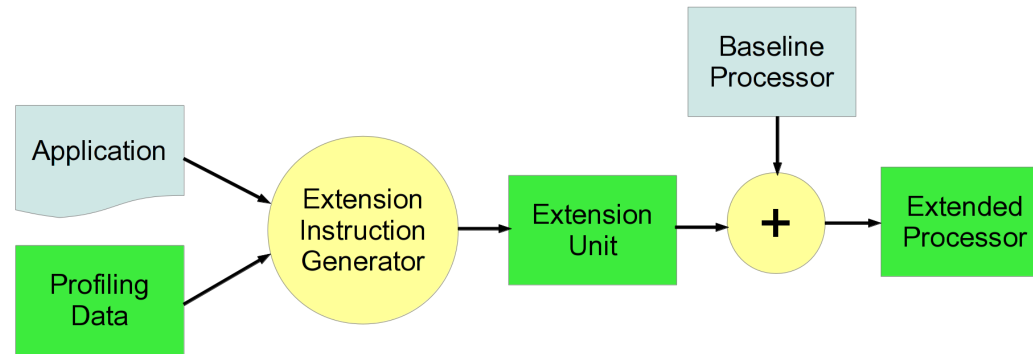

r63
.
r60
r31
.
.
.
.
.
.
r0
Baseline Processor
32-Bit

School of informatics

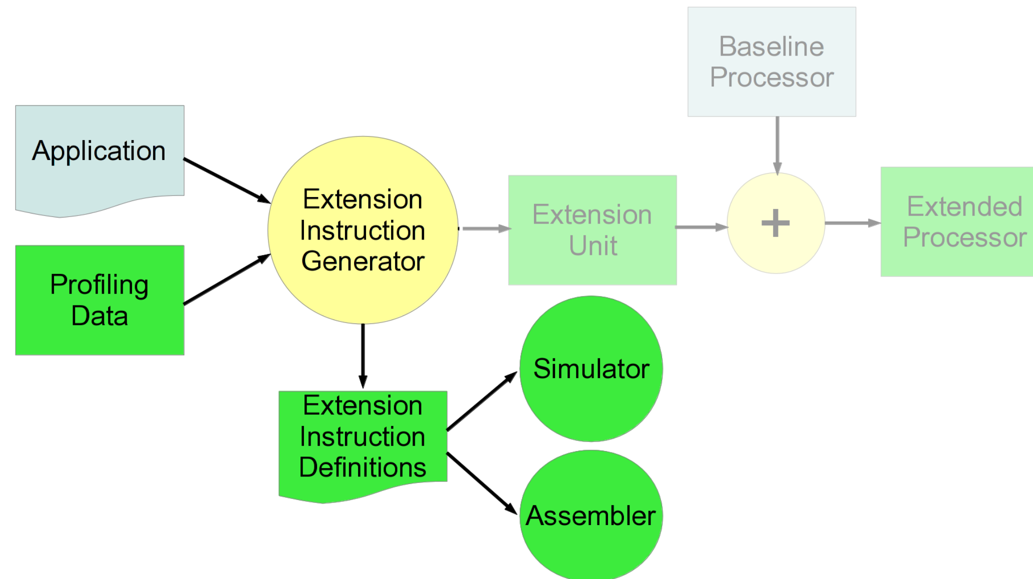



Extension Unit
128-Bit
Permutation Units
r56 r57 r58 r59
r32 r33 r34 r35
r63
r60
r31
r0
Baseline Processor
32-Bit

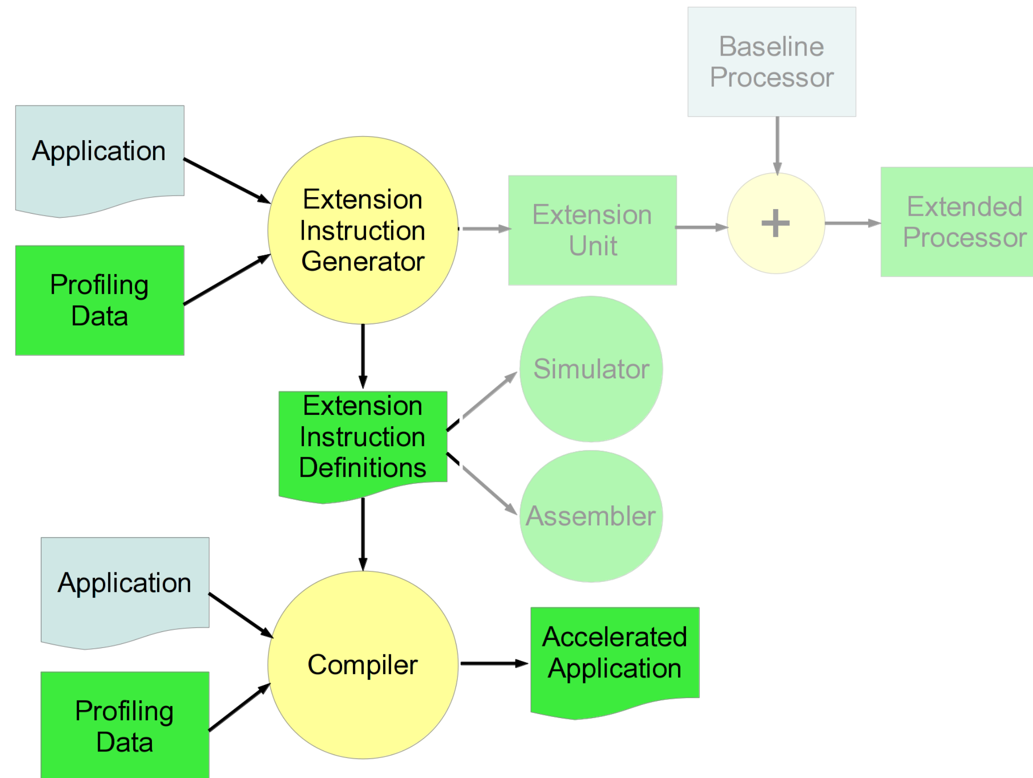
PASTA Work-flow



PASTA Work-flow



PASTA Work-flow

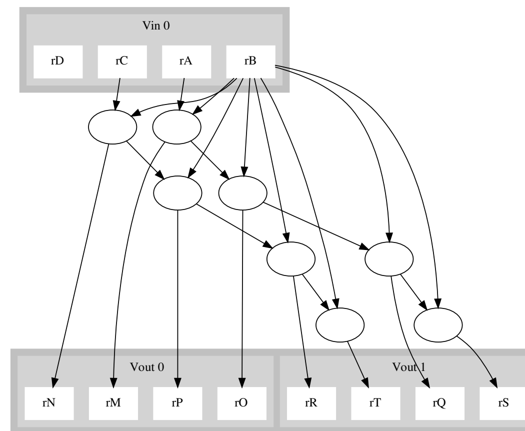


Compilation for Extension Instructions

- Current instruction set extension methodologies either:
  - Generate a standard tree-based instruction matcher, or,
  - Modify the original supplied code directly.
- Tree-based instruction matchers can not handle graph-shaped extension instructions.
- Modifying the original code prevents changes.
- Existing graph-based instruction matching techniques target small instructions.

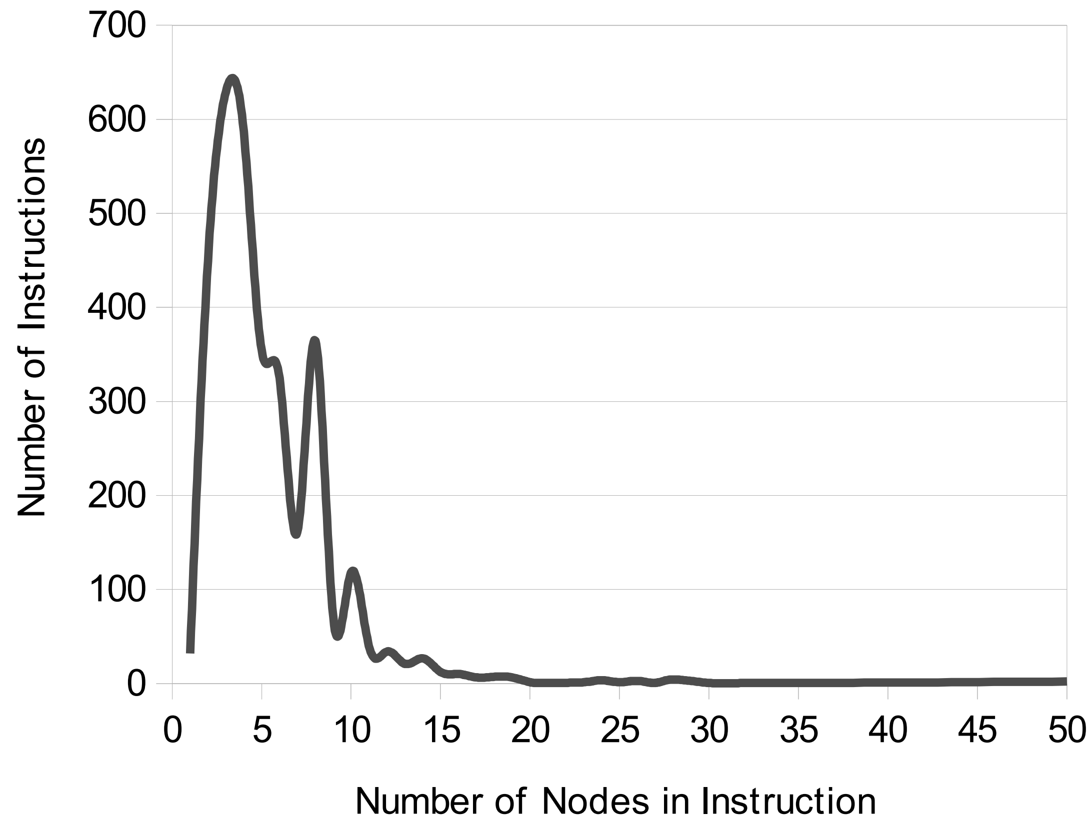
Motivation

```
  data = input;
  coef = coefficient;
  sum = 0.0;
  for (i = 0; i < 8; i++) {
#     term1 = *data++;
#     term2 = *coef++;
      sum += term1 * term2;
  }
  *output = sum;

.
```

