PROFILE-DIRECTED SEMI-AUTOMATIC PARALLELISATION

Björn Franke
Institute for Computing Systems Architecture
University of Edinburgh

Erlangen, 11 July 2011

Overview

- Motivation
- Part 1: Profile Directed Dependence Analysis
- Part 2: Machine Learning Based Mapping
- Part 3: Extraction of Pipeline Parallelism
- Conclusions

Motivation

- Multi-cores are ubiquitous
  - Try buying a single-core mobile phone, netbook, PC, or whatsoever!
- Legacy code base of sequential applications
  - Writing parallel applications is hard!
- No single parallel machine model
  - Different parallel programs for different parallel machines
  - Parallelisation is **not** a one-off activity: Need to parallelise each application for each new platform again

Motivation

- Multi-cores are ubiquitous
  - Try buying a single-core mobile phone, netbook

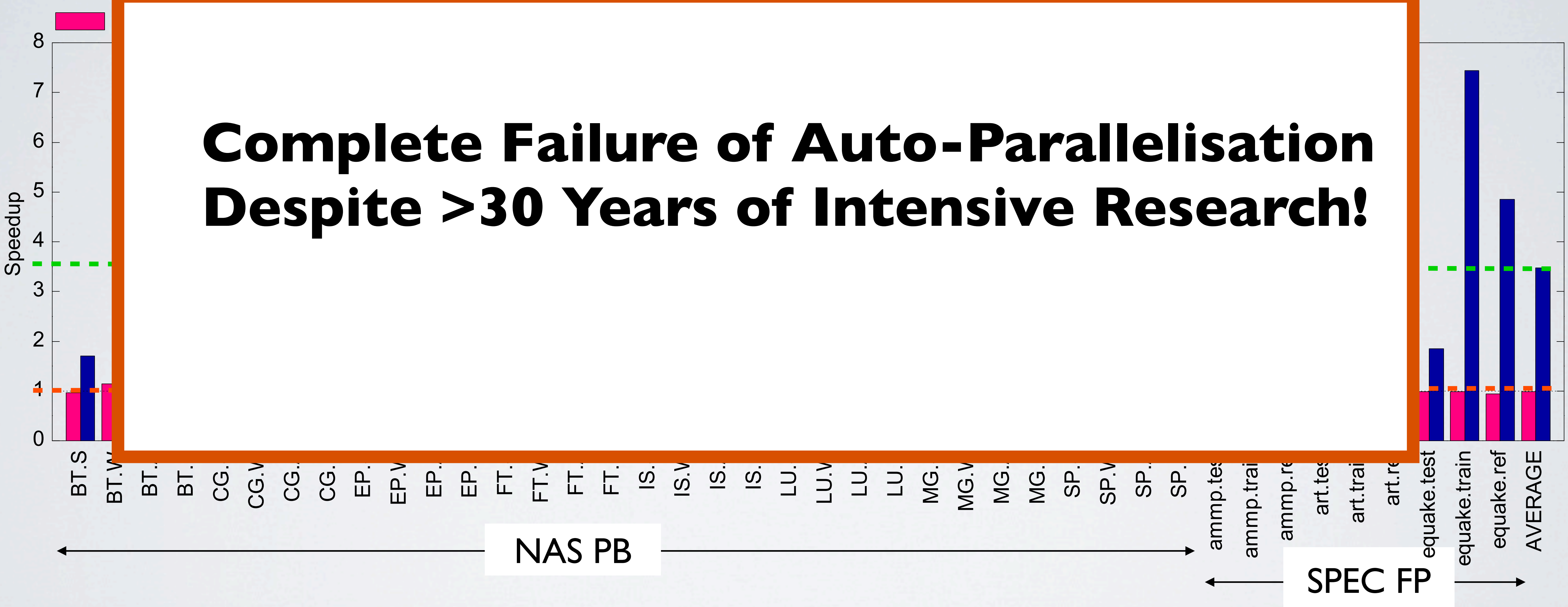
Tool Support for Parallelisation Increases Programmer Efficiency, Reduces Time-to-Market, Reduces Number of Bugs, Secures Software Investments etc.

BUT...

machines

  - Parallelisation is **not** a one-off activity: Need to parallelise each application for each new platform again

State-of-the-Art



Complete Failure of Auto-Parallelisation Despite >30 Years of Intensive Research!

NAS NPB 2.3 OMP-C and SPEC CFP2000
2 Quad-cores (8 cores in total) Intel Xeon X5450 @ 3.00GHz
Intel icc 10.1 -O2 -xT -axT -ipo

Observations

- Static Dependence Analysis Doesn't Work
  - ➞ Part 1: Profile Directed Dependence Analysis
- Mapping of Parallelism is Really Hard
  - ➞ Part 2: Machine Learning Based Mapping
- Parallelising FOR Loops is Not Enough
  - ➞ Part 3: Extraction of Pipeline Parallelism

PART 1:
PROFILE DIRECTED DEPENDENCE ANALYSIS

Motivating Example

***SPEC equake* (~75% of total exec. time)**

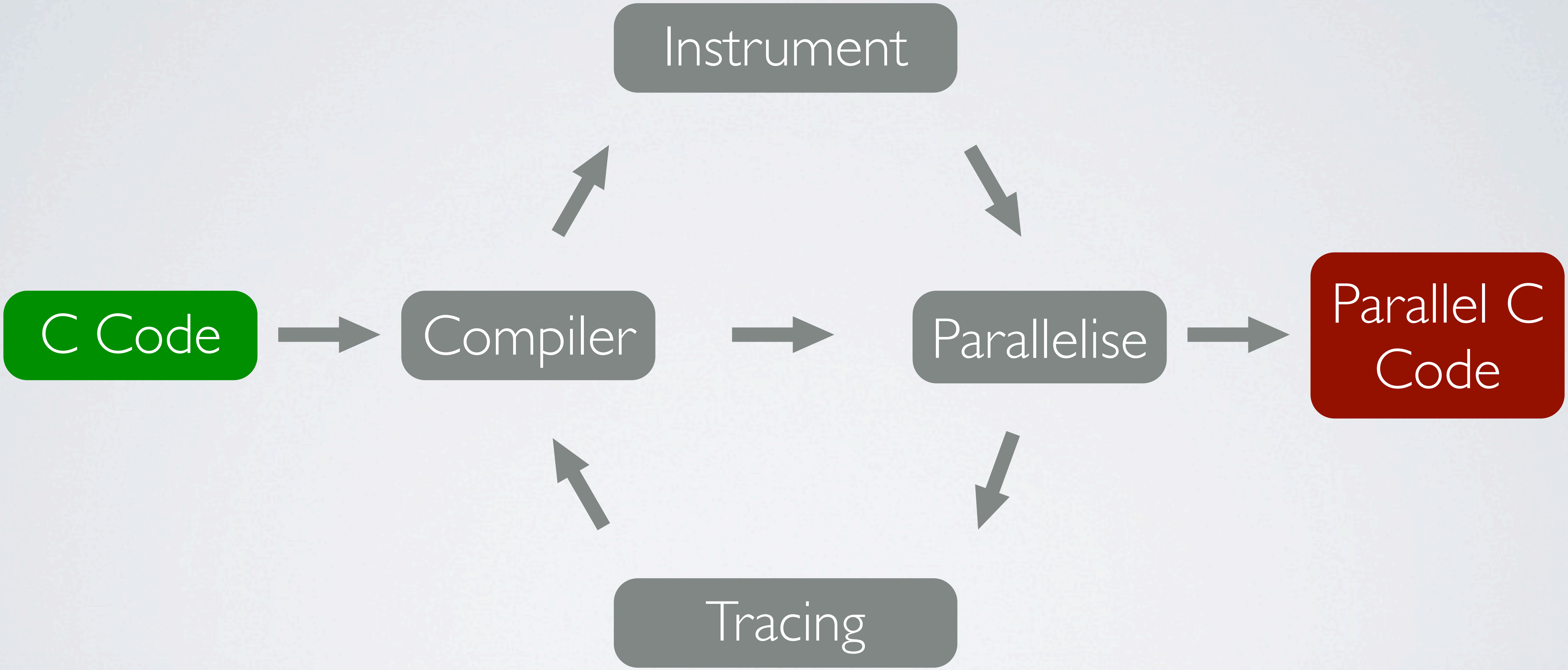
```
for (i = 0; i < nodes; i++) {
  Anext = Aindex[i];
  Alast = Aindex[i + 1];

  sum0 = A[Anext][0][0]*v[i][0] +
         A[Anext][0][1]*v[i][1] +
         A[Anext][0][2]*v[i][2];
  sum1 = ...

  Anext++;
  while (Anext < Alast) {
    col = Acol[Anext];

    sum0 += A[Anext][0][0]*v[col][0] +
            A[Anext][0][1]*v[col][1] +
            A[Anext][0][2]*v[col][2];
    sum1 += ...

    w[col][0] += A[Anext][0][0]*v[i][0] +
                 A[Anext][1][0]*v[i][1] +
                 A[Anext][2][0]*v[i][2];
    w[col][1] += ...
    Anext++;
  }
  w[i][0] += sum0;
  w[i][1] += ...
}
```

- Static analysis fails to detect any parallelism
- Problems :
  - indirect array accesses
  - compl. array reductions
  - variable iteration count
  - pointer aliasing
  - dynamic memory allocation
- But: Loop is parallel for all legal data inputs!

Profile Driven Parallelism Detection

- Use of profiling to capture data and control flow
  - Directly observe dependences ➙ accuracy
- But: Need to solve two important problems
  - No general correctness proof
    - May have missed dependences
    - Assisted user validation ➙ **semi**-automatic
  - Use low-level profiling information in compiler?
    - Instrumentation of intermediate representation
    - No actual ISA idiosyncracies

Approach


Instrument
C Code
Compiler
Parallelise
Parallel C Code
Tracing

Approach

- Instrument using high-level intermed. representation
  - Access to source-level information (memory accesses, loops, induction/reduction vars)
- Avoids ISA obfuscation
- Execute natively
  - Generates data and control flow traces
- Straightforward back-annotation
  - References to symbol table, IR nodes
- **Combination with conventional static analyses**

PART 2: MACHINE LEARNING BASED MAPPING

Motivating Example

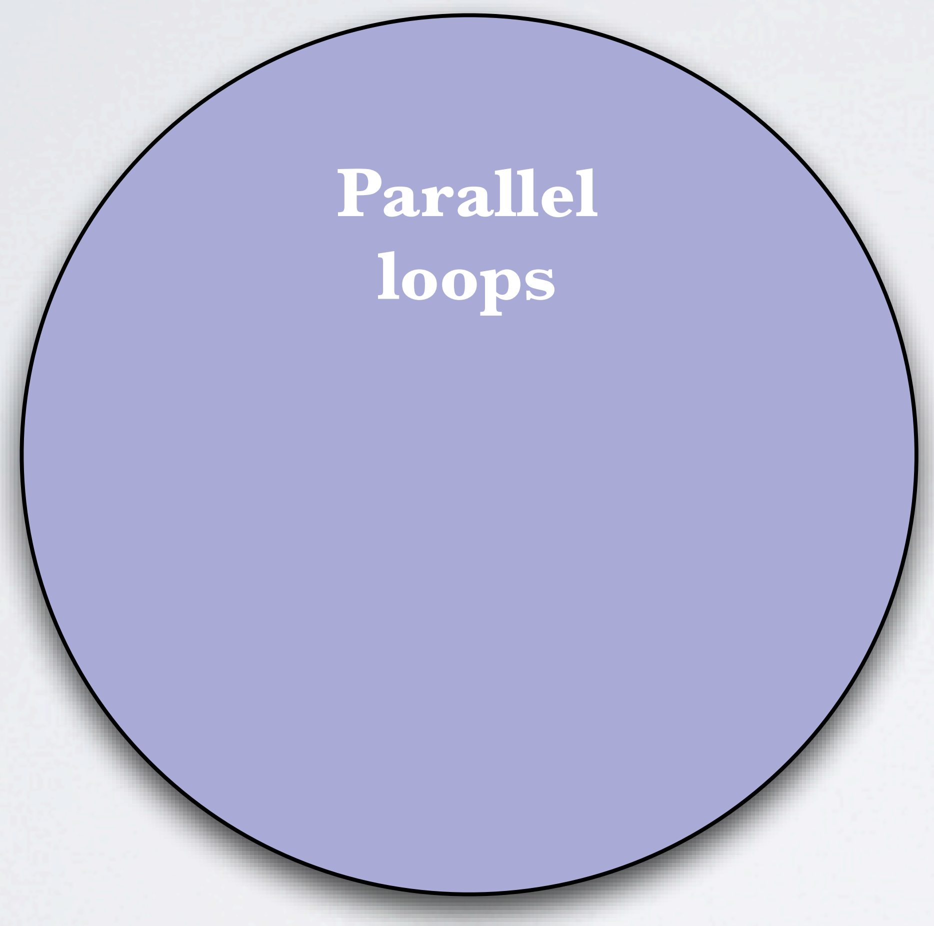
NAS CG

```
#pragma omp for reduction(+:sum) private(d)
for (j=1; j <= lastcol-firstcol-1; j++) {
    d = x[j] - r[j];
    sum = sum + d * d;
}
```

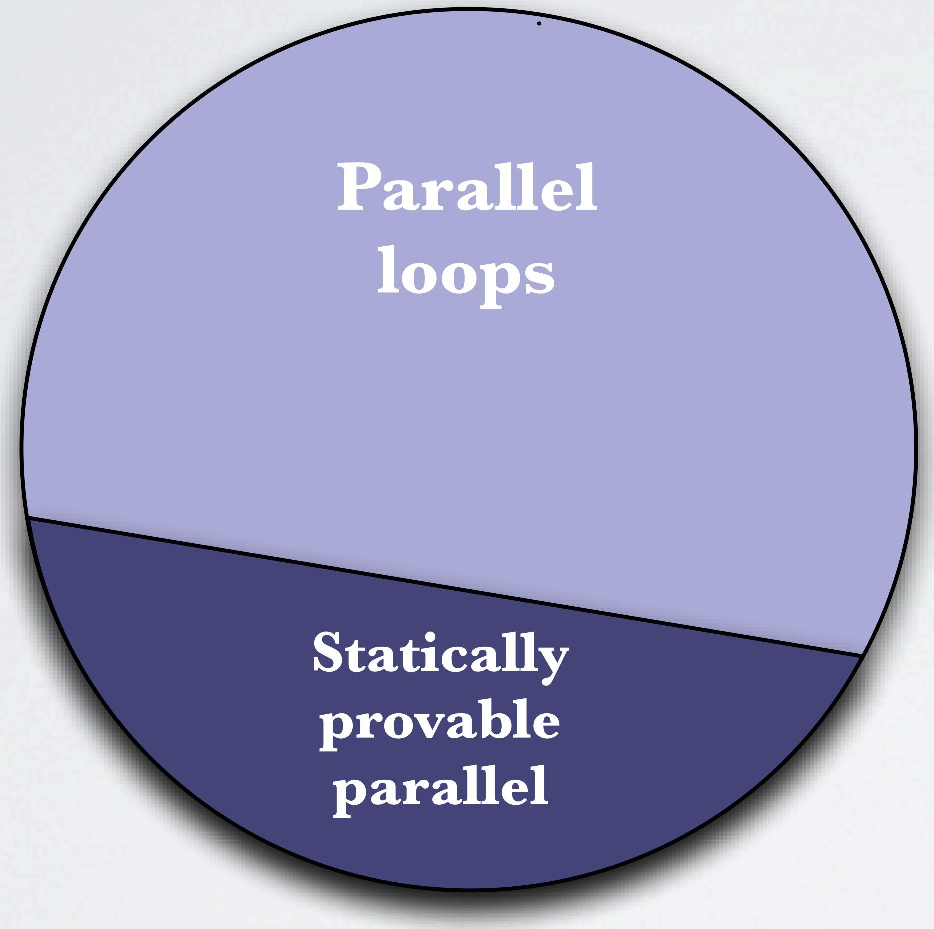
OpenMP

	Platform	
Scheduling	Cell BE	Intel Xeon
STATIC	slowdown	2.3x
DYNAMIC	slowdown	slowdown