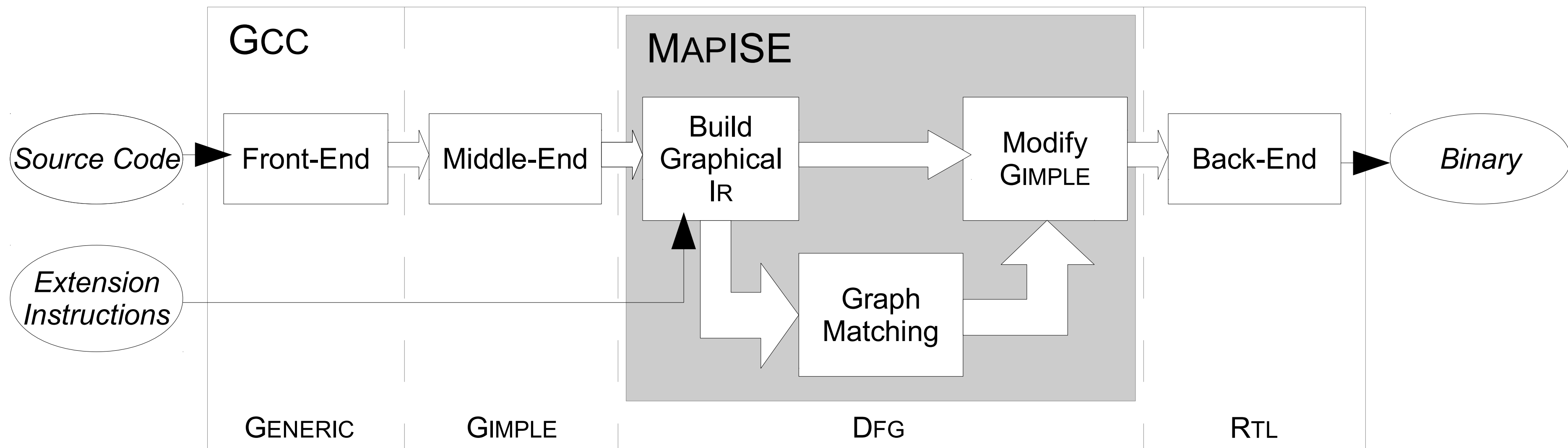



```
  data = input;
  coef = coefficient;
  sum = 0.0;
  term1 = *data++;
  term2 = *coef++;
  for (i = 1; i < 8; i++) {
      sum += term1 * term2;
#     term1 = *data++;
#     term2 = *coef++;
  }
  sum += term1 * term2;
  *output = sum;

.
```

Size of Extension Instructions

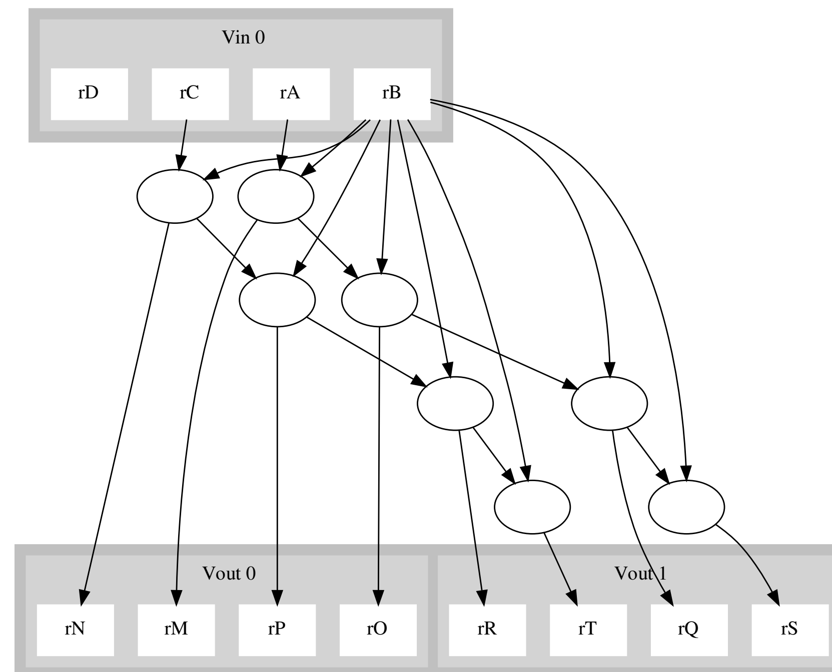


Gcc Passes



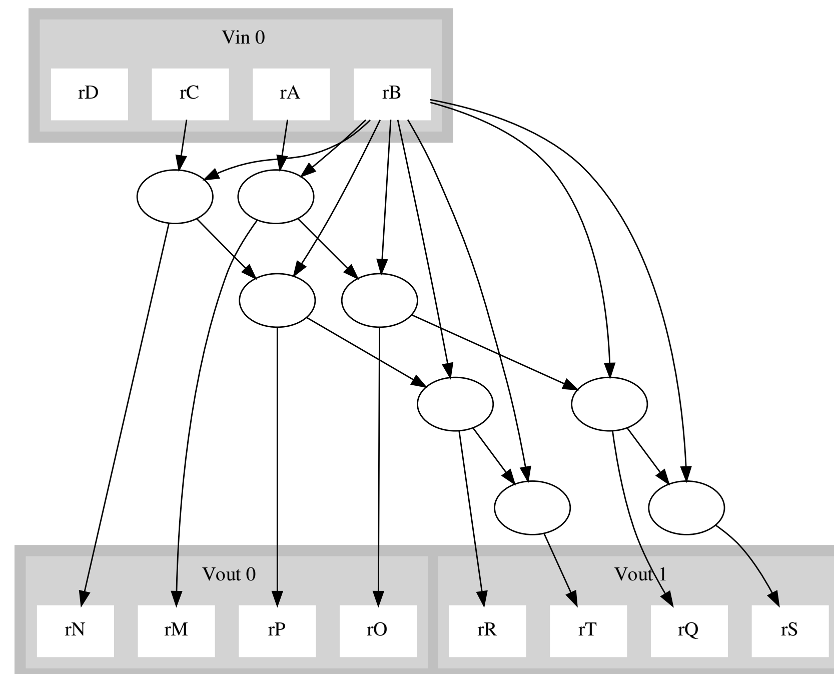
Example

```
  data = input;
  coef = coefficient;
  sum = 0.0;
  for (i = 0; i < 8; i++) {
#     term1 = *data++;
#     term2 = *coef++;
      sum += term1 * term2;
  }
  *output = sum;

.
```



Example

```
#     term1 = *data++;
#     term2 = *coef++;
      sum += term1 * term2;

  .
```



Example

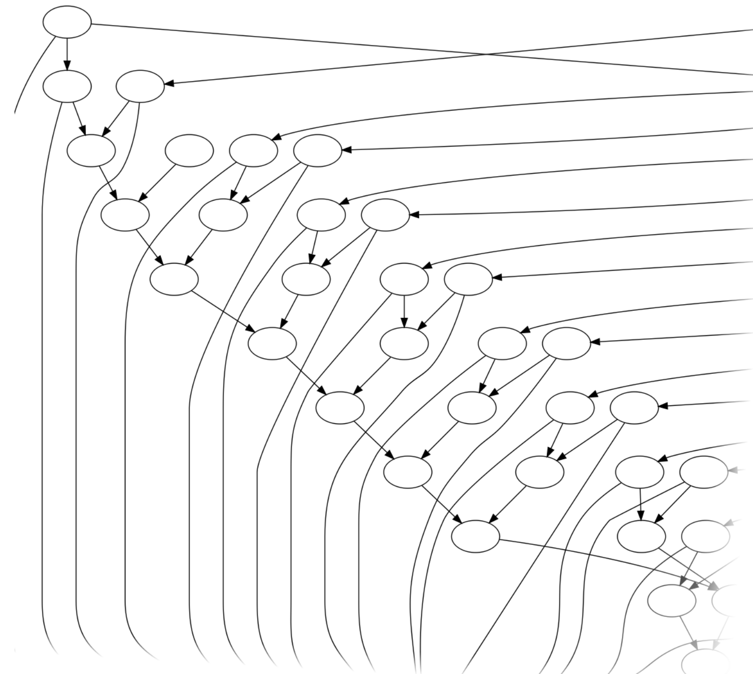
```
#     term1 = *data++;
#     term2 = *coef++;
      sum += term1 * term2;

 .
```

```
term1_72 = *input_10;
data_73 = input_10 + 4B;
term2_74 = *coefficient_12;
coef_75 = coefficient_12 + 4B;
D.1968_76 = term1_72 * term2_74;
sum_77 = D.1968_76 + 0.0;
term1_85 = *data_73;
data_86 = data_73 + 4B;
term2_87 = *coef_75;
coef_88 = coef_75 + 4B;
D.1968_89 = term1_85 * term2_87;
sum_90 = sum_77 + D.1968_89;
term1_98 = *data_86;
data_99 = data_86 + 4B;
... (70 more lines cut)
```