Problem Statement

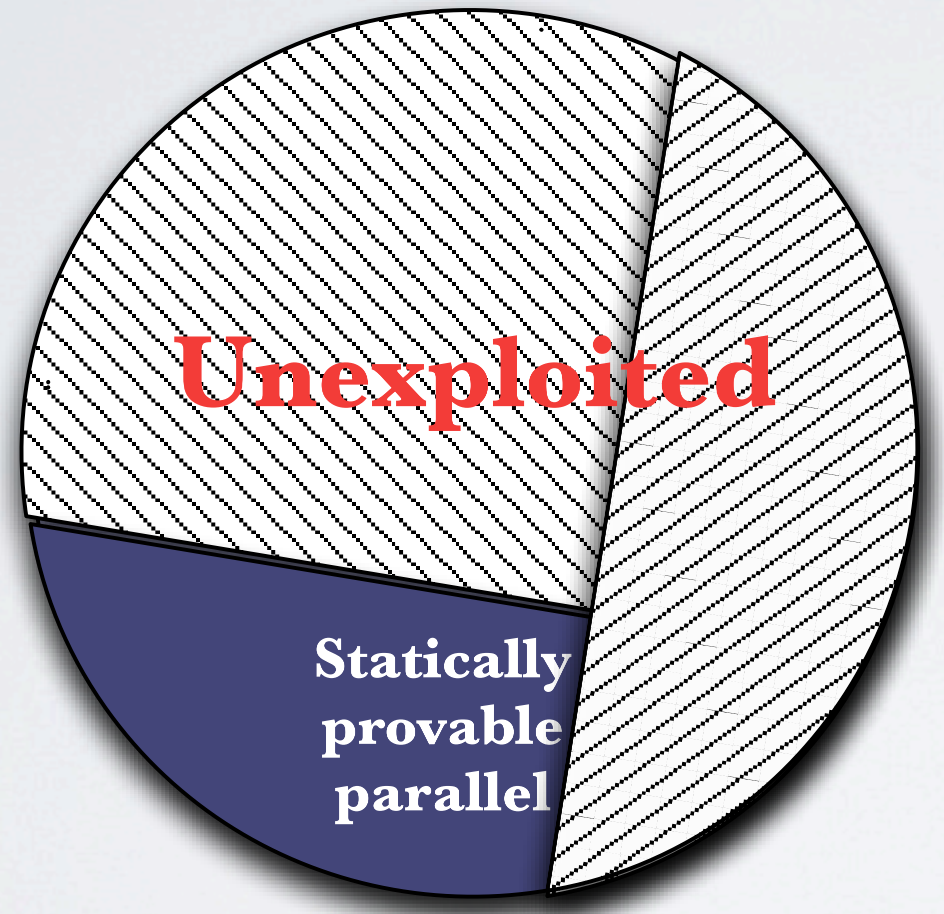


Problem Statement



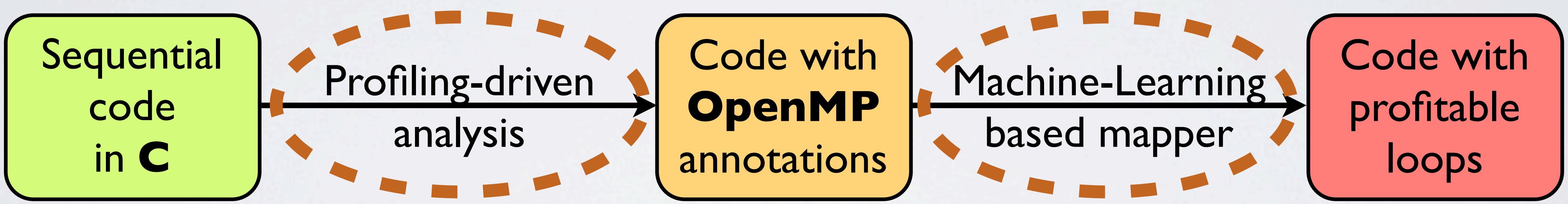
Problem Statement

Problem Statement

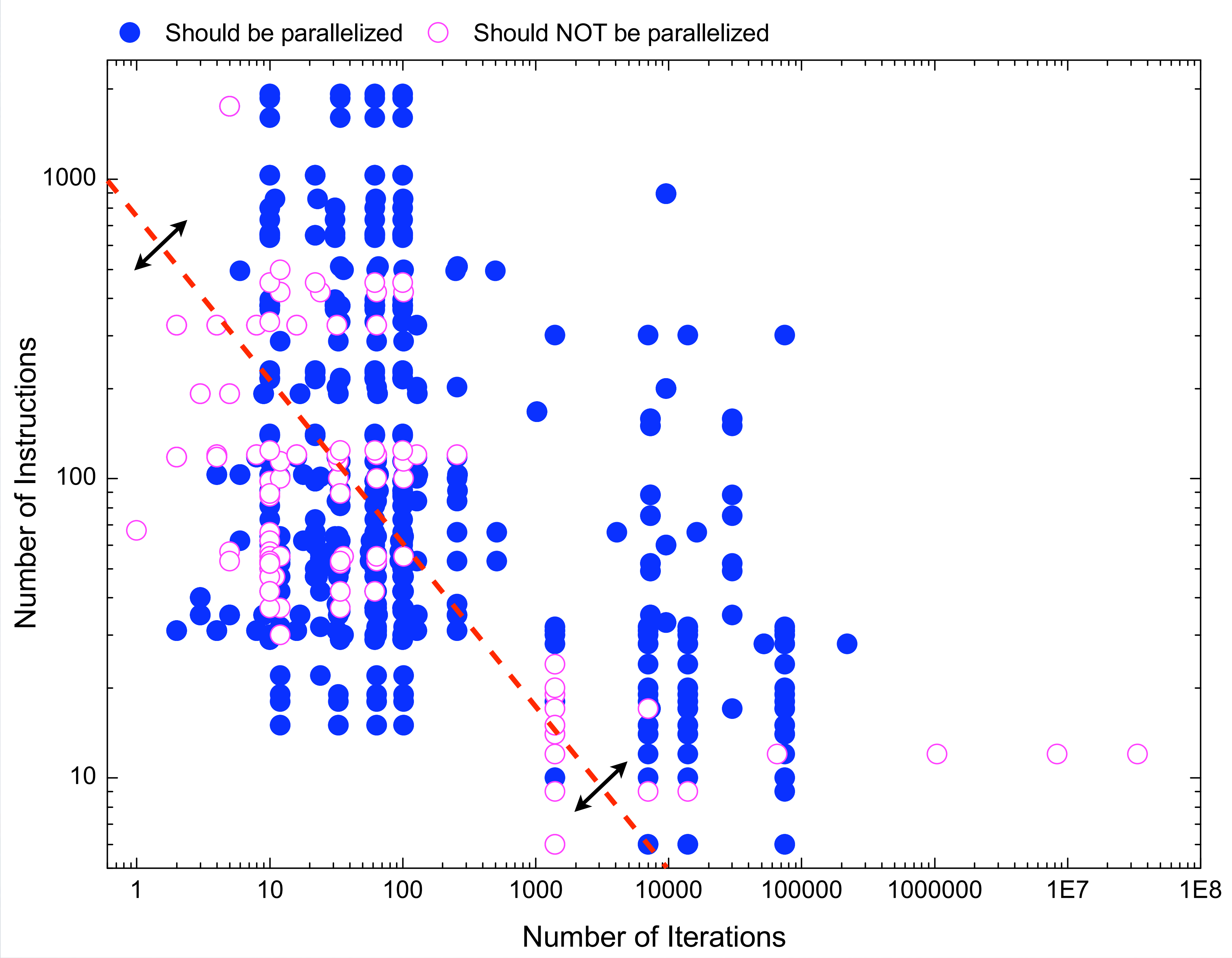


Profitable

Holistic Approach: Detection & Mapping



Conventional Mapping Heuristics



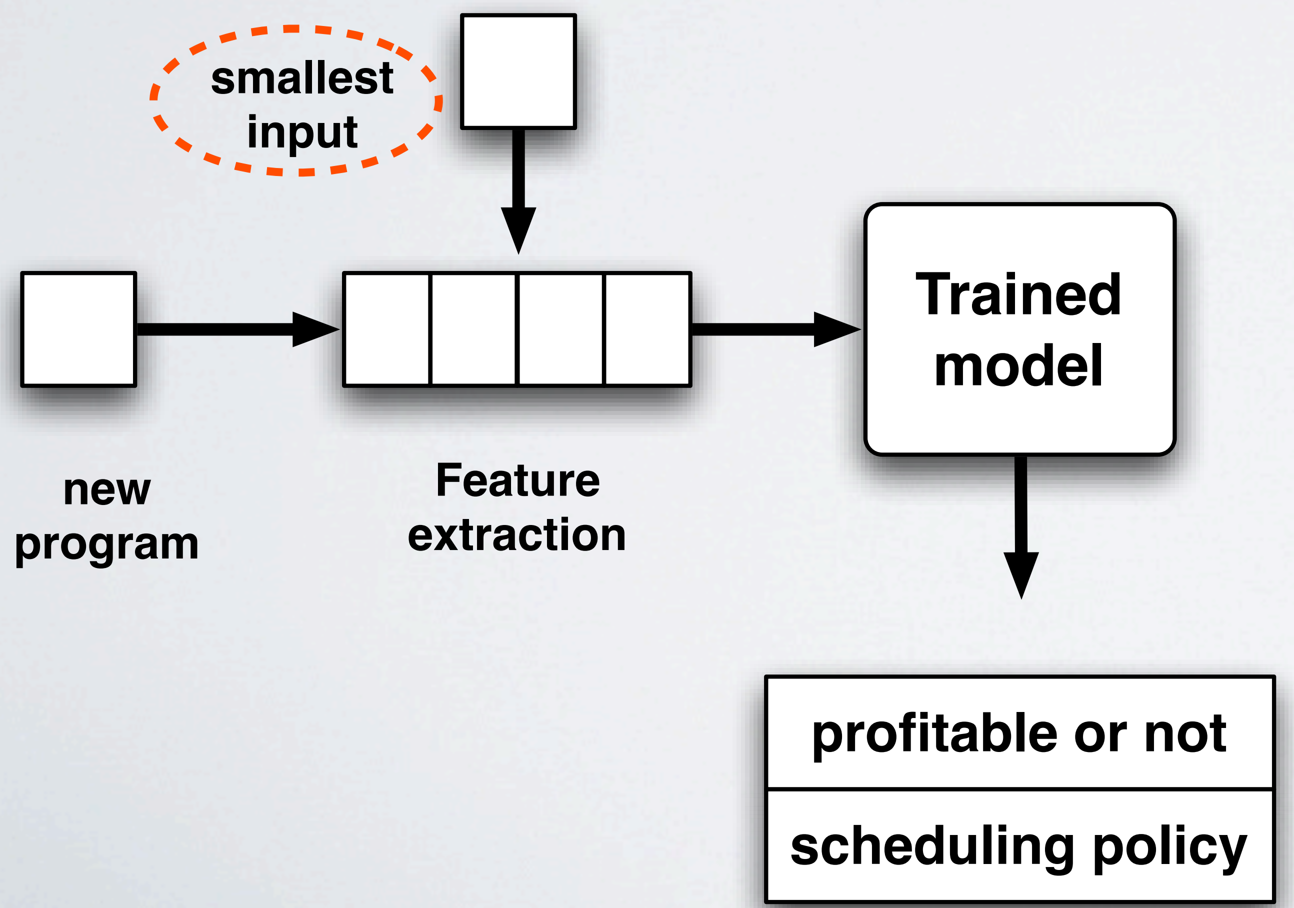
Static not suitable for separation.

Linear models not adequate either!

NAS PB 2.3+SPEC FP2000
Intel Xeon X5450

Machine Learning Based Mapping

- **Off-line** learning
- Predict using **smallest** input

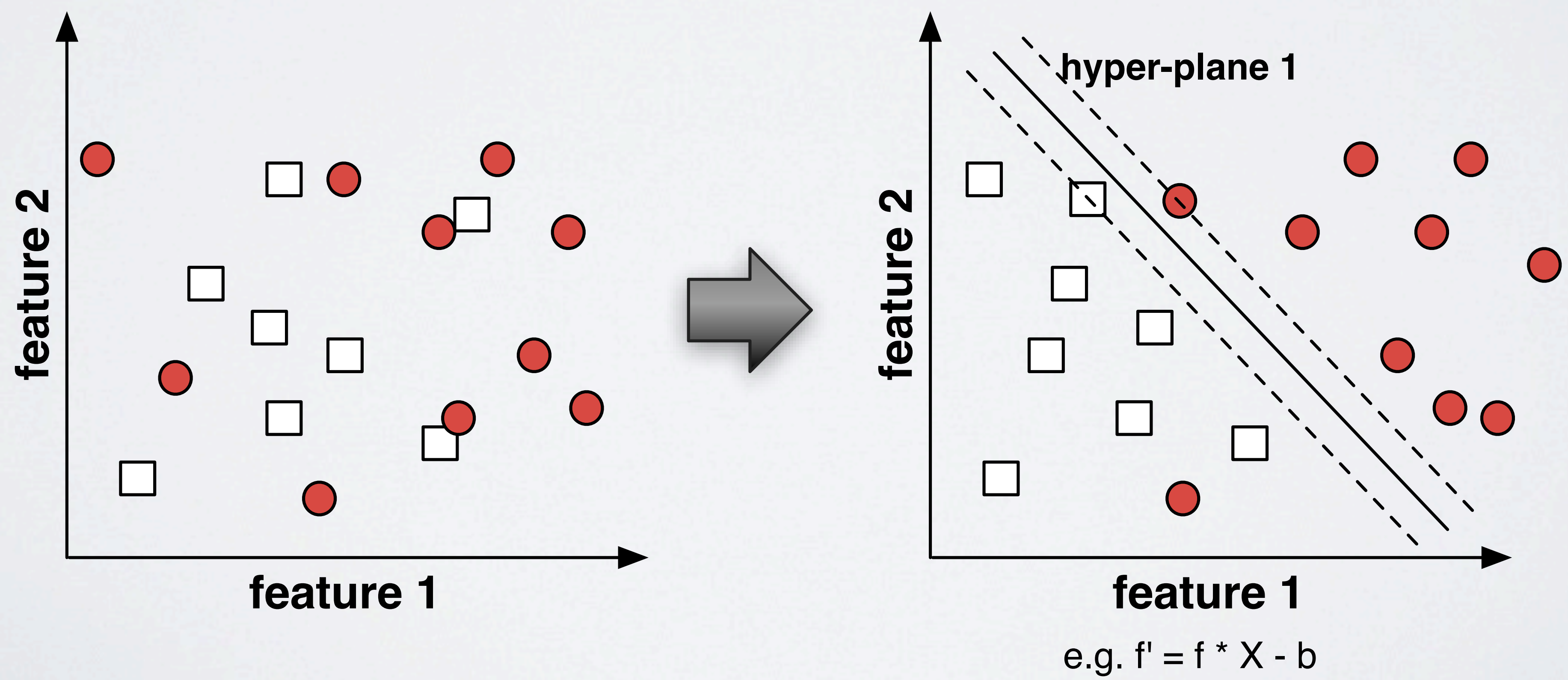


Features	
Static	IR instruction count IR Load/Store count IR Branch count Loop iteration count
Dynamic	Data access count Instruction count Branch count

Predictive Modelling

Support Vector Machine (SVM)

- Decide **(i)** profitability, **(ii)** loop scheduling
- Hyperplanes in **transformed higher-dimensional** space
- Non-linear & multi-class extensions



Experimental Evaluation

- 2 sets of applications
  - NAS Parallel Benchmarks 2.3
  - SPEC FP2000
- Sequential code in C
- Manually parallelized using OpenMP by expert programmers
- Use of multiple input datasets

Parallelism Coverage

- Profile-driven: almost **no lost opportunities**

Application	FP	FN
bt	0	0
cg	0	0
ep	0	0
ft	0	0
is	0	0
lu	0	0
mg	0	3
sp	0	0
equake	0	0
art	0	0
ammp	0	1

Parallelism Coverage

- *MG*: 3 loops never execute for all inputs
- *ammp*: critical loops require reshaping & locking

Application	FP	FN
bt	0	0
cg	0	0
ep	0	0
ft	0	0
is	0	0
lu	0	0
mg	0	3
sp	0	0
equake	0	0
art	0	0
ammp	0	1

Parallelism Coverage

- ICC finds many parallel loops

	icc		Profile-driven		Manual	
Applicatio	#loops(%cov)		#loops(%cov)		#loops(%cov)	
bt	72	(18.6%)	205	(99.9%)	54	(99.9%)
cg	16	(1.10%)	28	(93.1%)	22	(93.1%)
ep	6	(<1%)	8	(99.9%)	1	(99.9%)
ft	3	(<1%)	37	(88.2%)	6	(88.2%)
is	8	(29.4%)	9	(28.5%)	1	(27.3%)
lu	88	(65.9%)	54	(99.7%)	29	(81.5%)
mg	9	(4.70%)	48	(77.7%)	12	(77.7%)
sp	178	(88.0%)	287	(99.6%)	70	(61.8%)
equake	29	(23.8%)	69	(98.1%)	11	(98.0%)
art	16	(30.0%)	31	(85.6%)	5	(65.0%)
ammp	43	(<1%)	21	(1.40%)	7	(84.4%)