School of informatics

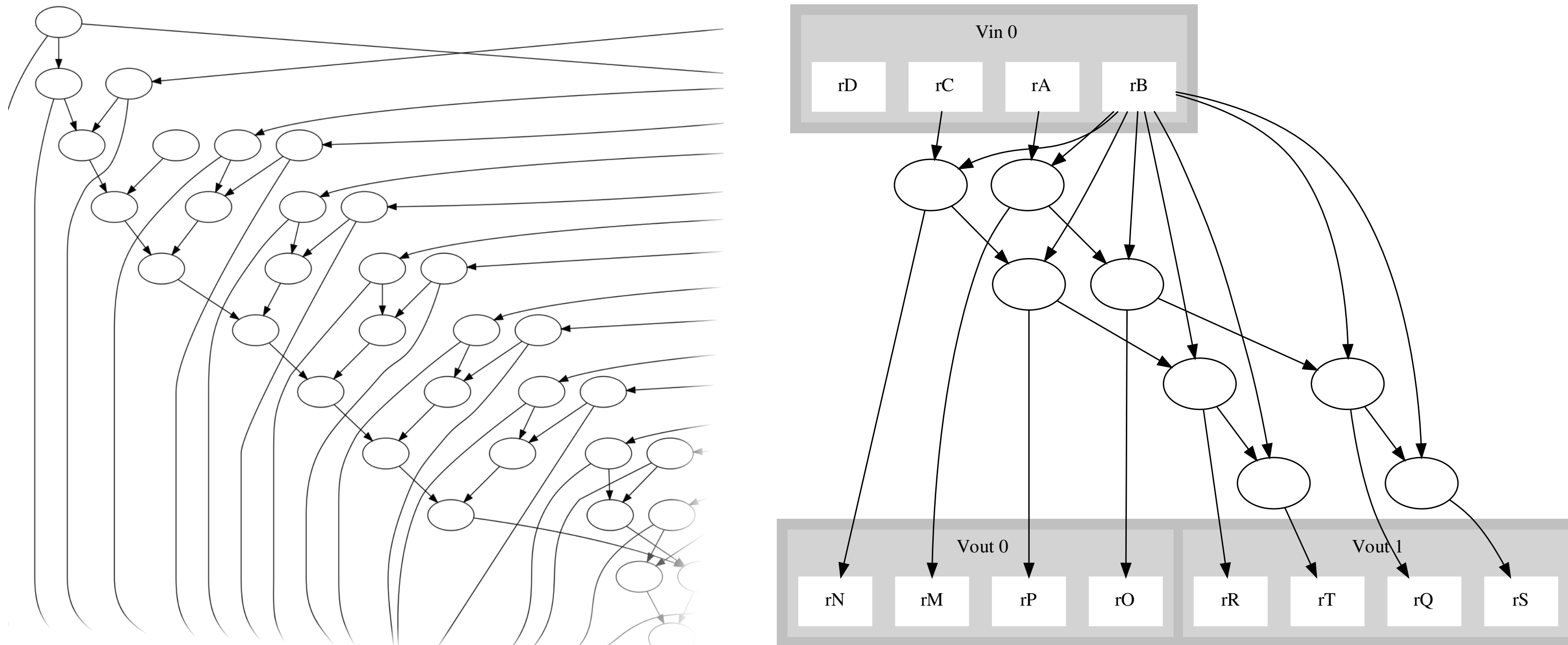
Example

```
term1_72 = *input_10;
data_73 = input_10 + 4B;
term2_74 = *coefficient_12;
coef_75 = coefficient_12 + 4B;
D.1968_76 = term1_72 * term2_74;
sum_77 = D.1968_76 + 0.0;
term1_85 = *data_73;
data_86 = data_73 + 4B;
term2_87 = *coef_75;
coef_88 = coef_75 + 4B;
D.1968_89 = term1_85 * term2_87;
sum_90 = sum_77 + D.1968_89;
term1_98 = *data_86;
data_99 = data_86 + 4B;
... (70 more lines cut)
```