Parallelism Coverage

- BUT: low sequential time coverage
- ICC: majority of loops **too short** to be profitable

	icc		Profile-driven		Manual	
Applicatio	**#loops(%cov)**		**#loops(%cov)**		**#loops(%cov)**	
bt	72	(18.6%)	205	(99.9%)	54	(99.9%)
cg	16	(1.10%)	28	(93.1%)	22	(93.1%)
ep	6	(<1%)	8	(99.9%)	1	(99.9%)
ft	3	(<1%)	37	(88.2%)	6	(88.2%)
is	8	(29.4%)	9	(28.5%)	1	(27.3%)
lu	88	(65.9%)	54	(99.7%)	29	(81.5%)
mg	9	(4.70%)	48	(77.7%)	12	(77.7%)
sp	178	(88.0%)	287	(99.6%)	70	(61.8%)
equake	29	(23.8%)	69	(98.1%)	11	(98.0%)
art	16	(30.0%)	31	(85.6%)	5	(65.0%)
ammp	43	(<1%)	21	(1.40%)	7	(84.4%)

Parallelism Coverage

- Profile-driven: coverage close to manually parallelized

	icc		Profile-driven		Manual	
Applicatio	#loops(%cov)		#loops(%cov)		#loops(%cov)	
bt	72	(18.6%)	205	(99.9%)	54	(99.9%)
cg	16	(1.10%)	28	(93.1%)	22	(93.1%)
ep	6	(<1%)	8	(99.9%)	1	(99.9%)
ft	3	(<1%)	37	(88.2%)	6	(88.2%)
is	8	(29.4%)	9	(28.5%)	1	(27.3%)
lu	88	(65.9%)	54	(99.7%)	29	(81.5%)
mg	9	(4.70%)	48	(77.7%)	12	(77.7%)
sp	178	(88.0%)	287	(99.6%)	70	(61.8%)
equake	29	(23.8%)	69	(98.1%)	11	(98.0%)
art	16	(30.0%)	31	(85.6%)	5	(65.0%)
ammp	43	(<1%)	21	(1.40%)	7	(84.4%)

Parallelism Coverage

- ammp: we fail to parallelize the critical loop

	icc		Profile-driven		Manual	
Applicatio	#loops(%cov)		#loops(%cov)		#loops(%cov)	
bt	72	(18.6%)	205	(99.9%)	54	(99.9%)
cg	16	(1.10%)	28	(93.1%)	22	(93.1%)
ep	6	(<1%)	8	(99.9%)	1	(99.9%)
ft	3	(<1%)	37	(88.2%)	6	(88.2%)
is	8	(29.4%)	9	(28.5%)	1	(27.3%)
lu	88	(65.9%)	54	(99.7%)	29	(81.5%)
mg	9	(4.70%)	48	(77.7%)	12	(77.7%)
sp	178	(88.0%)	287	(99.6%)	70	(61.8%)
equake	29	(23.8%)	69	(98.1%)	11	(98.0%)
art	16	(30.0%)	31	(85.6%)	5	(65.0%)
ammp	43	(<1%)	21	(1.40%)	7	(84.4%)

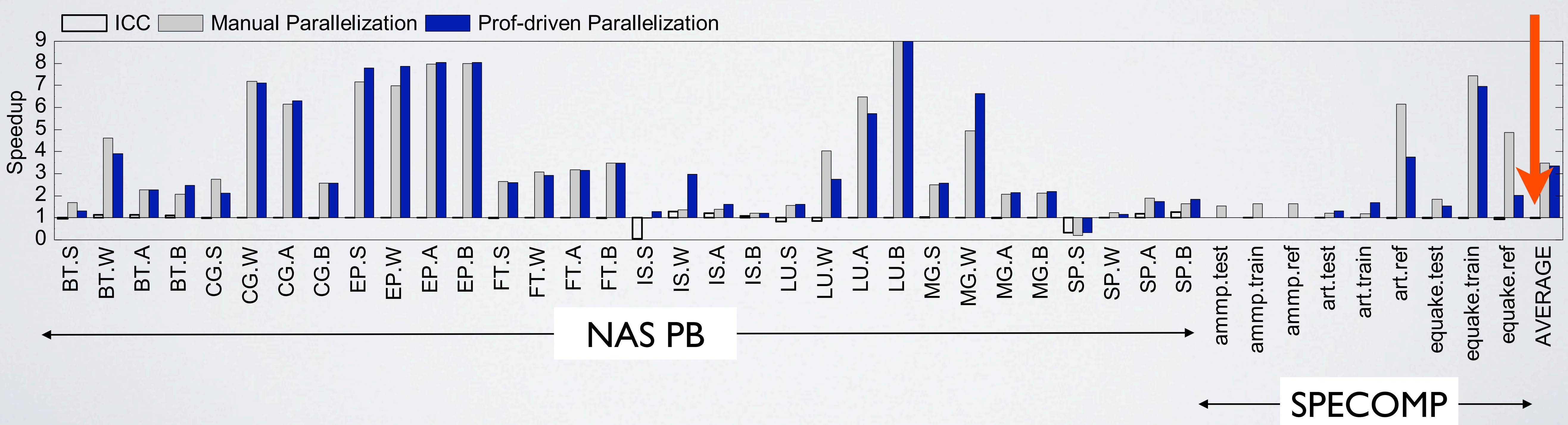
Safety

- Inherently unsafe, but surprisingly **no FP**
- Even when trained on the **smallest** dataset

Application	FP	FN
bt	0	0
cg	0	0
ep	0	0
ft	0	0
is	0	0
lu	0	0
mg	0	3
sp	0	0
equake	0	0
art	0	0
ammp	0	1

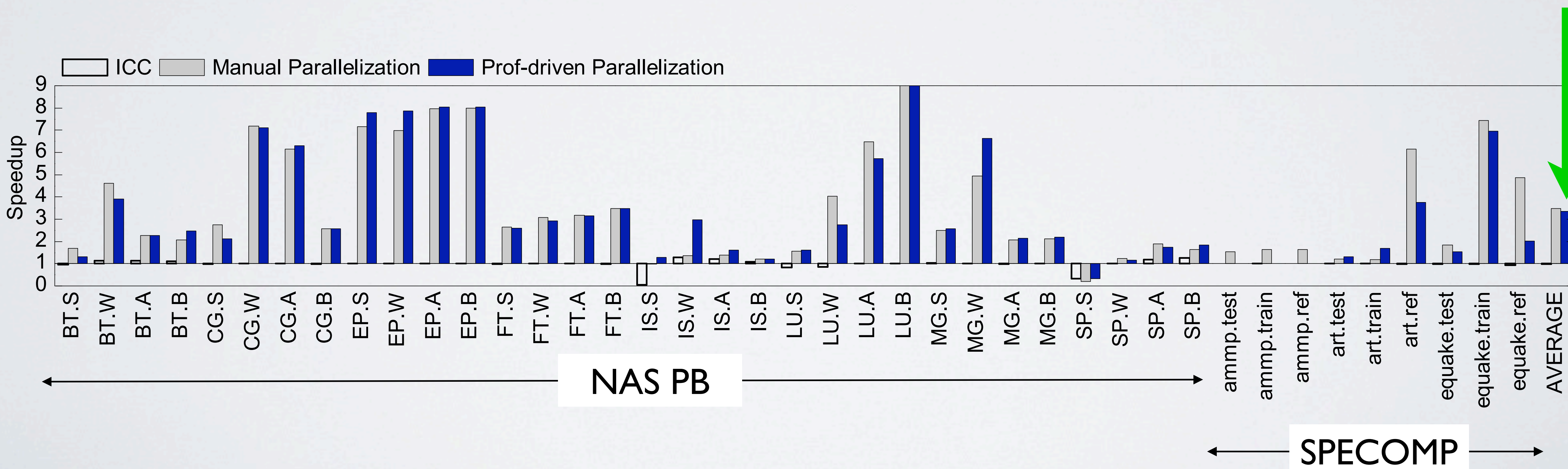
Speedup (Intel Xeon)

- Intel ICC **fails** to deliver any performance gain
- Even **slowdown** for some benchmarks



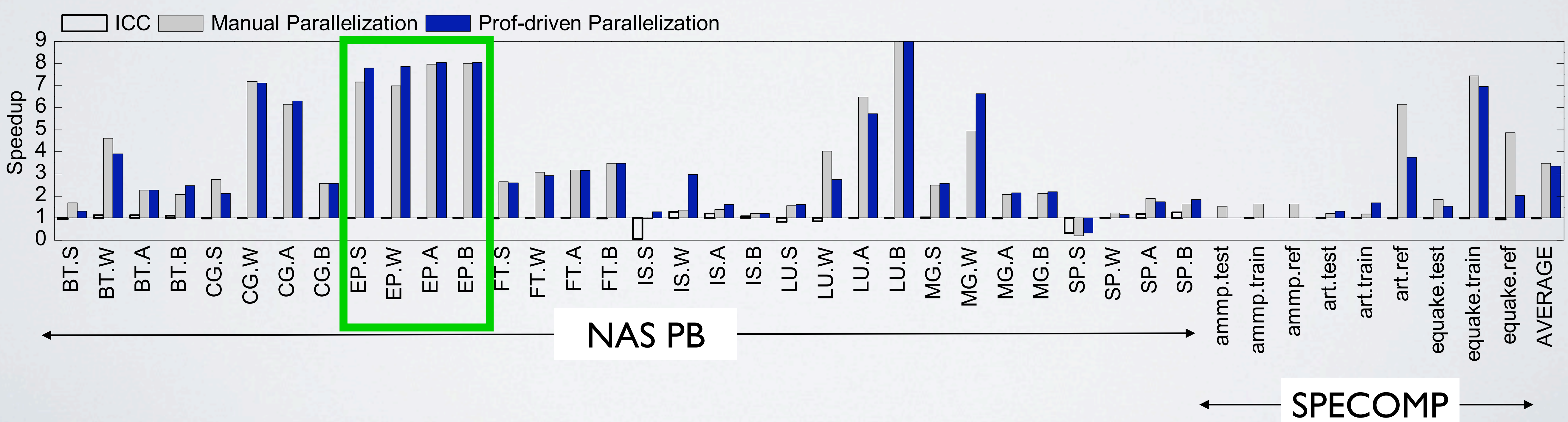
Speedup (Intel Xeon)

- Profile-driven parallelization achieves **96%** of the performance of manually parallelized benchmarks!



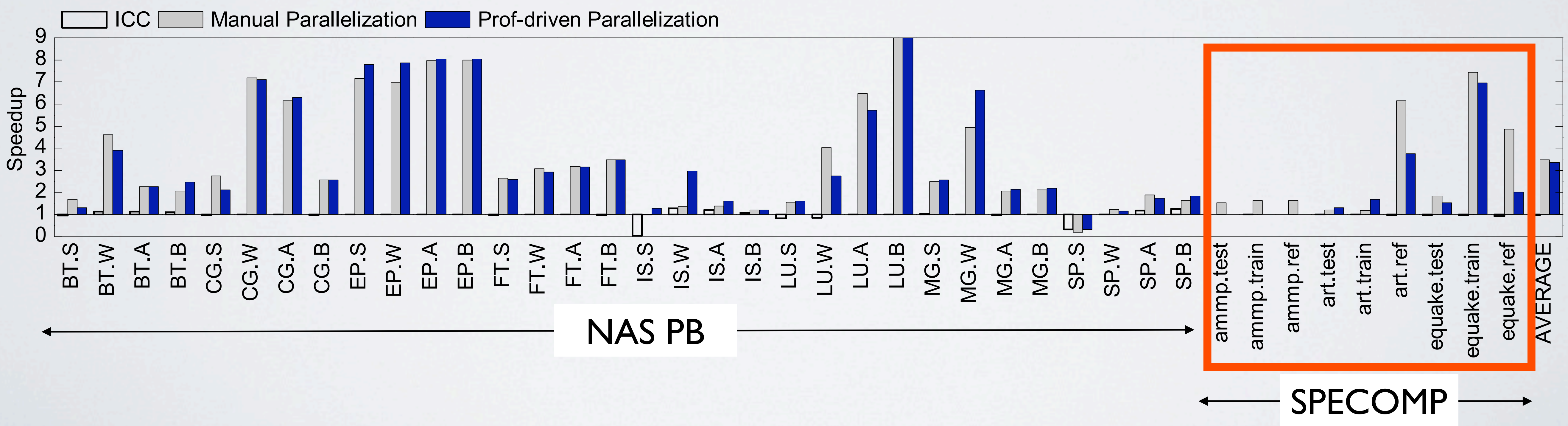
Speedup (Intel Xeon)

- EP is embarrassingly parallel, still ICC fails completely
- Profile-driven parallelisation detects critical loop



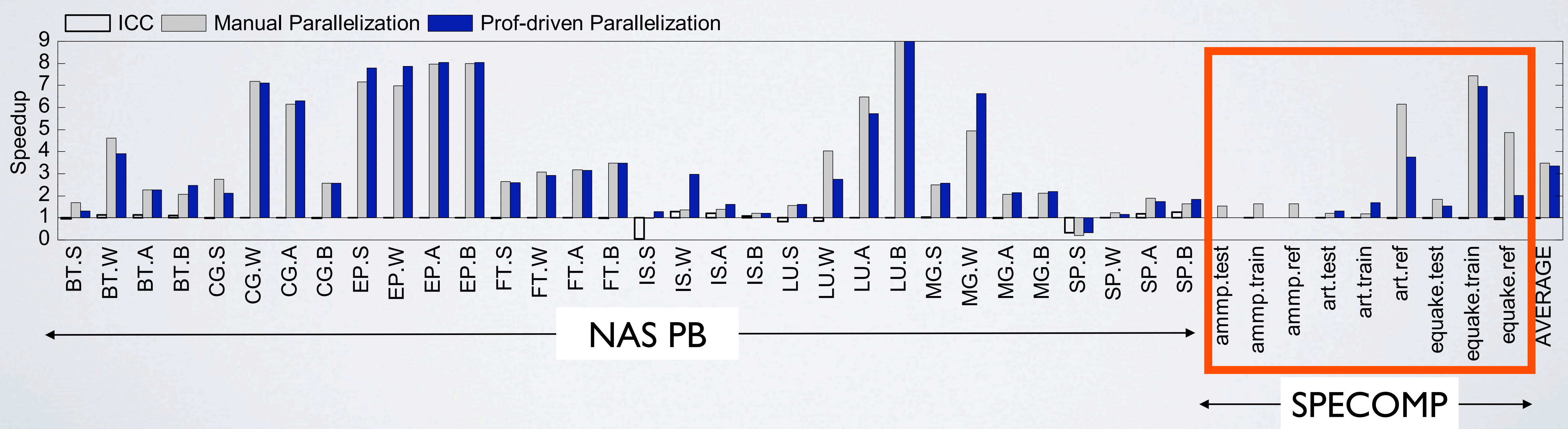
Speedup (Intel Xeon)

- SPECOMP 2001 benchmarks include additional sequential optimisations besides parallelisation



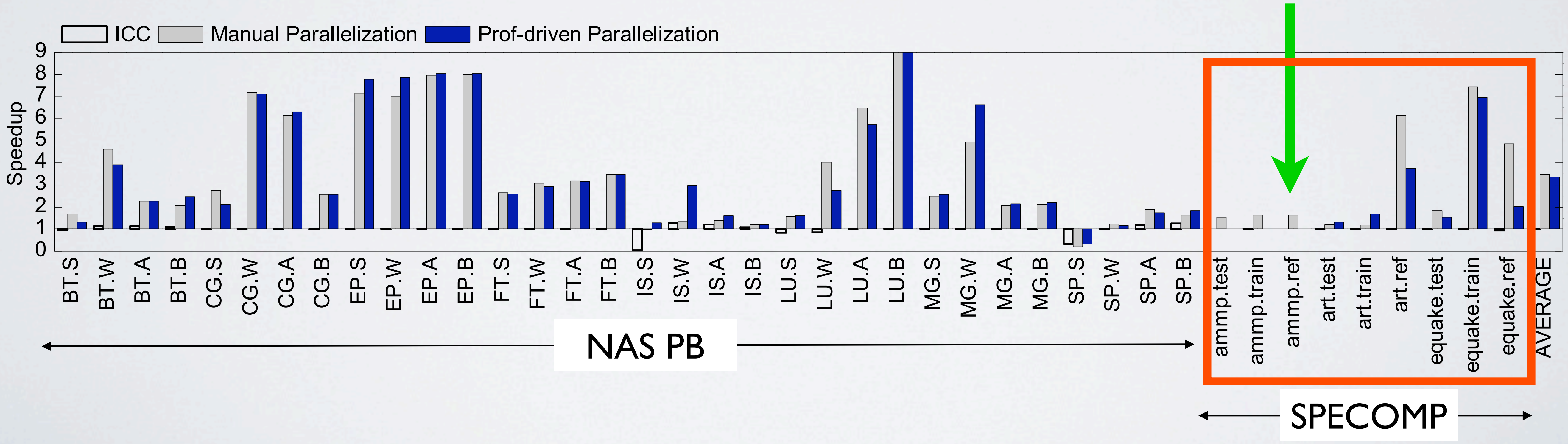
Speedup (Intel Xeon)