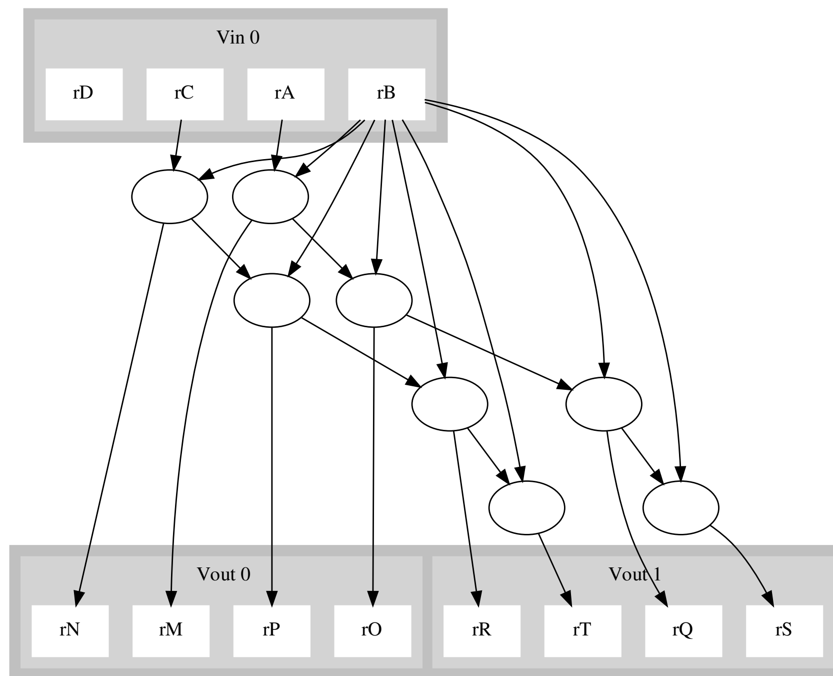


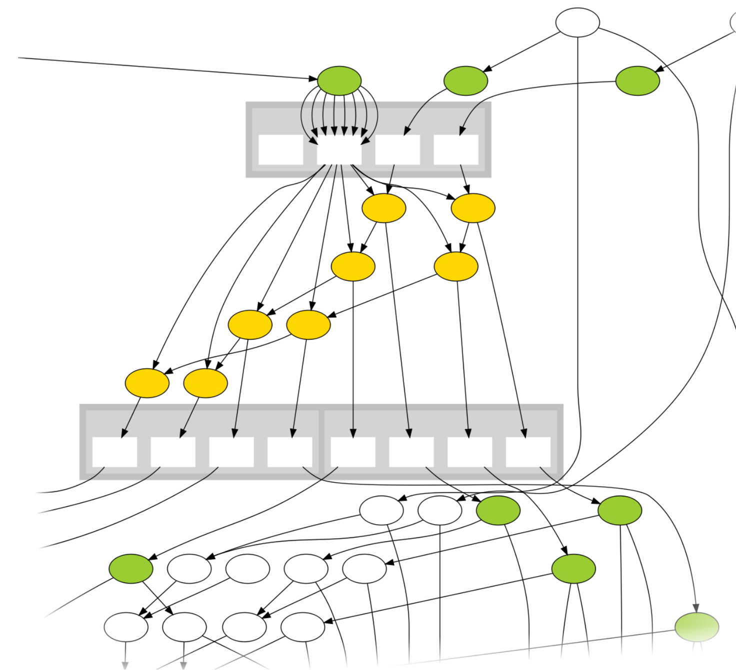
School of informatics

Example

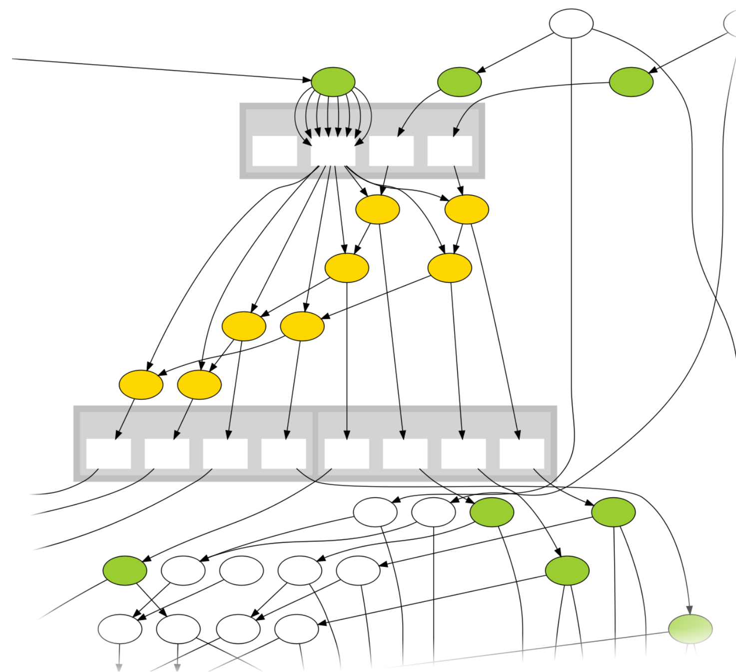


Example





Example

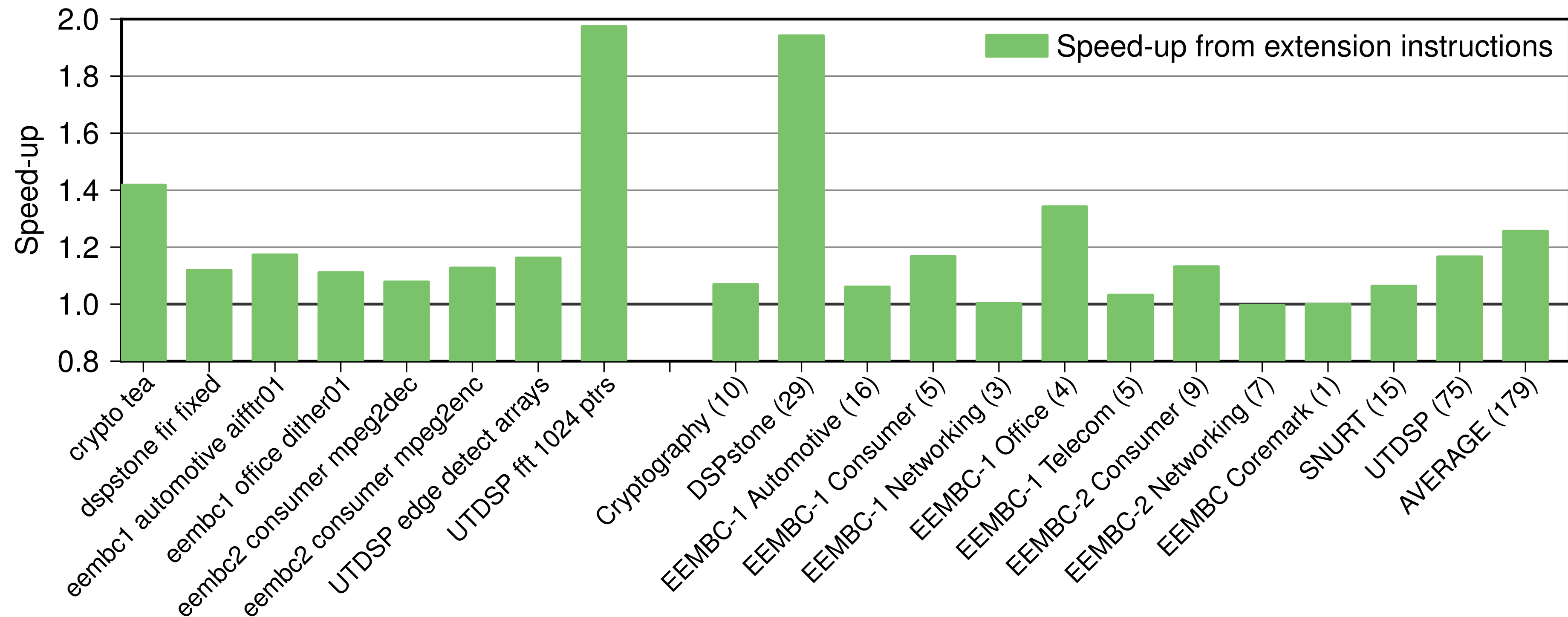


```
map_ise_const_tmp.44_11 = 4B + 0;
__asm__(  "vext001 vr02, vr03 = vr01, vr00 ;"
        : "=a36" data_73, "=a37" coef_75,
          "=a38" data_86, "=a39" coef_88,
          "=a40" data_99, "=a41" coef_101,
          "=a42" data_112, "=a43" coef_114
        : "a32" input_10,
          "a33" map_ise_const_tmp.44_11,
          "a34" coefficient_12);
coef_127 = coef_114 + 4B;
coef_140 = coef_127 + 4B;
coef_153 = coef_140 + 4B;
term2_19 = *coef_153;
data_125 = data_112 + 4B;
data_138 = data_125 + 4B;
```

Evaluation Methodology

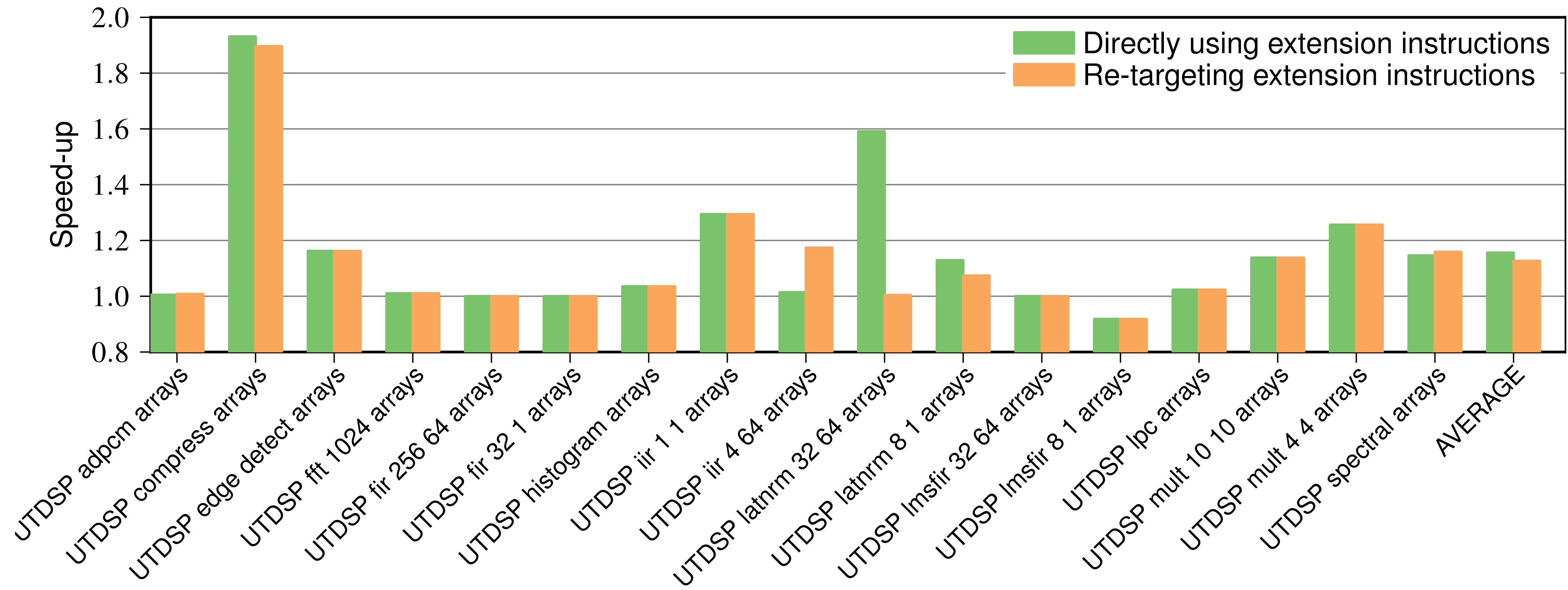
- ISEGEN algorithm used to generate extension instructions.
- Programs run on cycle-accurate simulator.
  - Simulator has been verified against hardware implementation.
  - Assume floating point hardware exists, for evaluation purposes
- 179 Benchmarks from:
  - EEMBC
  - UTDSP
  - DSPSTONE
  - SNURT
  - Reference Cryptography implementations

Results

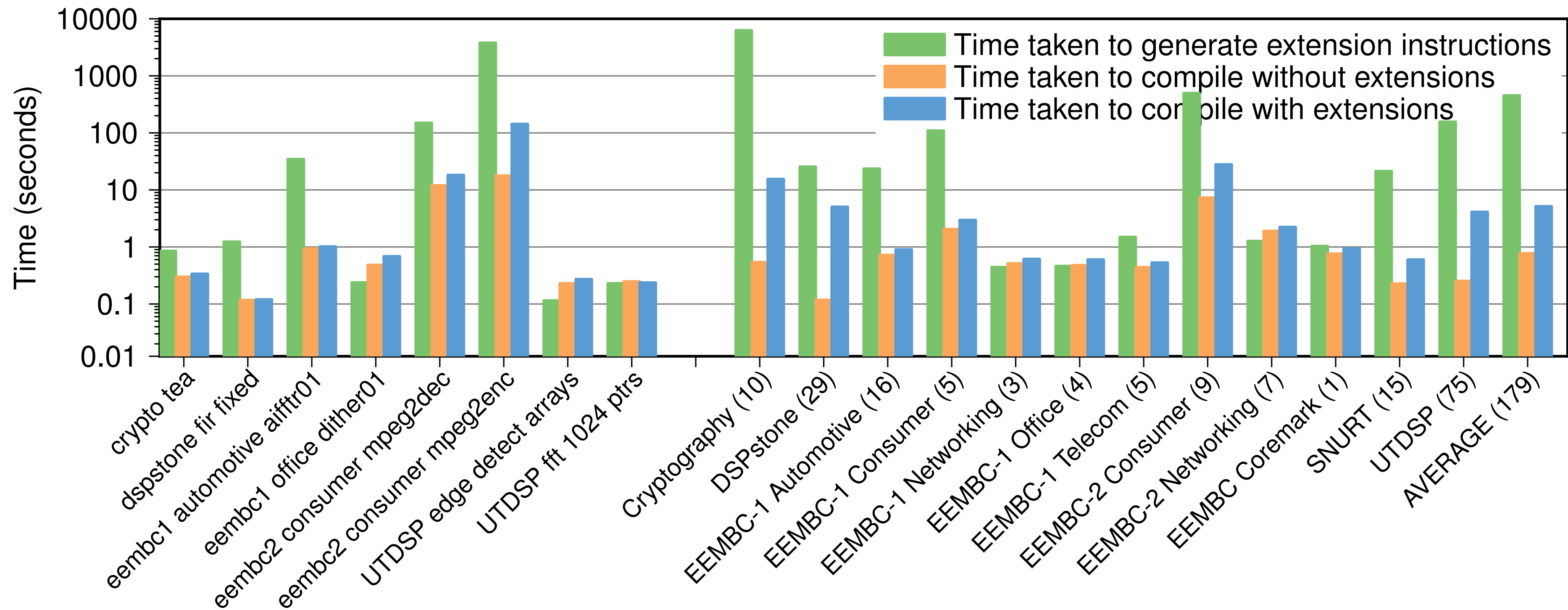


School of informatics

Results (Retargeting)



Timings



Conclusions

- A high-level pass focussing on extension instructions can:
  - Leave conventional instructions to the mature back-end.
  - Employ computationally expensive algorithms where they will help.
  - Maintain an acceptable run-time.
  - Achieve an average speed-up of 1.26 across 179 benchmarks.

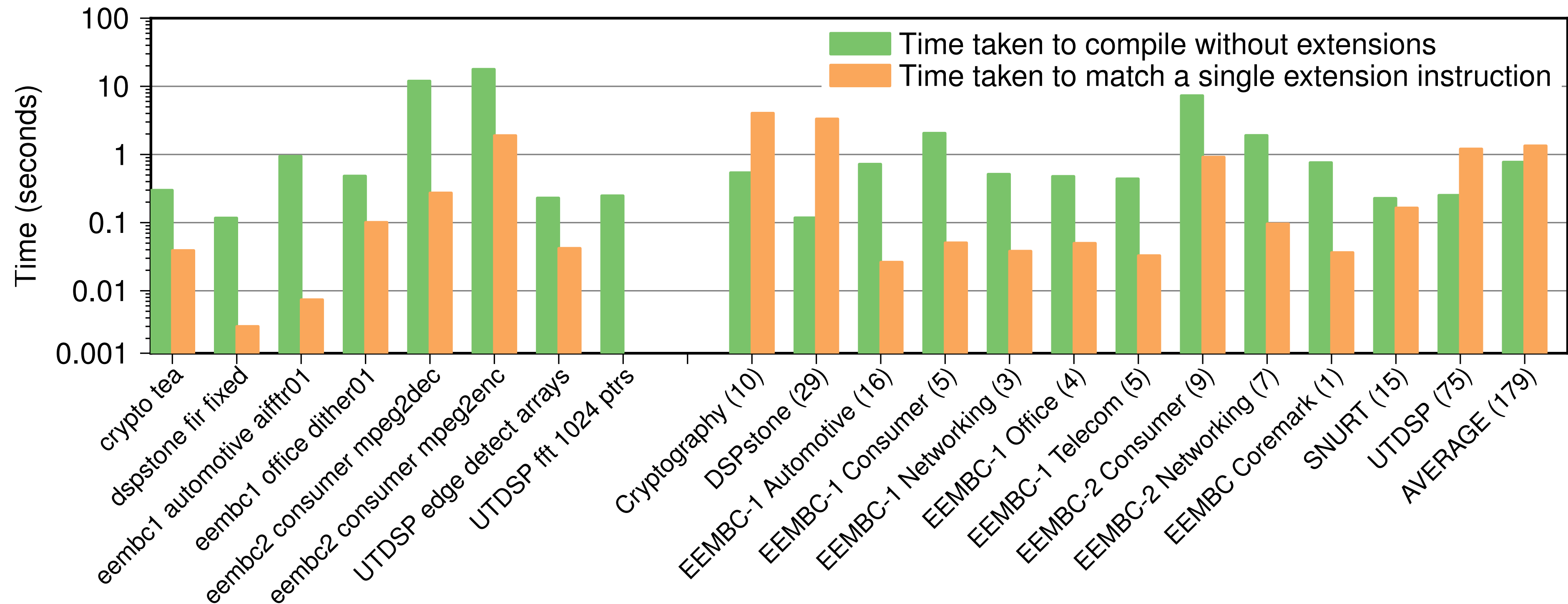
- Future work:
  - Integrate the use of this compiler into instruction set extension design space exploration.
  - Re-design the slowest parts of the matching algorithms.

Time Spent in Sub-passes

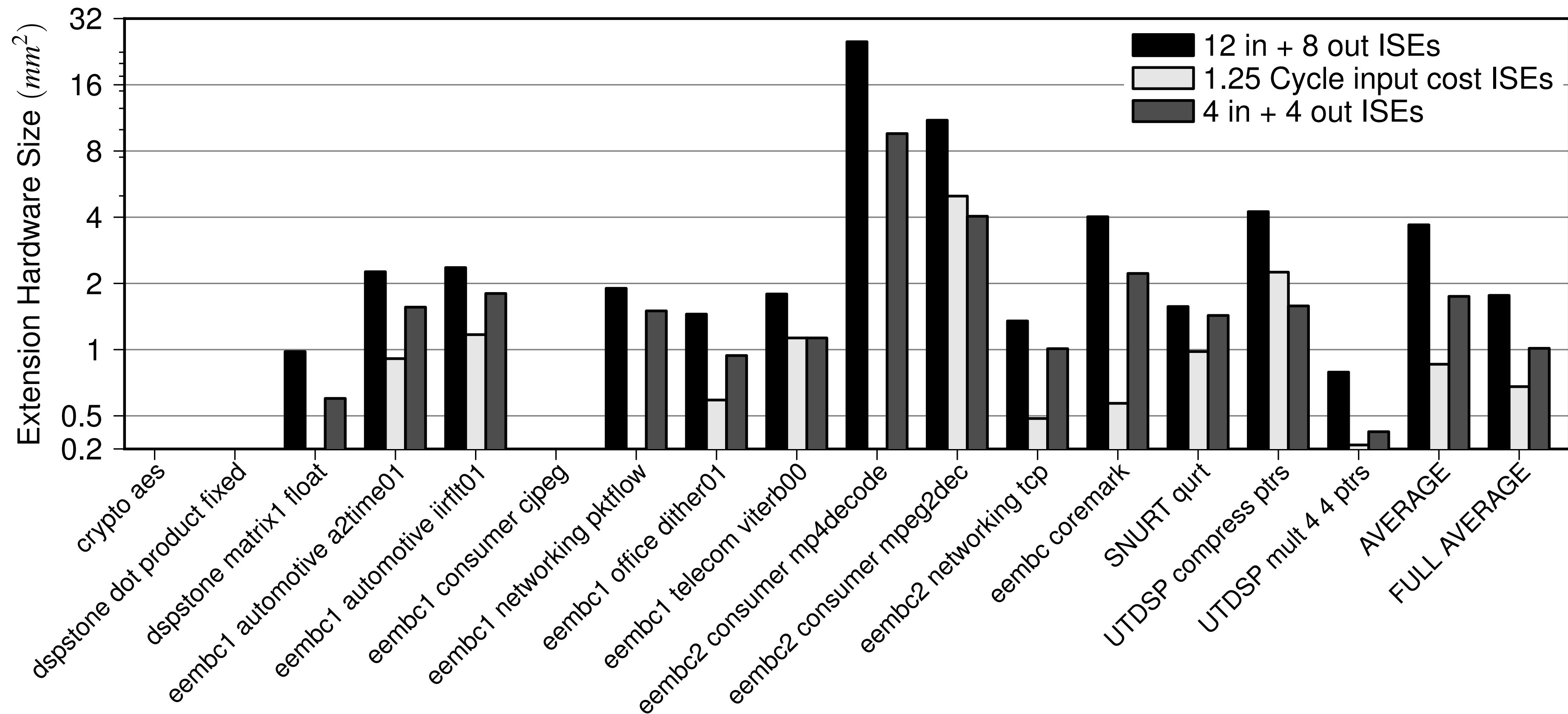
Task	Time (s)	Time %	Sub-Task	Time (s)	Time %
Build IRs	27.49	0.9%	Parse XML Instr's	9.33	0.3%
			Build DFG	5.09	0.2%
			Clean BB graphs	7.97	0.3%
			Build VF2 graphs	5.1	0.2%
Mapping	2,904.3	96.1%	VF2 Algorithm	1,482.6	49.1%
			Node Comparison	1,384.9	45.8%
			Viability Checking	33.07	1.1%
Register Allocation	6.92	0.2%			
GIMPLE Modification	83.43	2.8%	Scheduling	82.95	2.7%

Timings are from the summation of all 179 benchmarks.