- SPECOMP single-threaded has average speedup of **2x** over SPECFP due to sequential optimisations



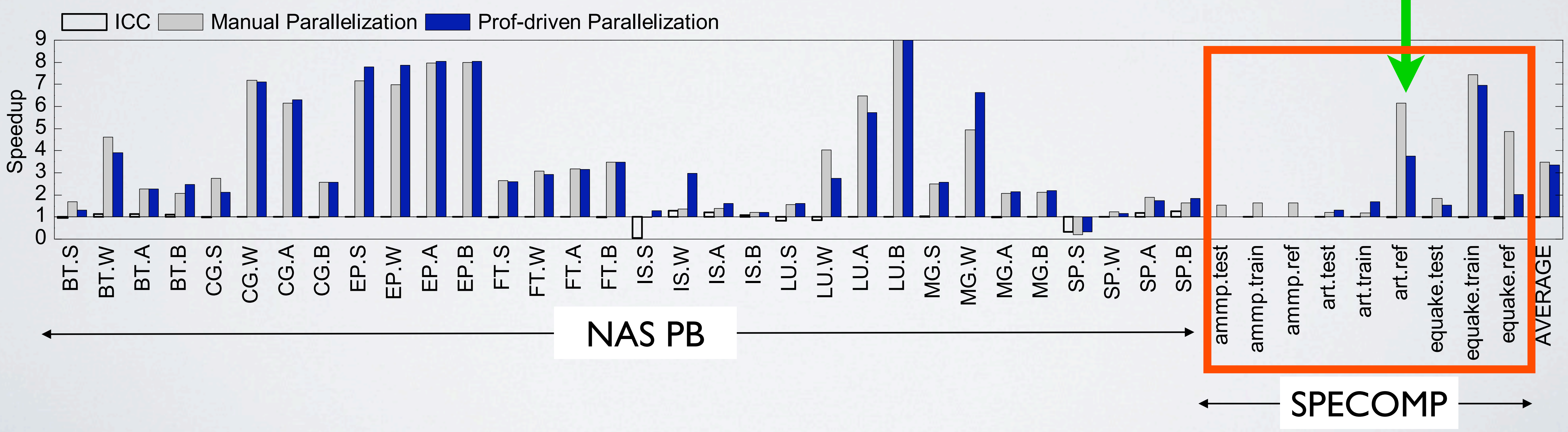
Speedup (Intel Xeon)

- ammp: critical loop not parallelised by profile-driven technique. Misclassification by ML.
- Manual parallelisation: **1.6x** on 8 cores



Speedup (Intel Xeon)

- SPECOMP.art **3.34x** with **1 thread**
- Profile-driven parallelisation delivers **4x** speedup without sequential optimisation



PART 3: EXTRACTION OF PIPELINE PARALLELISM

Observations

- There is more parallelism available beyond parallel FOR loops
  - Programmers exploit coarse-grained parallelism routinely
  - Auto-parallelising compilers don't!
- Parallel Design Patterns
  - Static: Pipelines, Task Graphs
  - Dynamic: Task Farms, Divide & Conquer, ...
- Serious Programme Restructuring Required
  - Let's Do It!

Motivating Example

EEMBC mp3player

```
while(end) {
  /* ...input... */
  decode_info(&bs, &fr_ps);
  /* ...input... */
  III_get_side_info(&bs, &III_side_info, &fr_ps);

  for (gr = 0; gr < max_gr; gr++) {
    for (ch = 0; ch < stereo; ch++) {
      III_get_scale_factors(gr, ch, ...);
      III_hufman_decode(gr, ch, ...);
      III_dequantize_sample(gr, ch, ...);
    } /* ch */

    III_stereo(gr, ...);

    for (ch = 0; ch < stereo; ch++) {
      III_reorder(ch, gr, ...);
      III_antialias(ch, gr, ...);

      for (sb = 0; sb < SBLIMIT; sb++) {
        III_hybrid(sb, ch, ...);
      } /* ss  */

      for (ss = 0; ss < SSLIMIT; ss++) {
        for (sb = 0; sb < SBLIMIT; sb++) {
          if ((ss % 2) && (sb % 2))
            polyPhaseIn[sb] = -hybridOut[sb][ss];
          else
            polyPhaseIn[sb] = hybridOut[sb][ss];
        } /* sb */
        clip += SubBandSynthesis(ch, ss, ...);
      } /* ss  */
    } /* ch */
  } /* gr */

  out_fifo(*pcm_sample, ...);
} /* while */
```

Algorithmic components

input, header info	**2%**
Huffman decoding	**5%**
dequantize	**44%**
stereo	**<1%**
reorder	**<1%**
antialias	**<1%**
hybrid	**22%**
subband synthesis	**24%**
output	**2%**

Motivating Example

EEMBC mp3player

```
while(end) {
  /* ...input... */
  decode_info(&bs, &fr_ps);
  /* ...input... */
  III_get_side_info(&bs, &III_side_info, &fr_ps);

  for (gr = 0; gr < max_gr; gr++) {
    for (ch = 0; ch < stereo; ch++) {
      III_get_scale_factors(gr, ch, ...);
      III_hufman_decode(gr, ch, ...);
      III_dequantize_sample(gr, ch, ...);
    } /* ch */

    III_stereo(gr, ...);

    for (ch = 0; ch < stereo; ch++) {
      III_reorder(ch, gr, ...);
      III_antialias(ch, gr, ...);

      for (sb = 0; sb < SBLIMIT; sb++) {
        III_hybrid(sb, ch, ...);
      } /* ss  */

      for (ss = 0; ss < SSLIMIT; ss++) {
        for (sb = 0; sb < SBLIMIT; sb++) {
          if ((ss % 2) && (sb % 2))
            polyPhaseIn[sb] = -hybridOut[sb][ss];
          else
            polyPhaseIn[sb] = hybridOut[sb][ss];
        } /* sb */
        clip += SubBandSynthesis(ch, ss, ...);
      } /* ss  */
    } /* ch */
  } /* gr */

  out_fifo(*pcm_sample, ...);
} /* while */
```

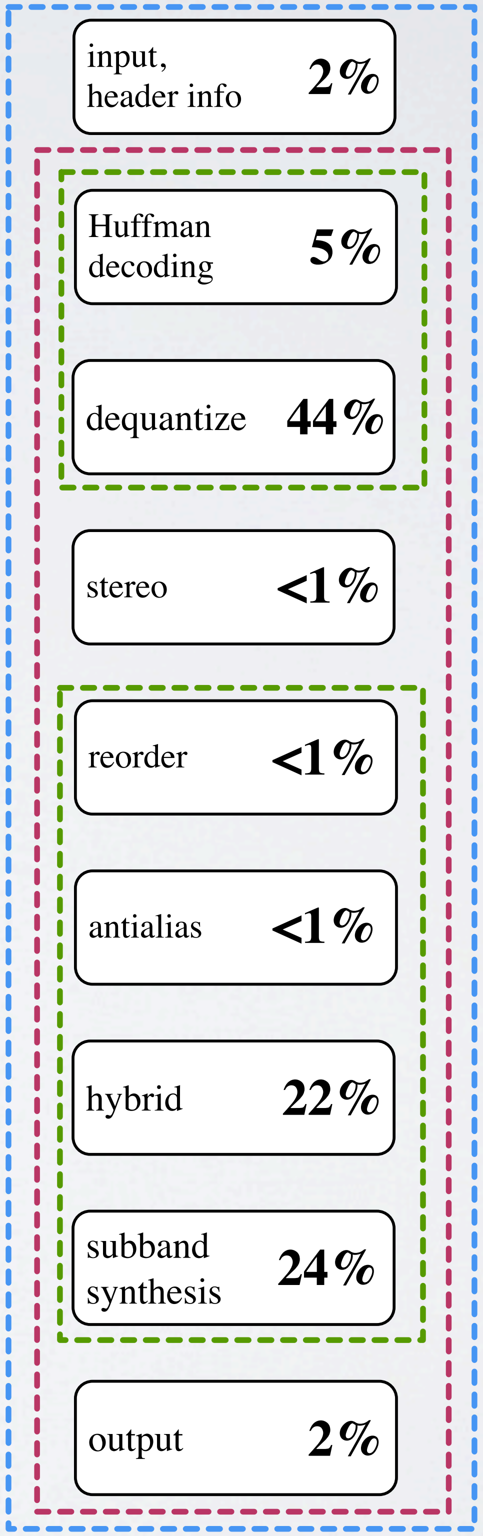
Level 1

Level 2

Level 3

Level 3

Algorithmic components



Motivating Example

EEMBC mp3player

```
while(end) {
  /* ...input... */
  decode_info(&bs, &fr_ps);
  /* ...input... */
  III_get_side_info(&bs, &III_side_info, &fr_ps);

  for (gr = 0; gr < max_gr; gr++) {
    for (ch = 0; ch < stereo; ch++) {
      III_get_scale_factors(gr, ch, ...);
      III_hufman_decode(gr, ch, ...);
      III_dequantize_sample(gr, ch, ...);
    } /* ch */

    III_stereo(gr, ...);

    for (ch = 0; ch < stereo; ch++) {
      III_reorder(ch, gr, ...);
      III_antialias(ch, gr, ...);

      for (sb = 0; sb < SBLIMIT; sb++) {
        III_hybrid(sb, ch, ...);
      } /* ss */

      for (ss = 0; ss < SSLIMIT; ss++) {
        for (sb = 0; sb < SBLIMIT; sb++) {
          if ((ss % 2) && (sb % 2))
            polyPhaseIn[sb] = -hybridOut[sb][ss];
          else
            polyPhaseIn[sb] = hybridOut[sb][ss];
        } /* sb */
        clip += SubBandSynthesis(ch, ss, ...);
      } /* ss */
    } /* ch */
  } /* gr */

  out_fifo(*pcm_sample, ...);
} /* while */
```

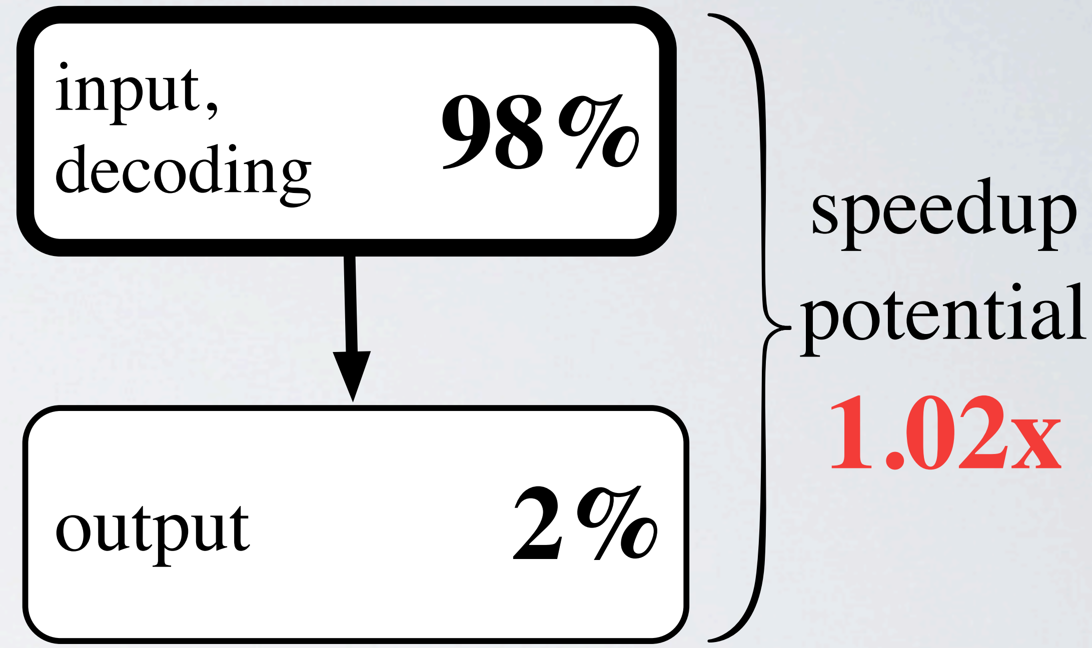
Level 1

Level 2

Level 3

Level 3

Single-level partitioning



Single-level pipeline is inefficient!

Motivating Example

EEMBC mp3player

```
while(end) {
  /* ...input... */
  decode_info(&bs, &fr_ps);
  /* ...input... */
  III_get_side_info(&bs, &III_side_info, &fr_ps);

  for (gr = 0; gr < max_gr; gr++) {
    for (ch = 0; ch < stereo; ch++) {
      III_get_scale_factors(gr, ch, ...);
      III_hufman_decode(gr, ch, ...);
      III_dequantize_sample(gr, ch, ...);
    } /* ch */

    III_stereo(gr, ...);

    for (ch = 0; ch < stereo; ch++) {
      III_reorder(ch, gr, ...);
      III_antialias(ch, gr, ...);

      for (sb = 0; sb < SBLIMIT; sb++) {
        III_hybrid(sb, ch, ...);
      } /* ss */

      for (ss = 0; ss < SSLIMIT; ss++) {
        for (sb = 0; sb < SBLIMIT; sb++) {
          if ((ss % 2) && (sb % 2))
            polyPhaseIn[sb] = -hybridOut[sb][ss];
          else
            polyPhaseIn[sb] = hybridOut[sb][ss];
        } /* sb */
        clip += SubBandSynthesis(ch, ss, ...);
      } /* ss */
    } /* ch */
  } /* gr */

  out_fifo(*pcm_sample, ...);
} /* while */
```

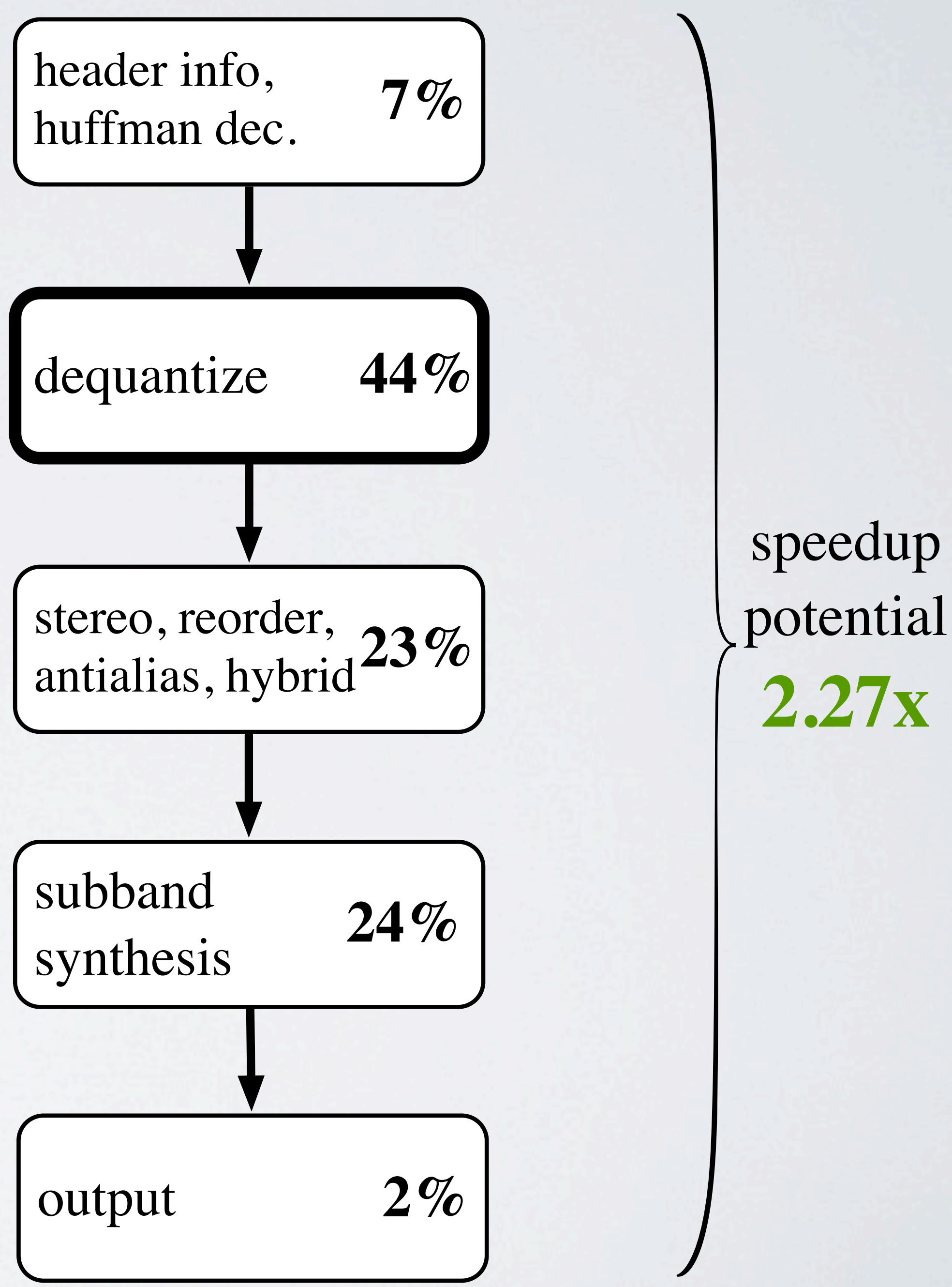
Level 1

Level 2

Level 3

Level 3

Multi-level partitioning



Motivating Example

EEMBC mp3player

```
while(end) {
  /* ...input... */
  decode_info(&bs, &fr_ps);
  /* ...input... */
  III_get_side_info(&bs, &III_side_info, &fr_ps);

  for (gr = 0; gr < max_gr; gr++) {
    for (ch = 0; ch < stereo; ch++) {
      III_get_scale_factors(gr, ch, ...);
      III_hufman_decode(gr, ch, ...);
      III_dequantize_sample(gr, ch, ...);
    } /* ch */

    III_stereo(gr, ...);

    for (ch = 0; ch < stereo; ch++) {
      III_reorder(ch, gr, ...);
      III_antialias(ch, gr, ...);

      for (sb = 0; sb < SBLIMIT; sb++) {
        III_hybrid(sb, ch, ...);
      } /* ss */

      for (ss = 0; ss < SSLIMIT; ss++) {
        for (sb = 0; sb < SBLIMIT; sb++) {
          if ((ss % 2) && (sb % 2))
            polyPhaseIn[sb] = -hybridOut[sb][ss];
          else
            polyPhaseIn[sb] = hybridOut[sb][ss];
        } /* sb */
        clip += SubBandSynthesis(ch, ss, ...);
      } /* ss */
    } /* ch */
  } /* gr */

  out_fifo(*pcm_sample, ...);
} /* while */
```