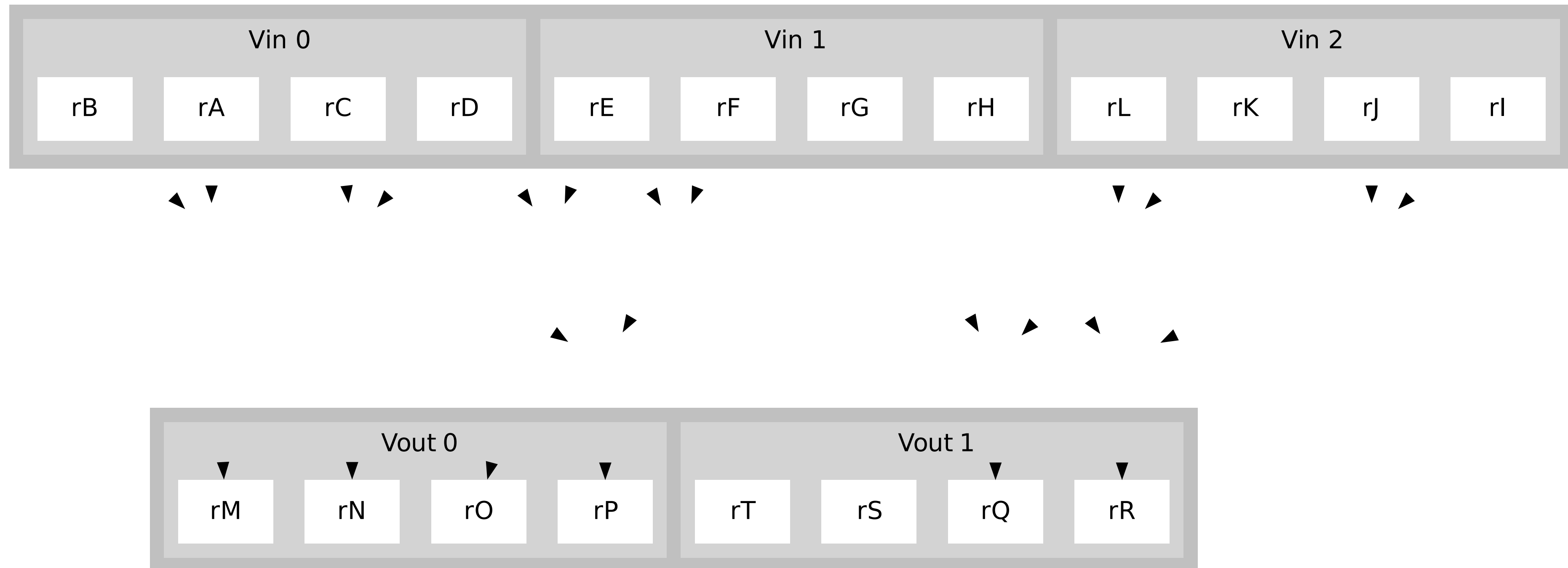
Timings (Normalised)



Hardware Sizes

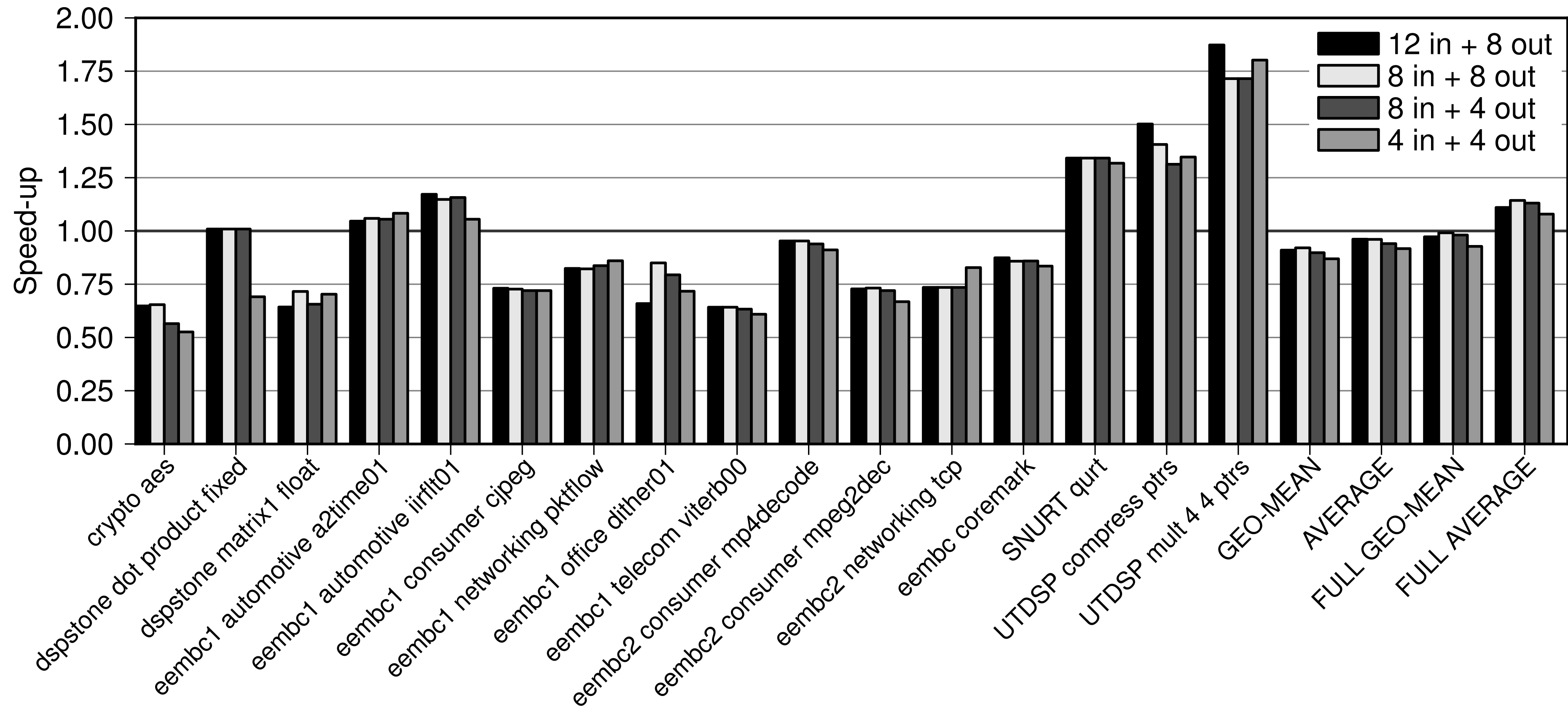


Match Validity

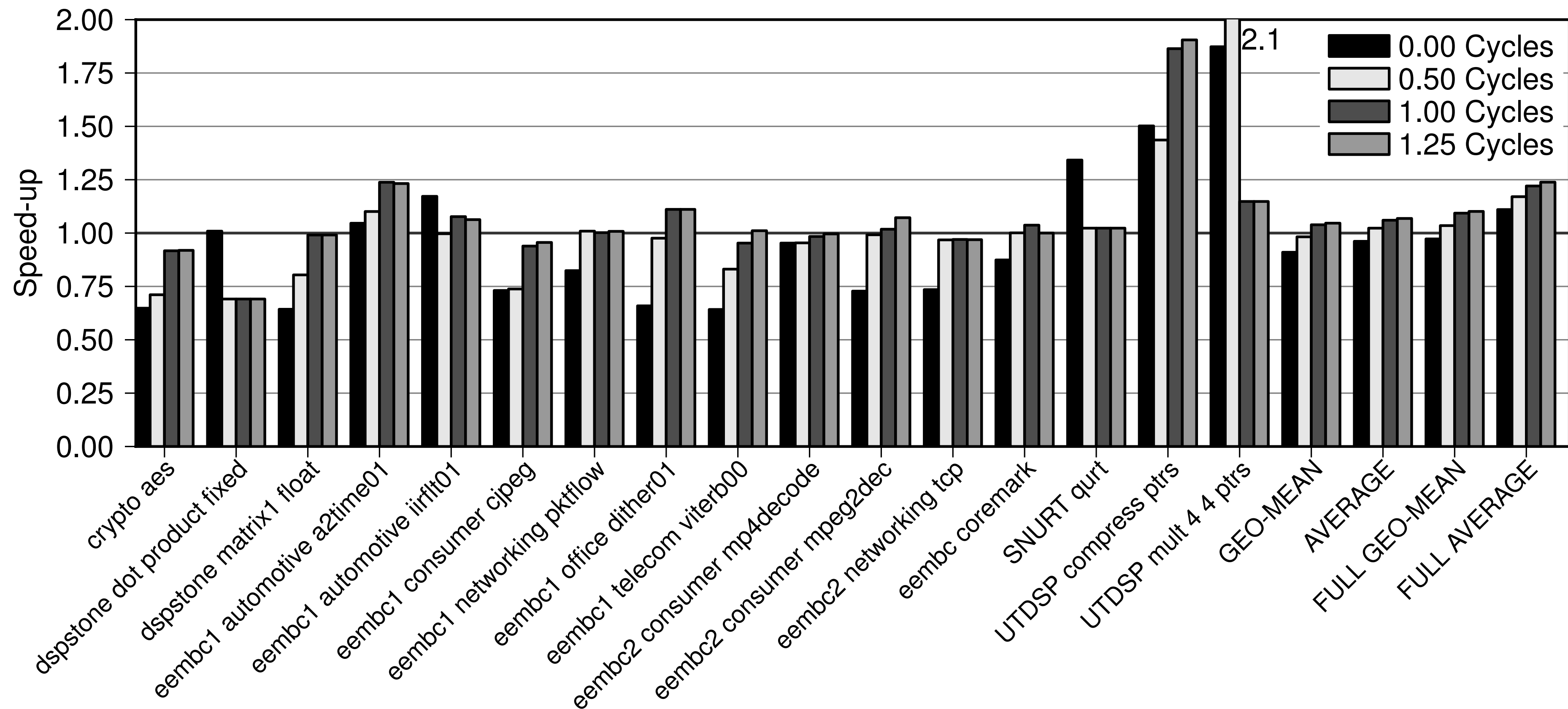


School of informatics

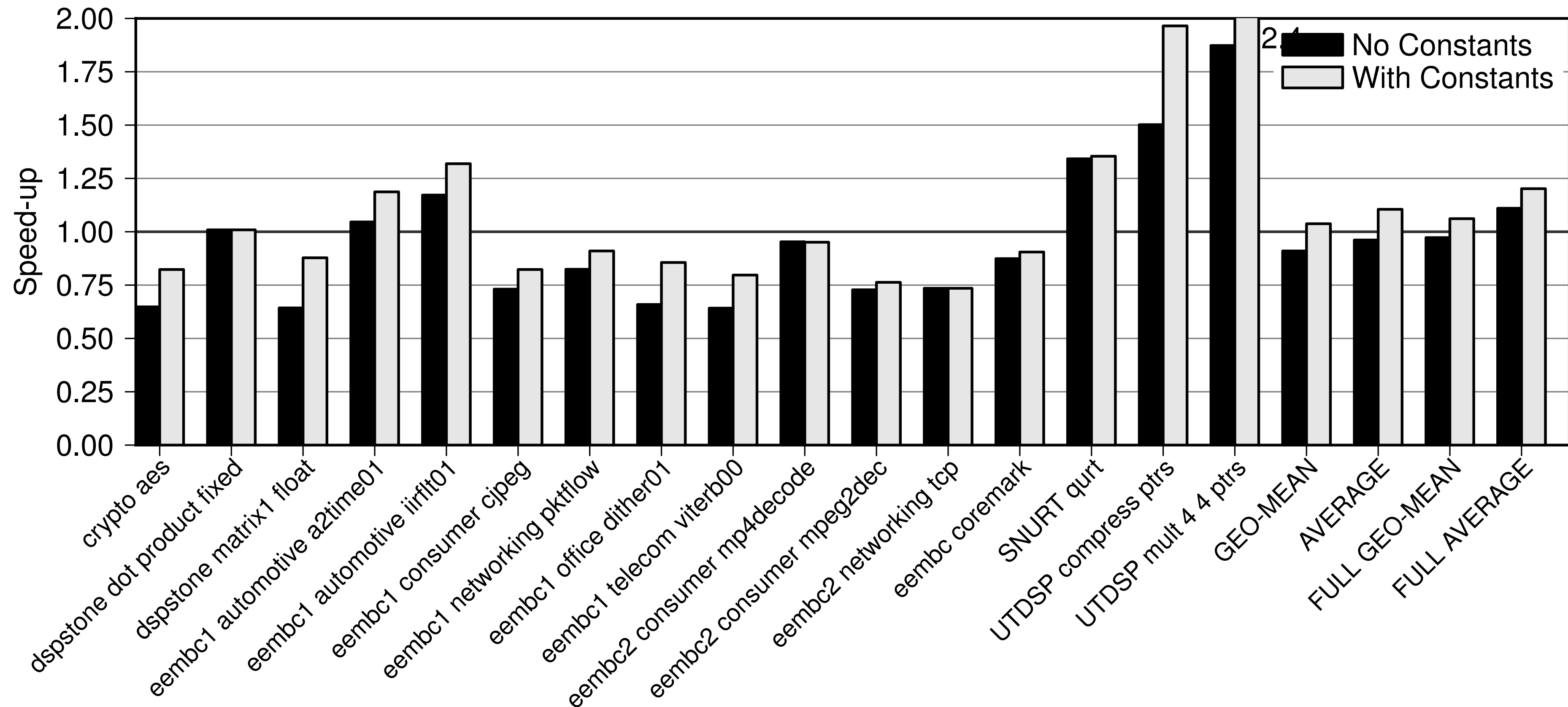
Varying Register Port Contraints



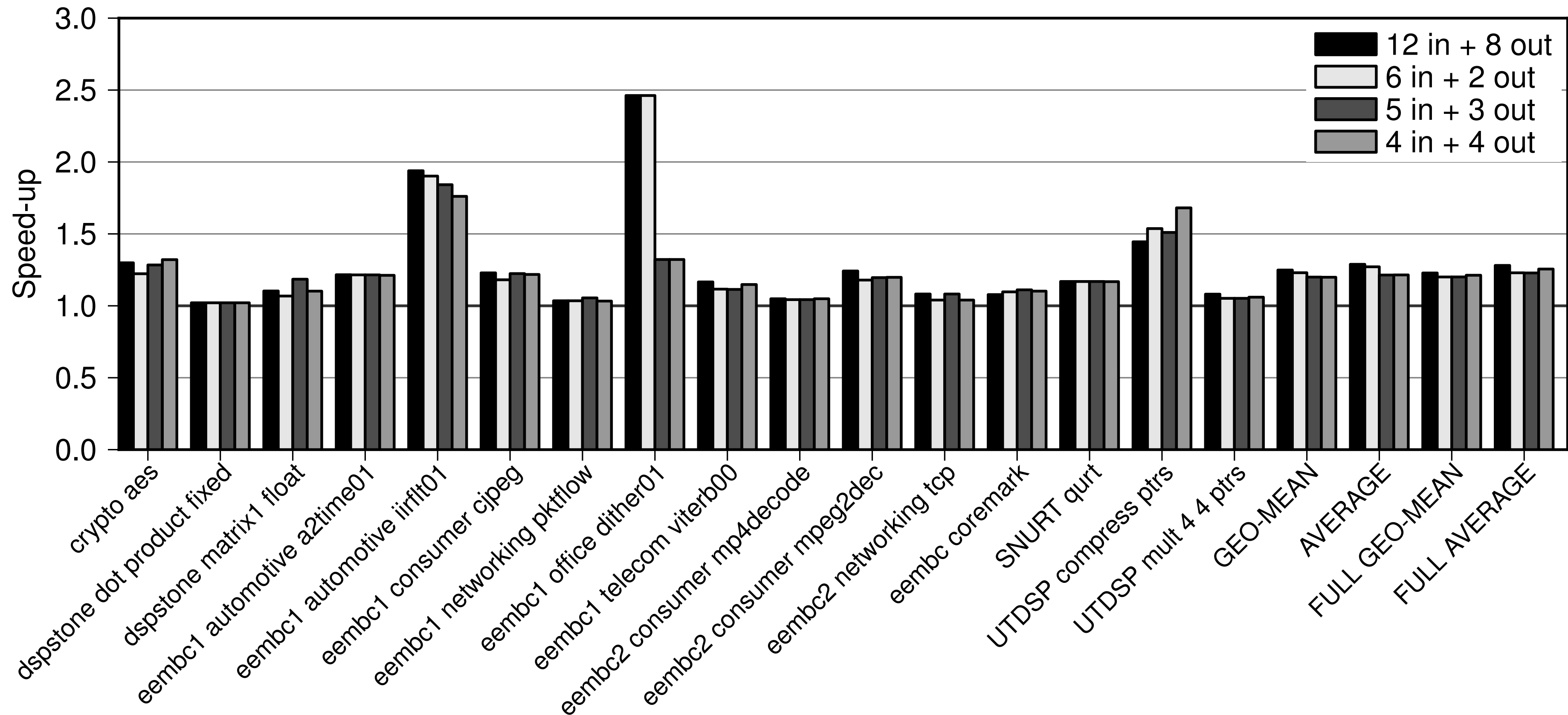