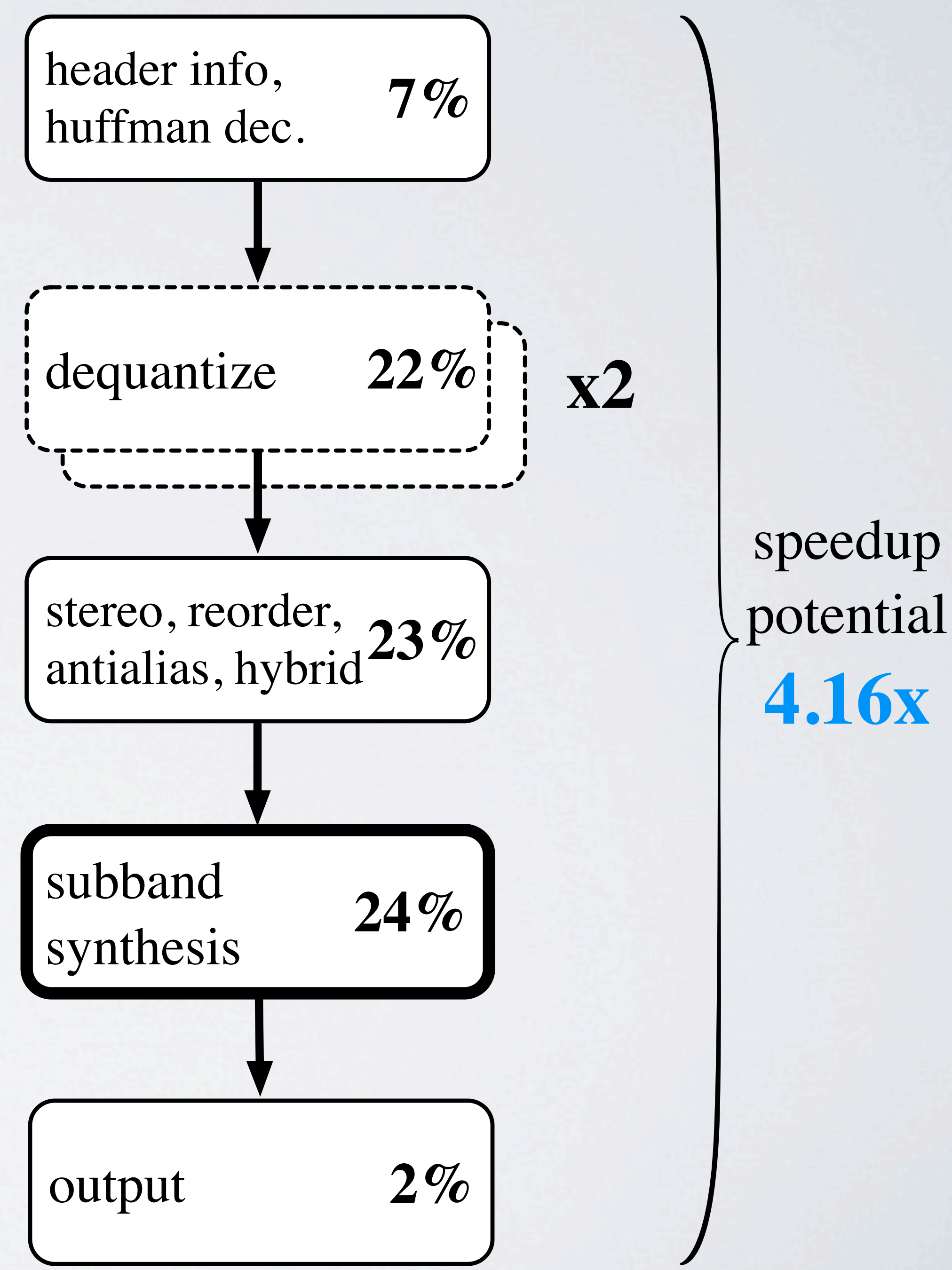
Level 1

Level 2

Level 3

Level 3

Multi-level + Replication Partitioning



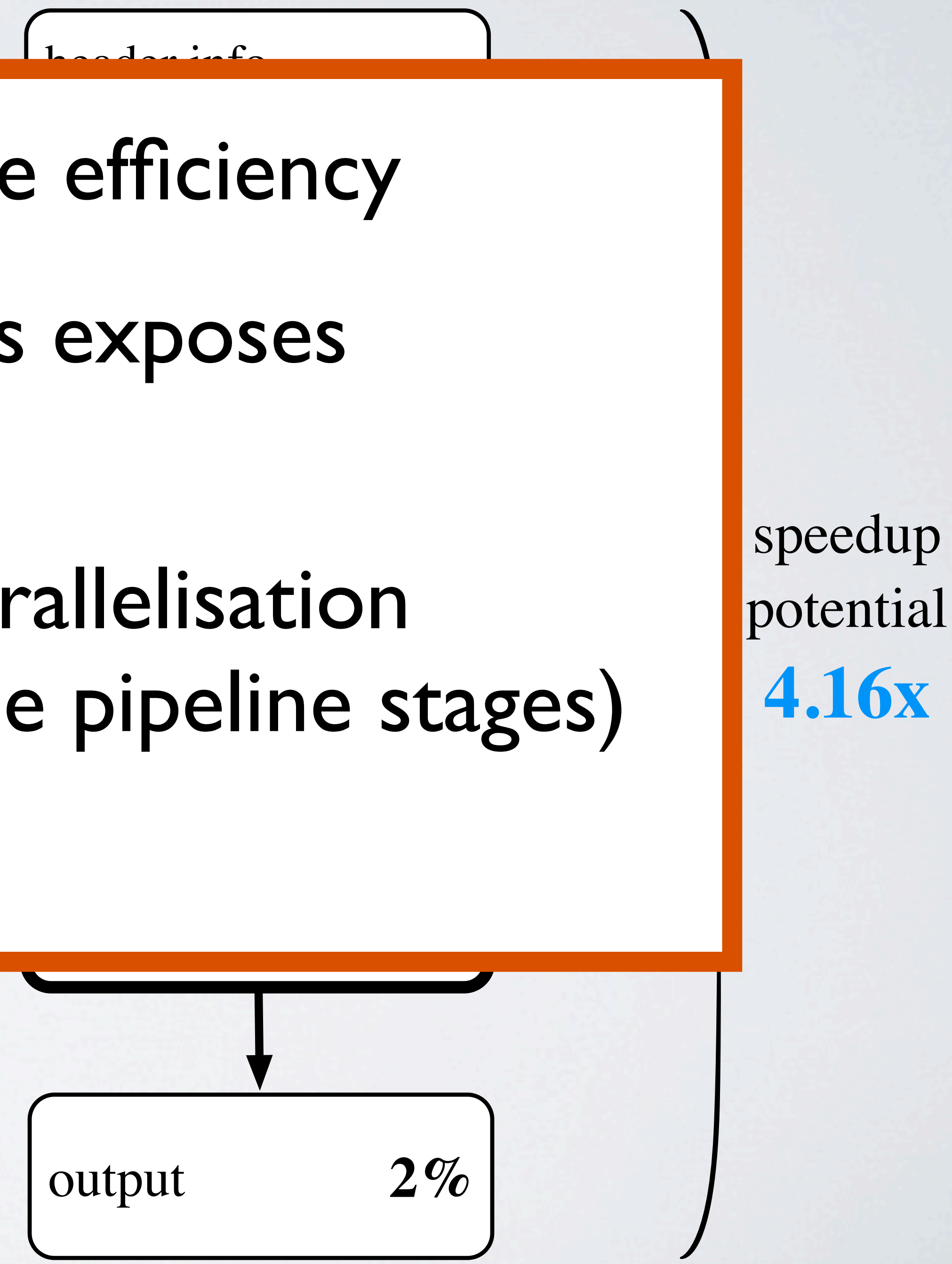
Motivating Example

EEMBC mp3player

Level 1

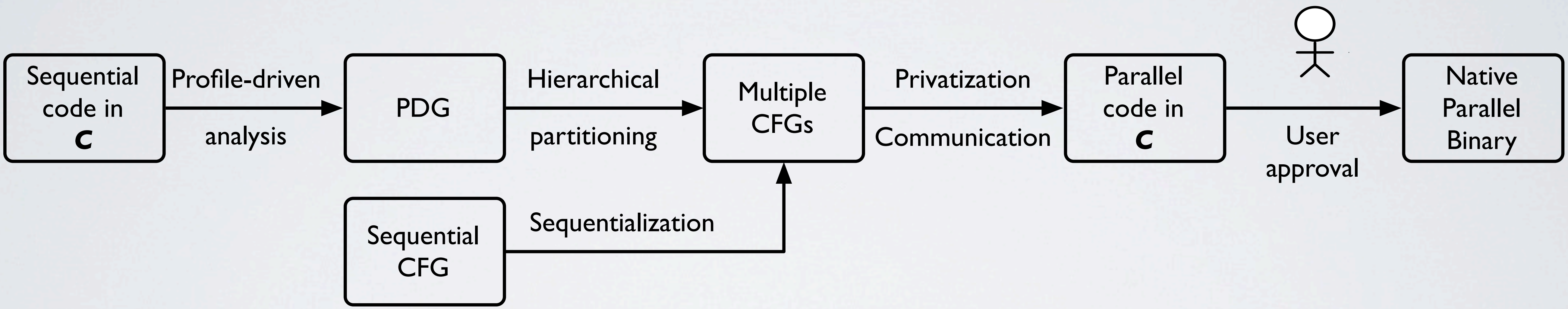
```
while(end) {
  /* ...input... */
  decode_info(&bs, &fr_ps);
  /* .
  III_

  for
    fo

    }

    II

    fo

        else
          polyPhaseIn[sb] = hybridOut[sb][ss];
      } /* sb */
      clip += SubBandSynthesis(ch, ss, ...);
    } /* ss */
  } /* ch */
} /* gr */

  out_fifo(*pcm_sample, ...);
} /* while */
```

Multi-level + Replication Partitioning



- **Hierachical** pipelines increase efficiency
- **Replication** of pipeline stages exposes additional parallelism
- **Orthogonal** to traditional parallelisation approaches (parallel loops inside pipeline stages)

Approach


Sequential code in C
Profile-driven
analysis
PDG
Hierarchical
partitioning
Multiple CFGs
Privatization
Communication
Parallel code in C
User approval
Native Parallel Binary
Sequential CFG
Sequentialization

Partitioning Strategy

- Pipeline performance determined by the **slowest stage**
- Apply **code transformations** only to the slowest stage to uncover further parallelism

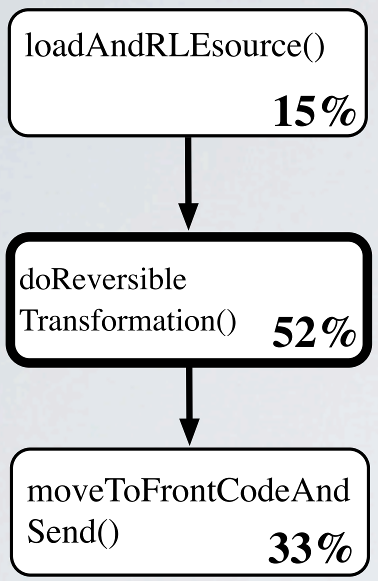
Partitioning Algorithm

- Top-down approach: loops and functions ***folded***
- Preprocess PDG of the loop:
  - Form Strongly Connected Components
  - Focus on slowest component:
    - If data-parallel (i) greedily augment it, and (ii) replicate until another component is the slowest
    - If not data-parallel try to reduce the execution time by ***unfolding*** loop/function nodes in the component
- Partition pipeline using the load of the slowest component as threshold

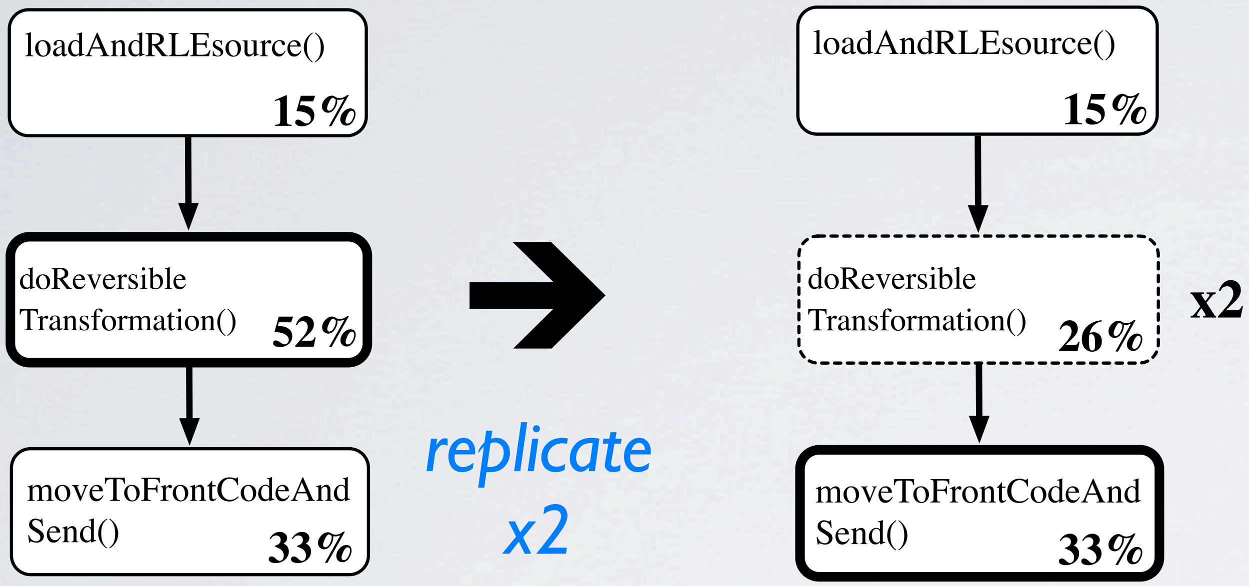
Partitioning Operations

- Loop/Function unfolding
  - "Opening up" loop/function for hierarchical partitioning
- Replication
  - Duplication of partitioning unit for parallel execution
- Split function
  - Insert pipeline stage boundary within function body
- Augment block
  - Merge separate blocks into single pipeline stage

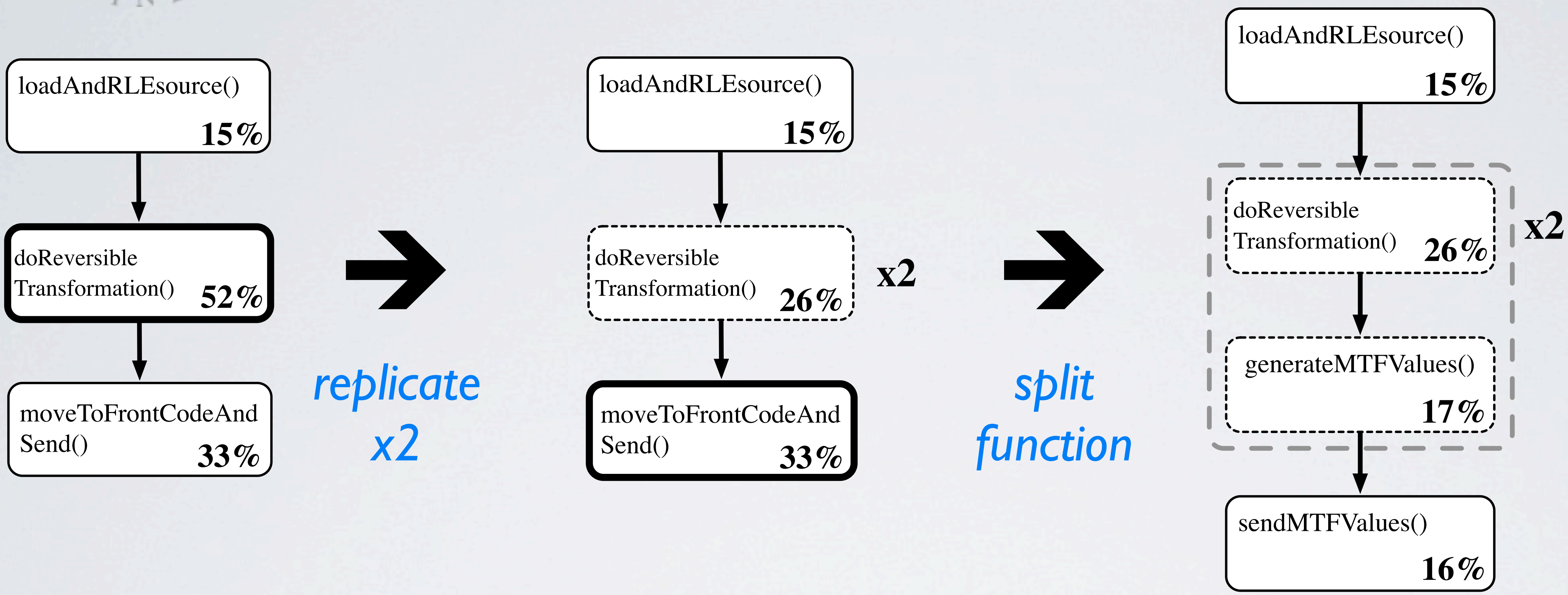
Example


loadAndRLEsource()
15%
doReversible
Transformation()
52%
moveToFrontCodeAnd
Send()
33%

Example

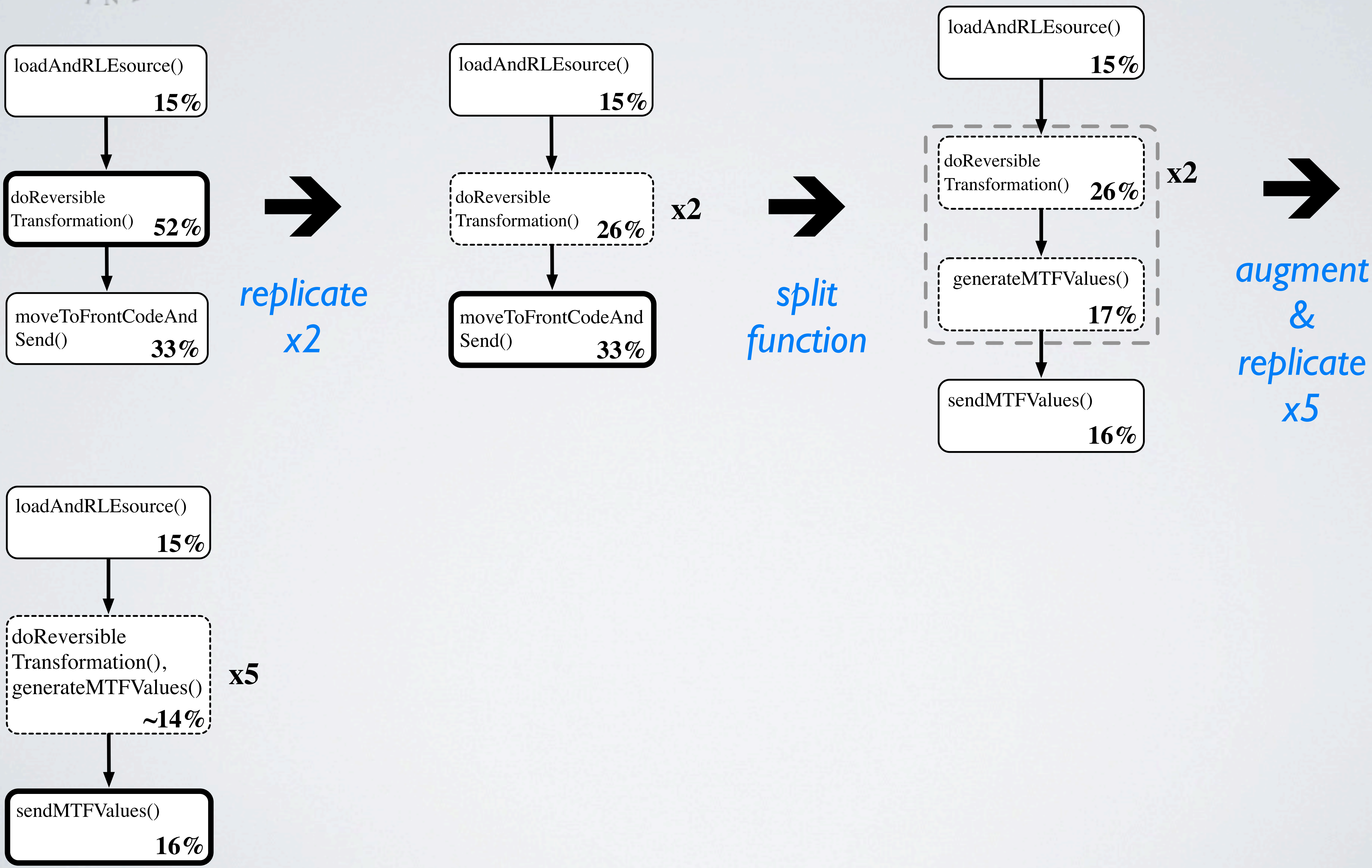

loadAndRLEsource()
15%
doReversible
Transformation()
52%
moveToFrontCodeAnd
Send()
33%
replicate
x2
loadAndRLEsource()
15%
doReversible
Transformation()
26%
x2
moveToFrontCodeAnd
Send()
33%

Example



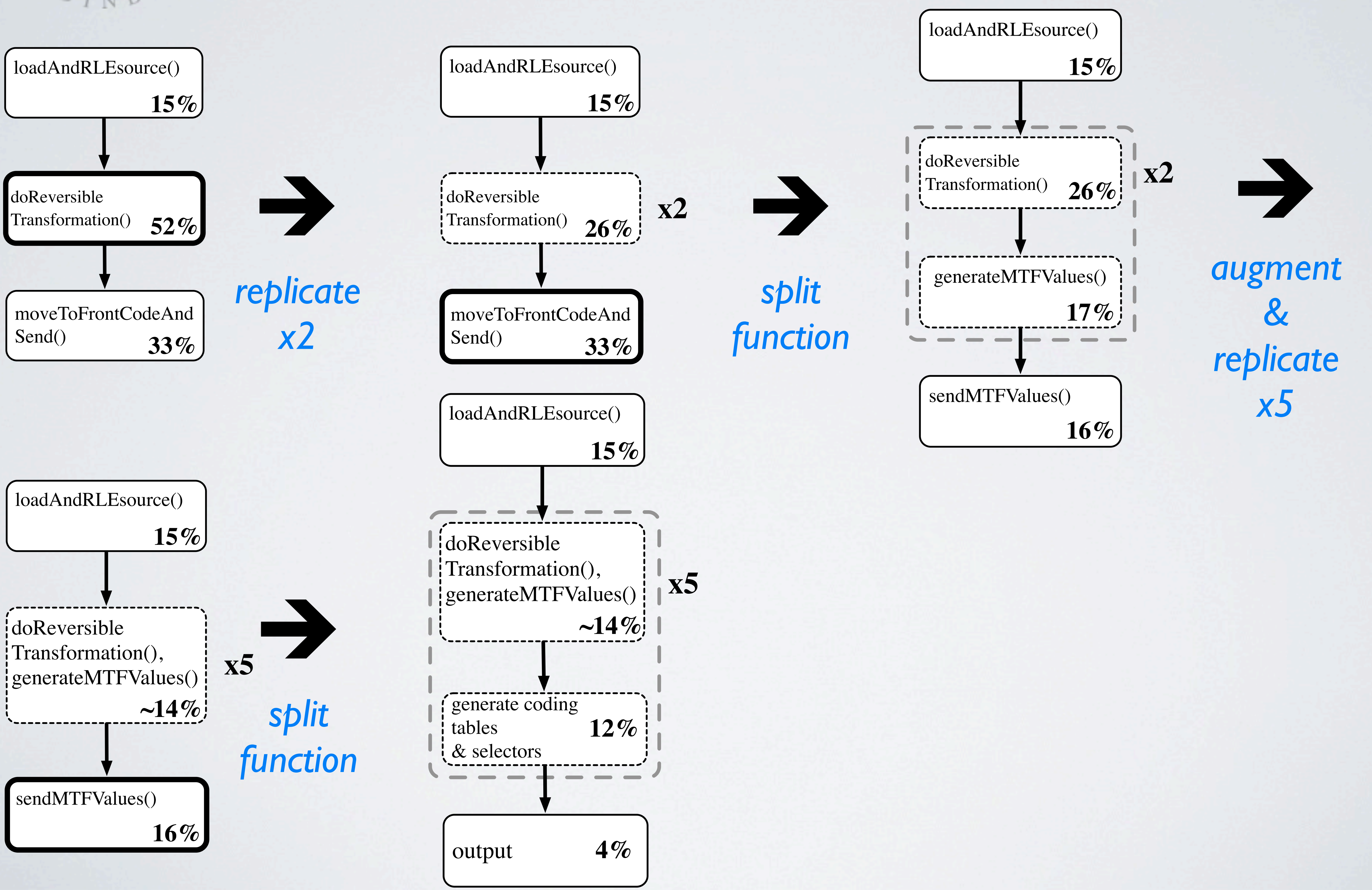
loadAndRLEsource()
15%
doReversible
Transformation()
52%
moveToFrontCodeAnd
Send()
33%
replicate
x2
loadAndRLEsource()
15%
doReversible
Transformation()
26%
x2
moveToFrontCodeAnd
Send()
33%
split
function
loadAndRLEsource()
15%
doReversible
Transformation()
26%
x2
generateMTFValues()
17%
sendMTFValues()
16%

Example



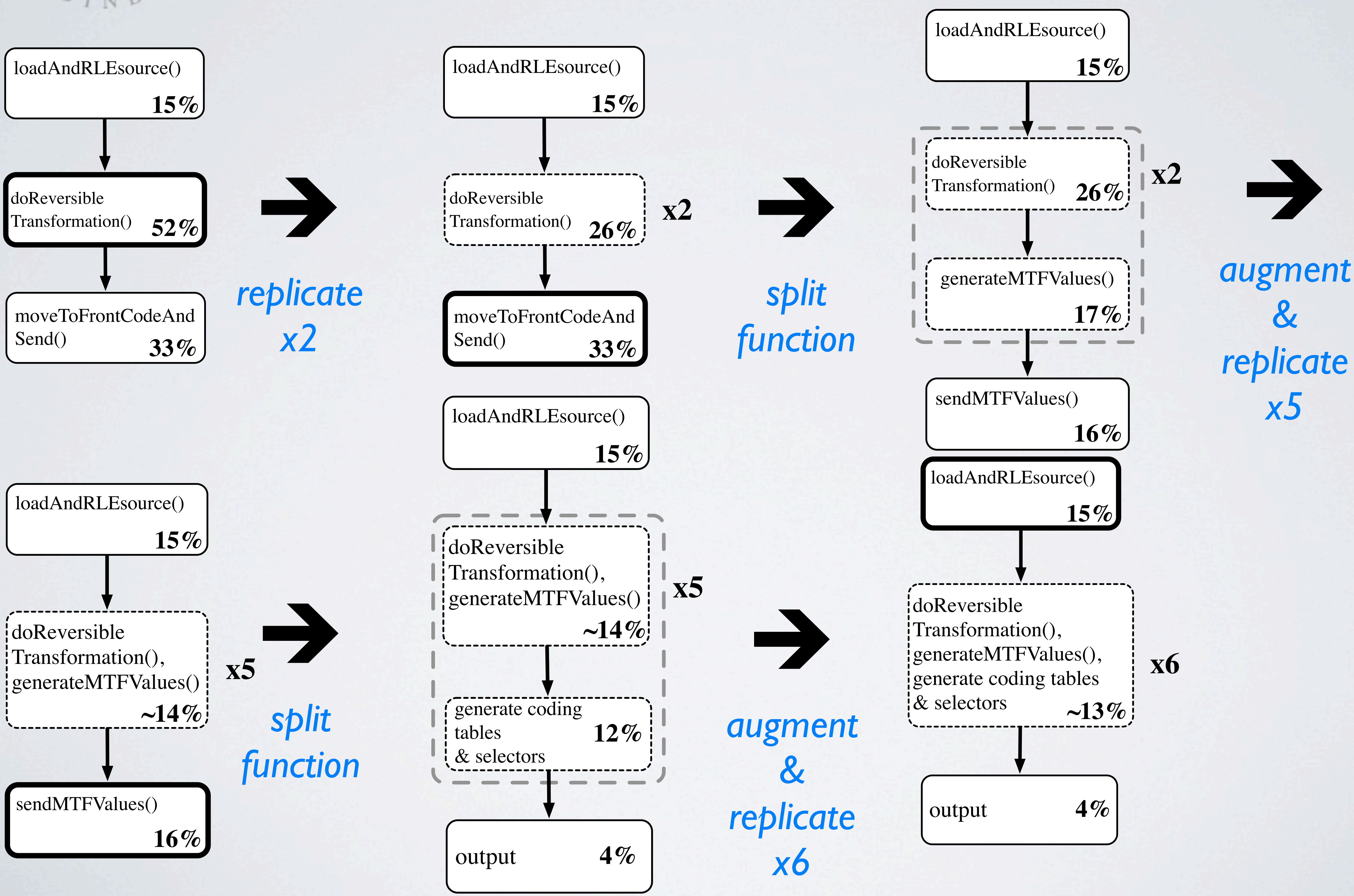
loadAndRLEsource()
15%
doReversible
Transformation()
52%
moveToFrontCodeAnd
Send()
33%
replicate
x2
loadAndRLEsource()
15%
doReversible
Transformation()
26%
x2
moveToFrontCodeAnd
Send()
33%
split
function
loadAndRLEsource()
15%
doReversible
Transformation()
26%
x2
generateMTFValues()
17%
sendMTFValues()
16%
augment
&
replicate
x5
loadAndRLEsource()
15%
doReversible
Transformation(),
generateMTFValues()
~14%
x5
sendMTFValues()
16%

Example



loadAndRLEsource() 15%
doReversible Transformation() 52%
moveToFrontCodeAnd Send() 33%
replicate x2
loadAndRLEsource() 15%
doReversible Transformation() 26%
x2
moveToFrontCodeAnd Send() 33%
split function
loadAndRLEsource() 15%
doReversible Transformation() 26%
x2
generateMTFValues() 17%
sendMTFValues() 16%
augment & replicate x5
loadAndRLEsource() 15%
doReversible Transformation(), generateMTFValues() ~14%
x5
sendMTFValues() 16%
split function
loadAndRLEsource() 15%
doReversible Transformation(), generateMTFValues() ~14%
x5
generate coding tables & selectors 12%
output 4%

Example


loadAndRLEsource() 15%
doReversible Transformation() 52%
moveToFrontCodeAnd Send() 33%
replicate x2
loadAndRLEsource() 15%
doReversible Transformation() 26%
x2
moveToFrontCodeAnd Send() 33%
split function
loadAndRLEsource() 15%
doReversible Transformation() 26%
generateMTFValues() 17%
x2
sendMTFValues() 16%
augment & replicate x5
loadAndRLEsource() 15%
doReversible Transformation(), generateMTFValues() ~14%
x5
sendMTFValues() 16%
split function
loadAndRLEsource() 15%
doReversible Transformation(), generateMTFValues() ~14%
x5
generate coding tables & selectors 12%
output 4%
augment & replicate x6
loadAndRLEsource() 15%
doReversible Transformation(), generateMTFValues(), generate coding tables & selectors ~13%
x6
output 4%