Tuning for Small Extension Instructions



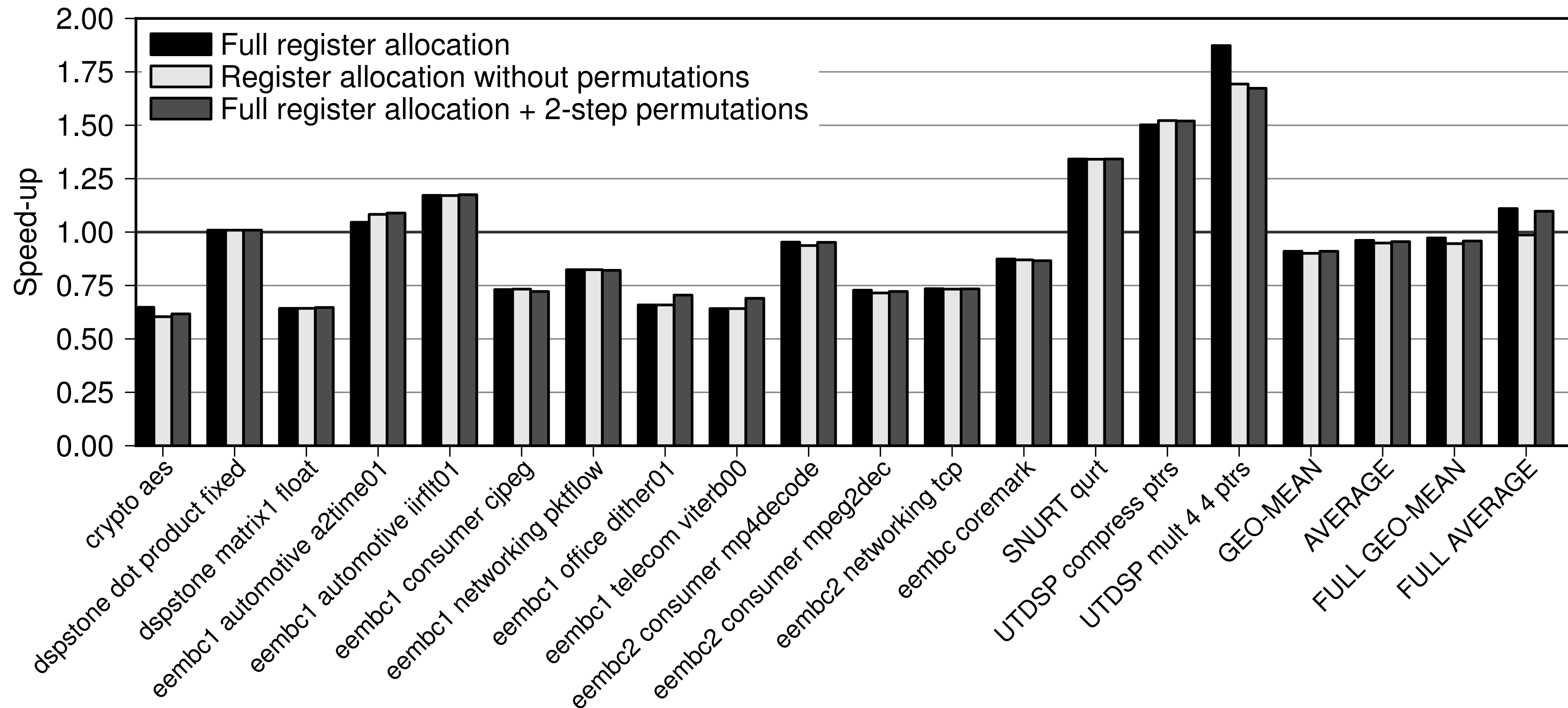
Embedded Constants



Scalar Registers

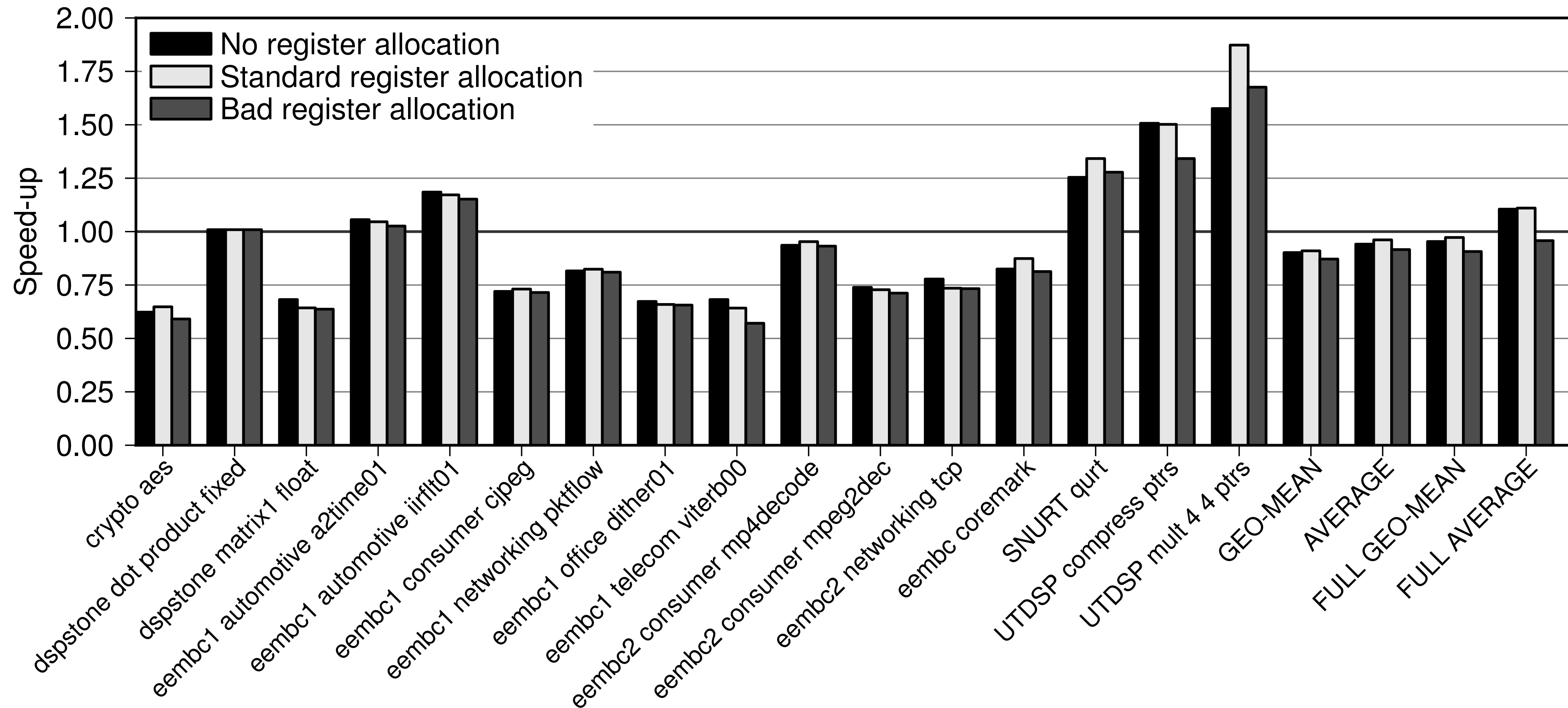


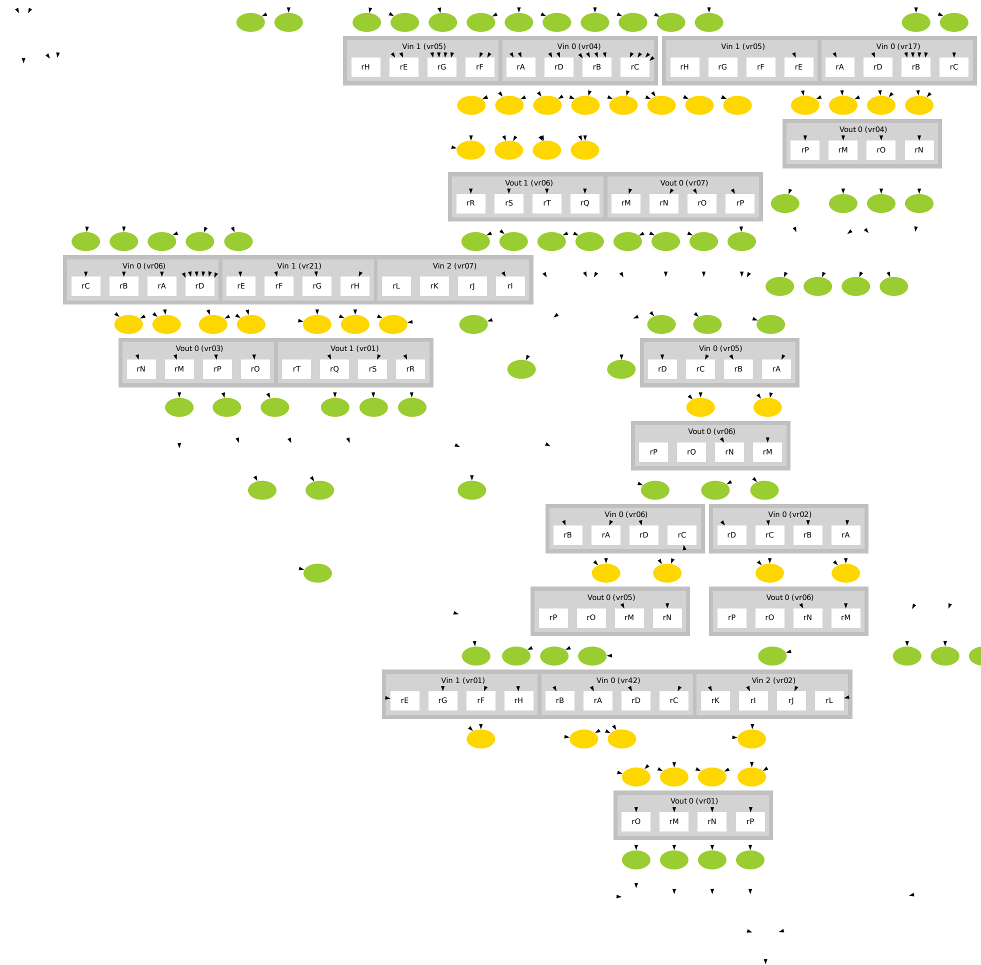
Permuation Units



School of informatics

Register Allocation





School of informatics