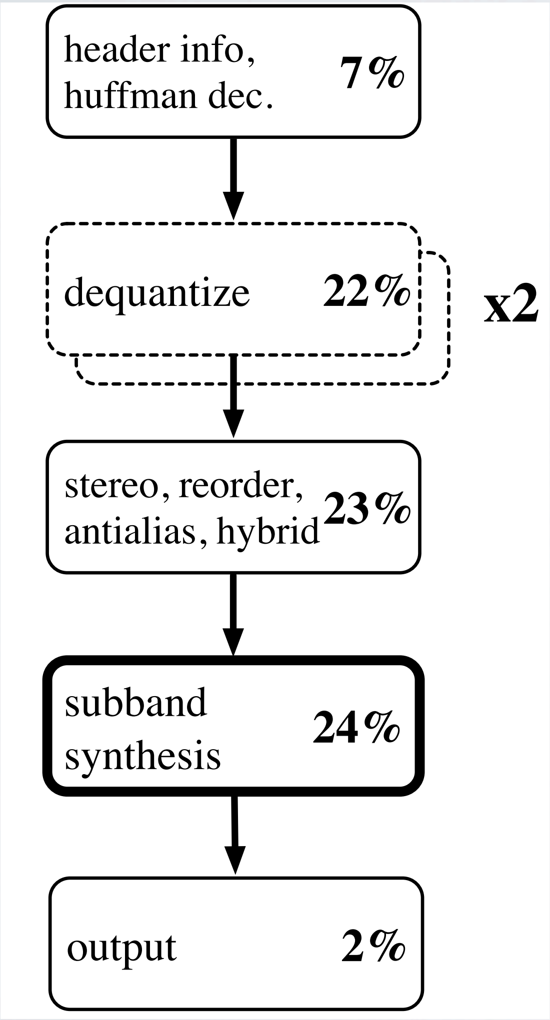
Experimental Evaluation

Application	source	lines
MP3 decode	EEMBC 2.0	20K
MPEG-2 decode	EEMBC 2.0	23K
JPEG encode	EEMBC 2.0	22K
bzip2 compress	SPEC CPU2000	5K

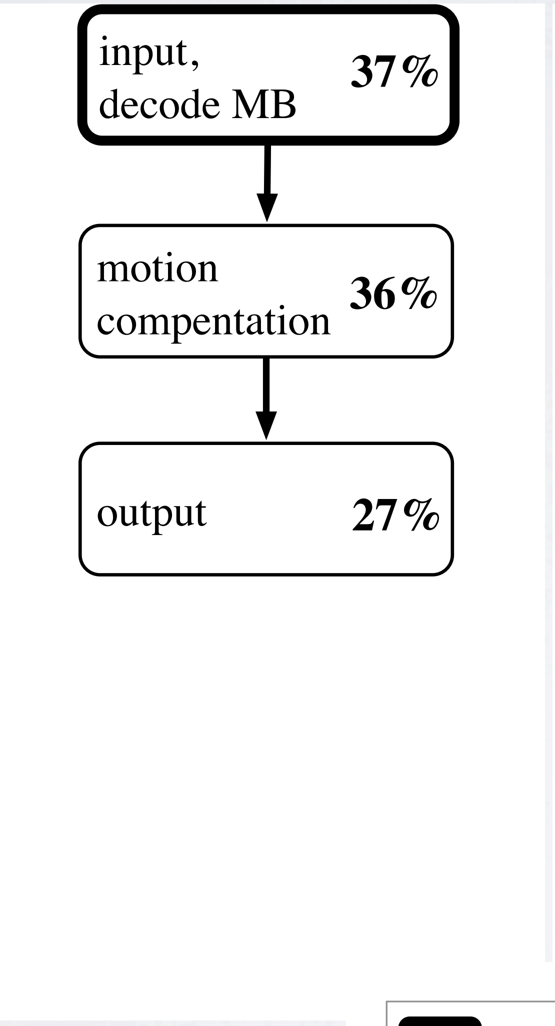
Evaluation platform	
Hardware	Dual Socket, Intel Xeon X5450 @ 3.00GHz 2 Quad-cores, 8 cores in total SSE2, SSE3 and SSE4.1 extensions 6Mb L2-cache shared/2 cores (12Mb/chip) 16Gb DDR2 SDRAM
O.S.	64-bit Scientific Linux kernel 2.6.9-55 x86_64
Compiler	GNU GCC 4.4.1 -O3 -march=core2

Extracted Pipelines

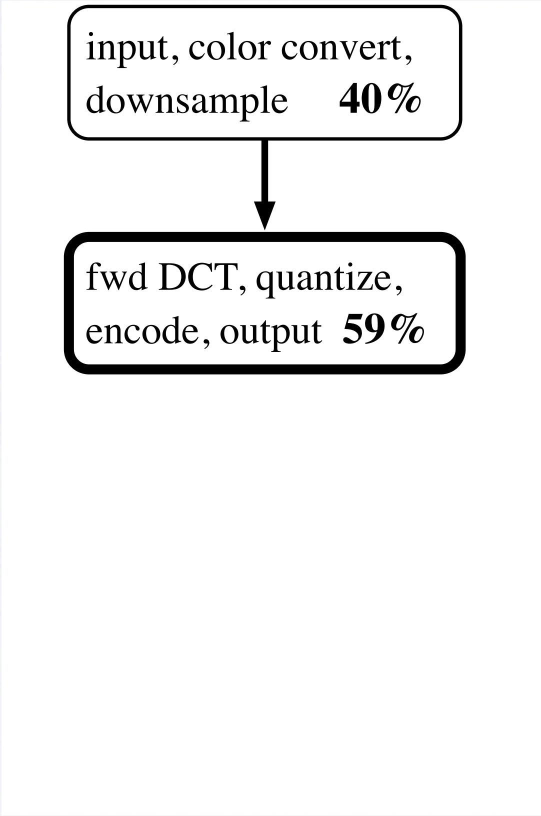
<u>MP3 decoding</u>



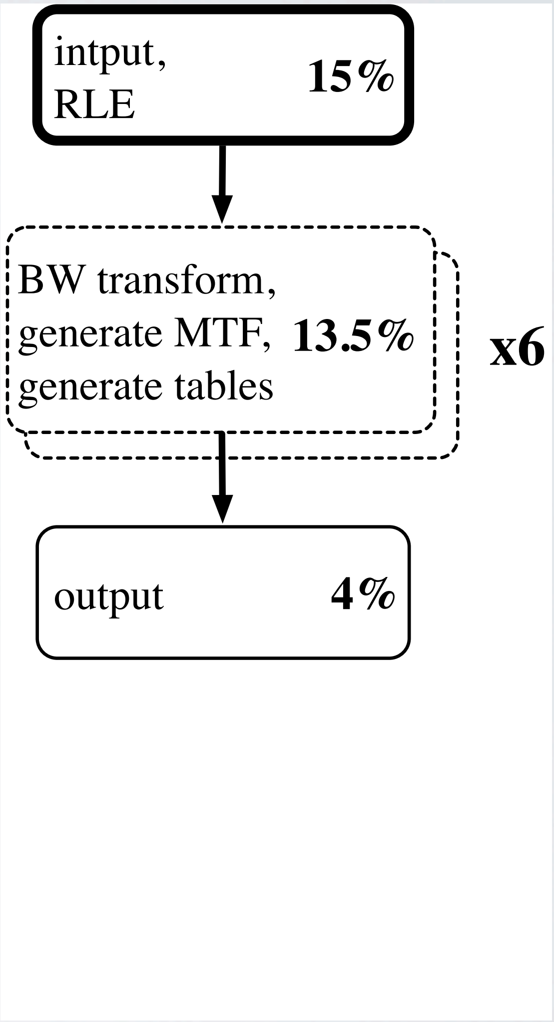
<u>MPEG-2 decoding</u>

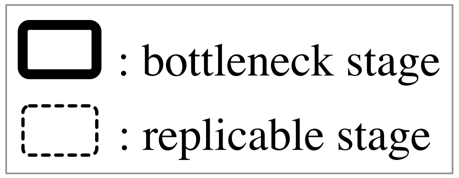


<u>JPEG encoding</u>

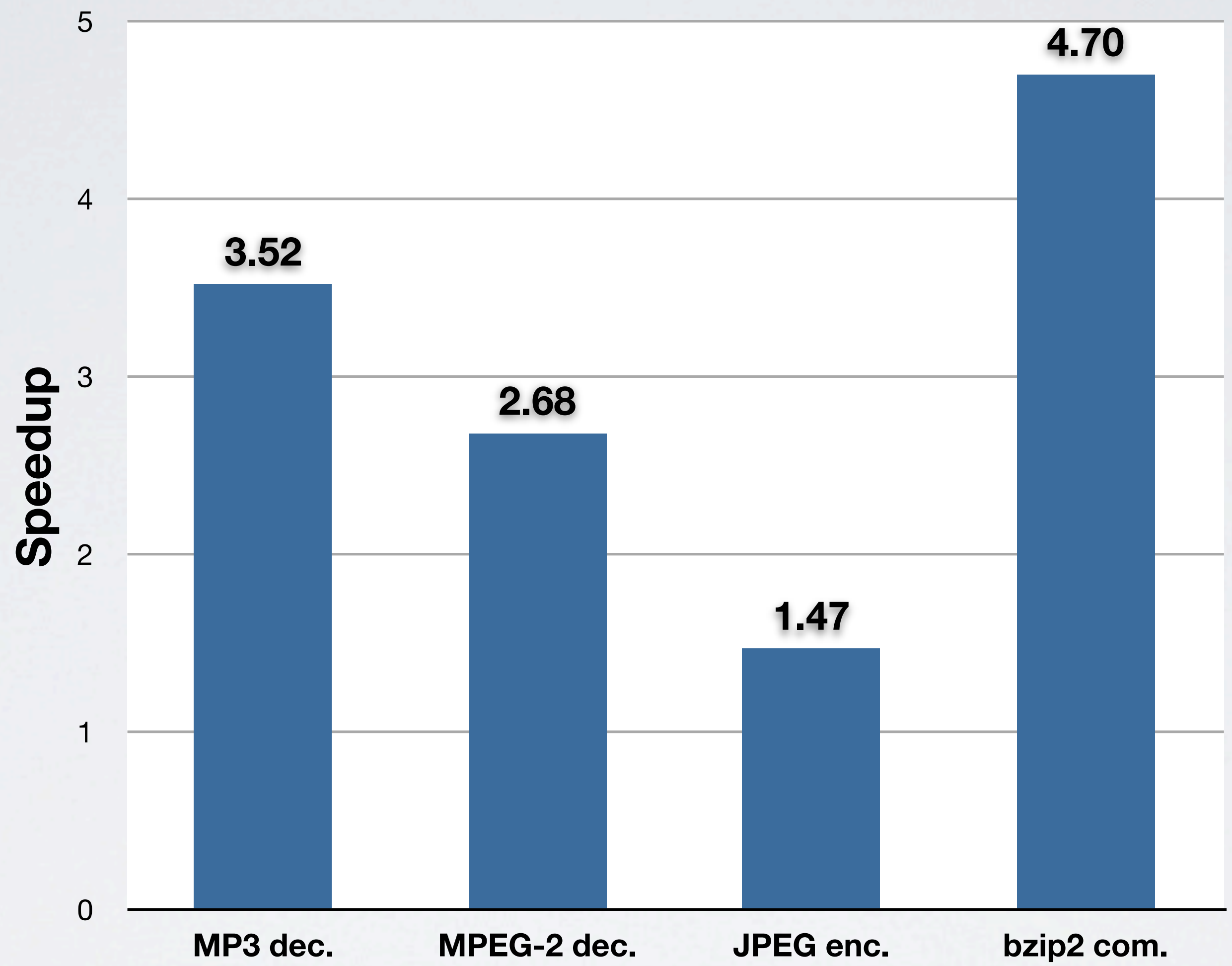


<u>bzip2 compression</u>





Performance Results



Application	replication	multi-loop	func. split	# cores	speedup
MP3 dec.	✓	✓	–	7	3.52x
MPEG-2 dec.	–	✓	✓	3	2.68x
JPEG enc.	–	✓	✓	2	1.47x
bzip2 com.	✓	–	✓	8	4.70x

Further Details

- Sequentialisation of the PDG
- Data privatisation
- Inter-thread communication
- Dynamic memory disambiguation
- Pipeline runtime system

Thanks

- Georgios Tournavitis (now with Intel Research)
- Zheng Wang
- Tobias Edler von Koch
- Igor Böhm
- Damon Fenacci
- Alastair Murray
- Daniel Powell
- Stephen Kyle
- Harry Wagstaff
- Miles Gould
- and my colleagues Mike O'Boyle & Nigel Topham

Summary

- Serious demand for parallelisation tool support
- Static analysis are too conservative
  - Profile driven analyses detect more parallelism, but require additional manual checking
- Mapping of SW parallelism to HW parallelism is non-intuitive and depends on target platform
  - Successful application of machine learning
- More scope for parallelisation beyond FOR loops
  - Start exploiting parallel design patterns

Other Interests

- Everything Parallel
- Code Generation for Embedded Processors
- Fast Instruction Set Simulation
- Parallel JIT Compilation
- Statistical Performance Modelling
- Detection of Parallel Design Patterns
- Mapping for Heterogeneous Multi-Cores

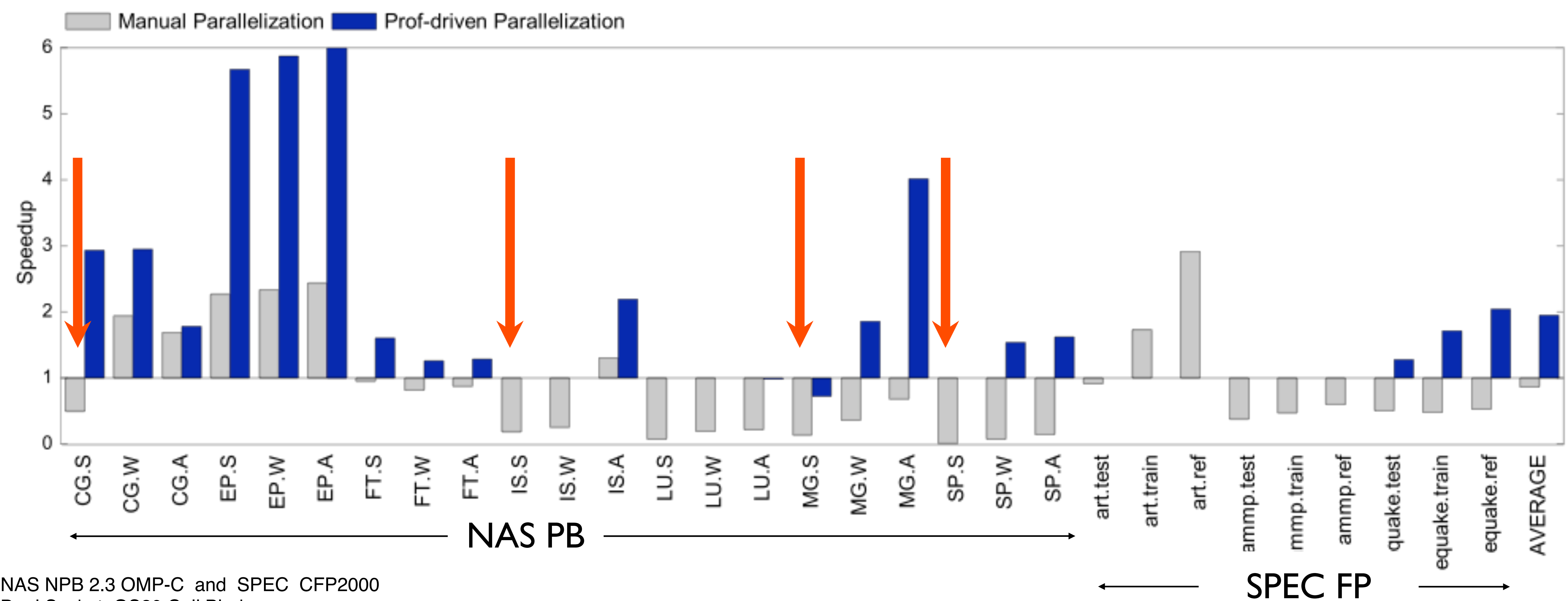
Questions?

BACKUP SLIDES

RESULTS FOR CELL

Performance on Cell

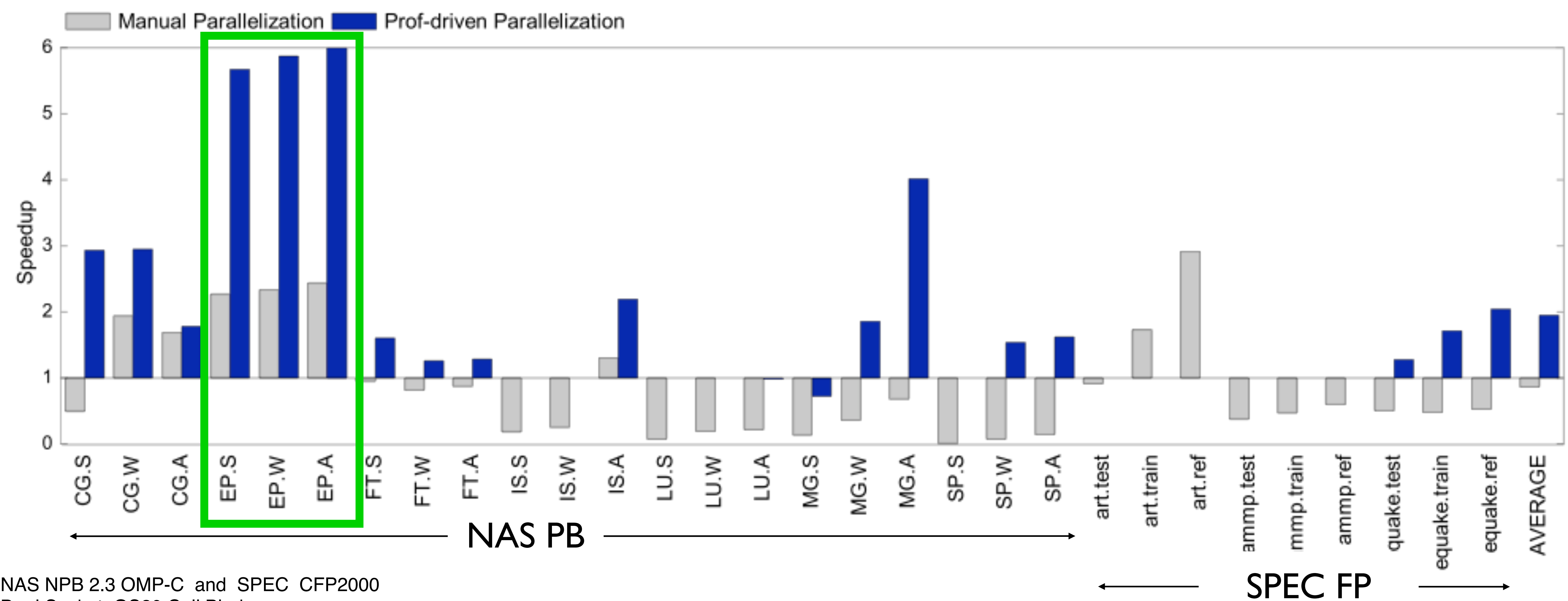
- Overhead is more obvious on small datasets



NAS NPB 2.3 OMP-C and SPEC CFP2000
Dual Socket, QS20 Cell Blade
IBM xlc ssc v0.9 O5 -qstrict -qarch=cell -qipa=partition=minute

Performance on Cell

- EP gets significant speedup
  - No synchronization
  - Not memory bound



NAS NPB 2.3 OMP-C and SPEC CFP2000
Dual Socket, QS20 Cell Blade
IBM xlc ssc v0.9 O5 -qstrict -qarch=cell -qipa=partition=minute

PIPELINES:
CODE GENERATION

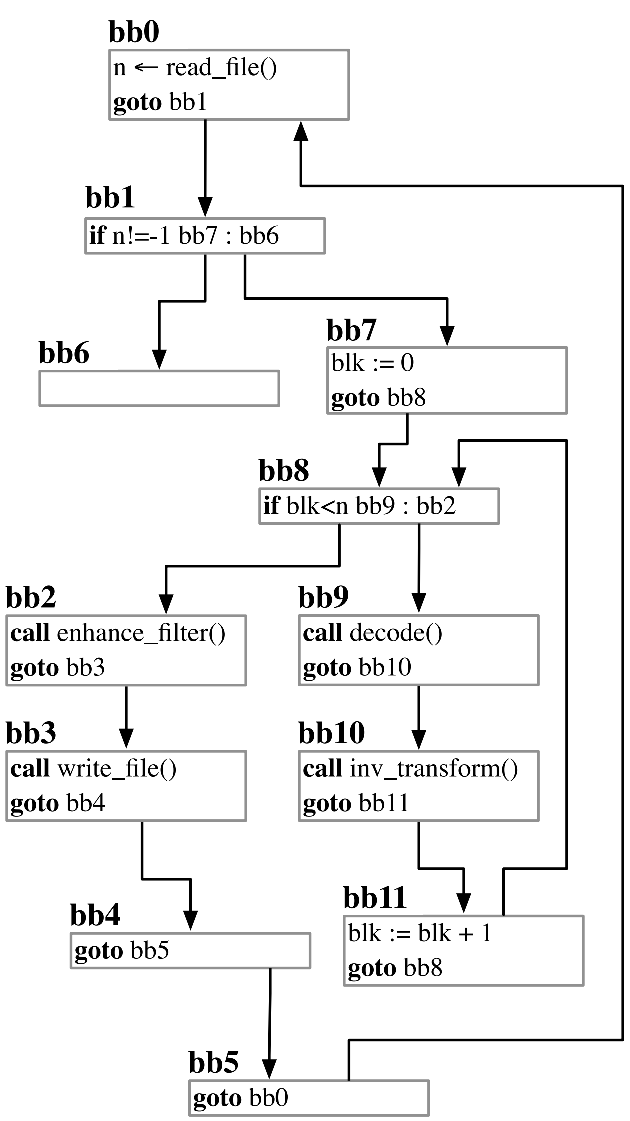
Parallel code generation

```
5%    while((n = read_file(inf, data)) != EOF) {
        for (blk=0; blk<n; blk++) {
20%       coef[blk] = decode(data, blk);
50%       raw_data[blk] = inv_transform(coef, blk);
        }
20%     out_data = enhance_filter(raw_data);
5%      write_file(outf, out_data);
      } /* while */
```

Parallel code generation

```
5%    while((n = read_file(inf, data)) != EOF) {
        for (blk=0; blk<n; blk++) {
20%       coef[blk] = decode(data, blk);
50%       raw_data[blk] = inv_transform(coef, blk);
        }
20%     out_data = enhance_filter(raw_data);
5%      write_file(outf, out_data);
      } /* while */
```

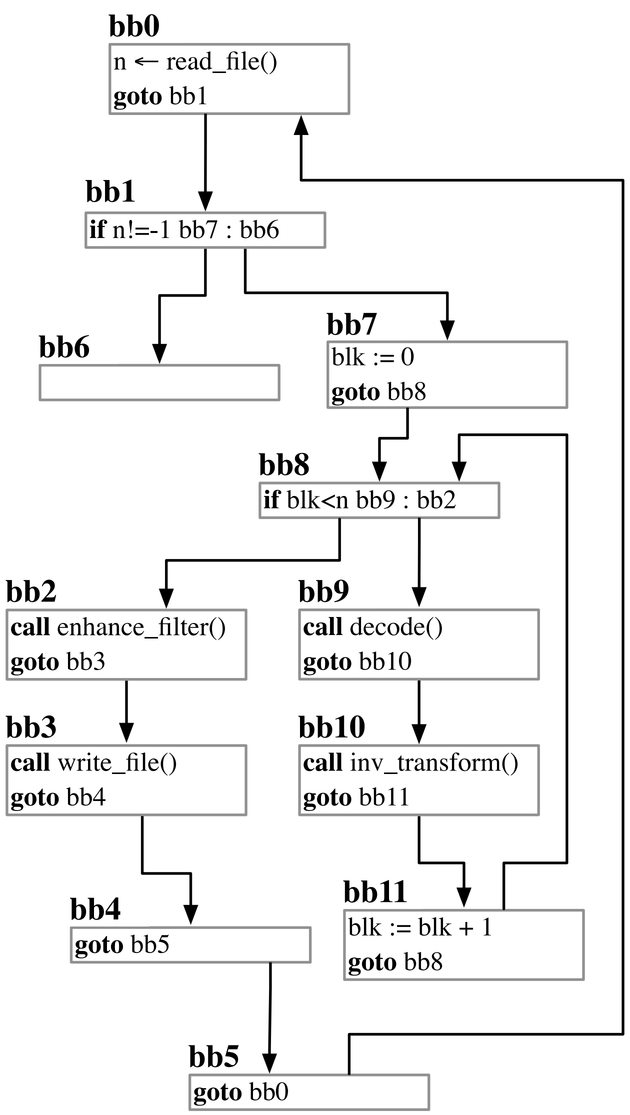
Sequential CFG



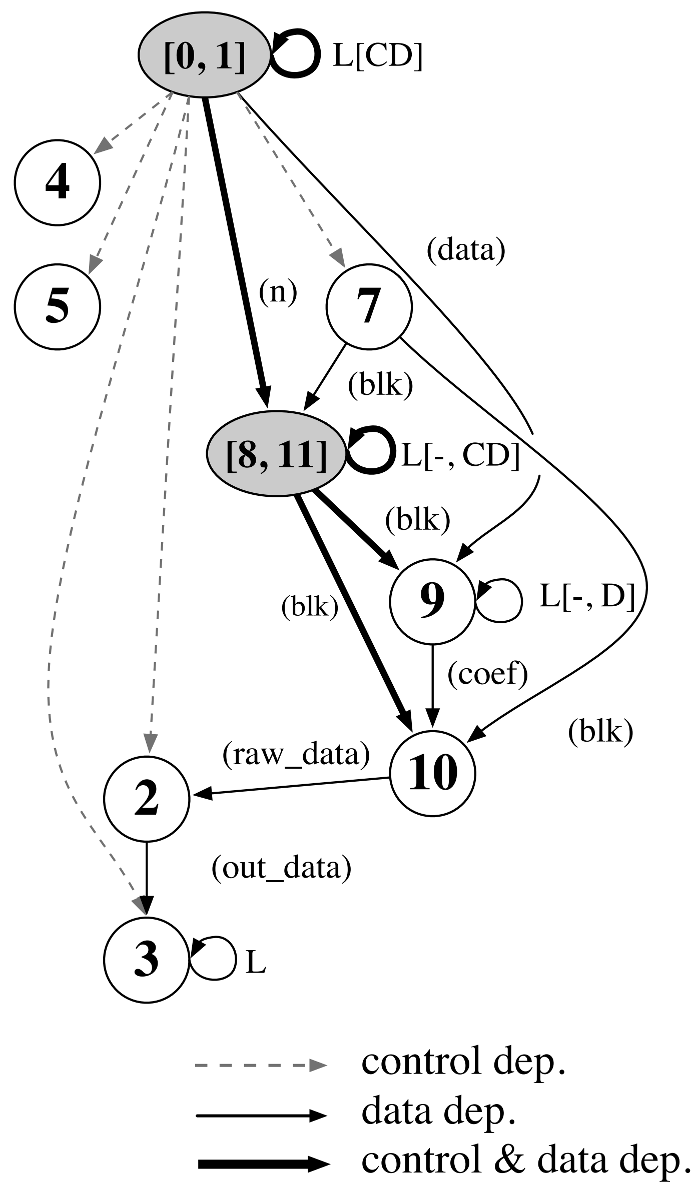
Parallel code generation

```
5%   while((n = read_file(inf, data)) != EOF) {
       for (blk=0; blk<n; blk++) {
20%      coef[blk] = decode(data, blk);
50%      raw_data[blk] = inv_transform(coef, blk);
       }
20%    out_data = enhance_filter(raw_data);
5%     write_file(outf, out_data);
     } /* while */
```

Sequential CFG



PDG



Parallel code generation

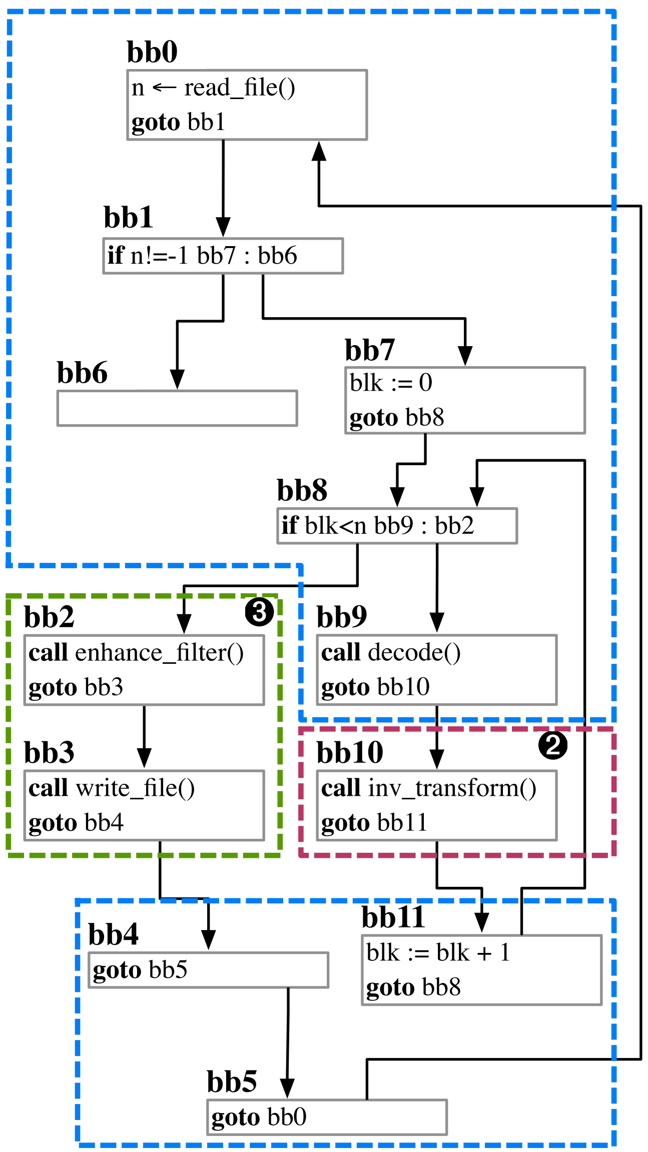
```
5%   while((n = read_file(inf, data)) != EOF) {
       for (blk=0; blk<n; blk++) {
20%      coef[blk] = decode(data, blk);
50%      raw_data[blk] = inv_transform(coef, blk);
       }
20%    out_data = enhance_filter(raw_data);
5%     write_file(outf, out_data);
     } /* while */
```

Stage 1

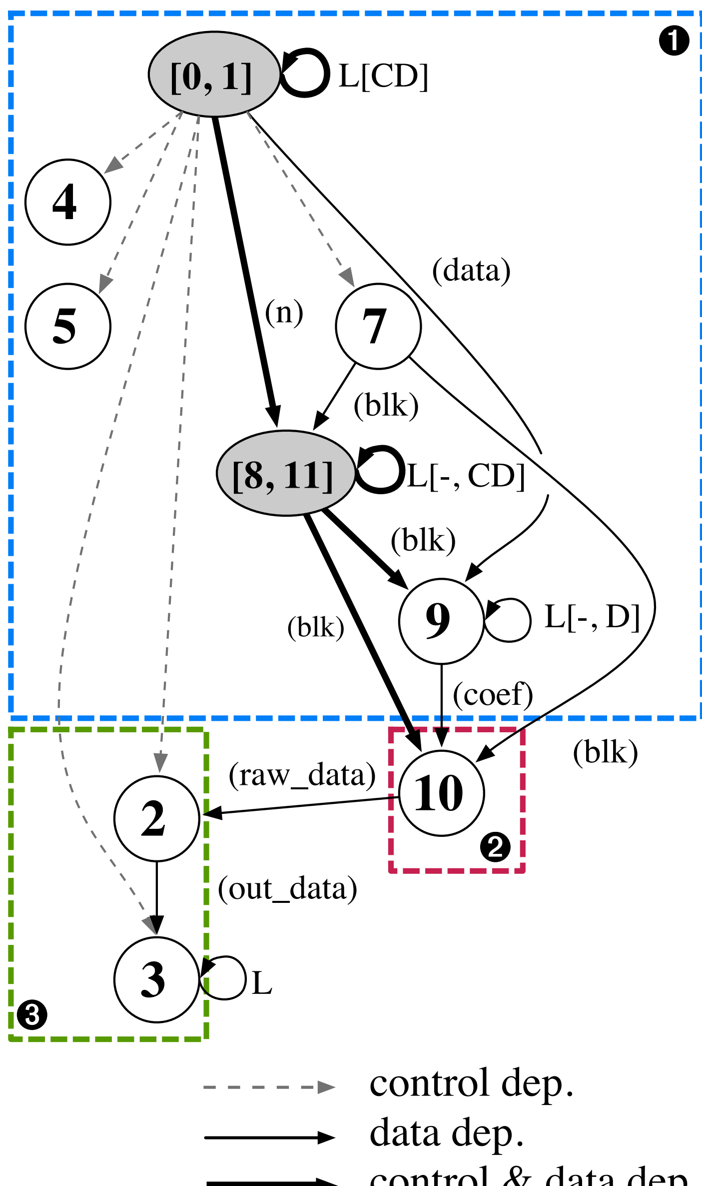
Stage 2

Stage 3

Sequential CFG



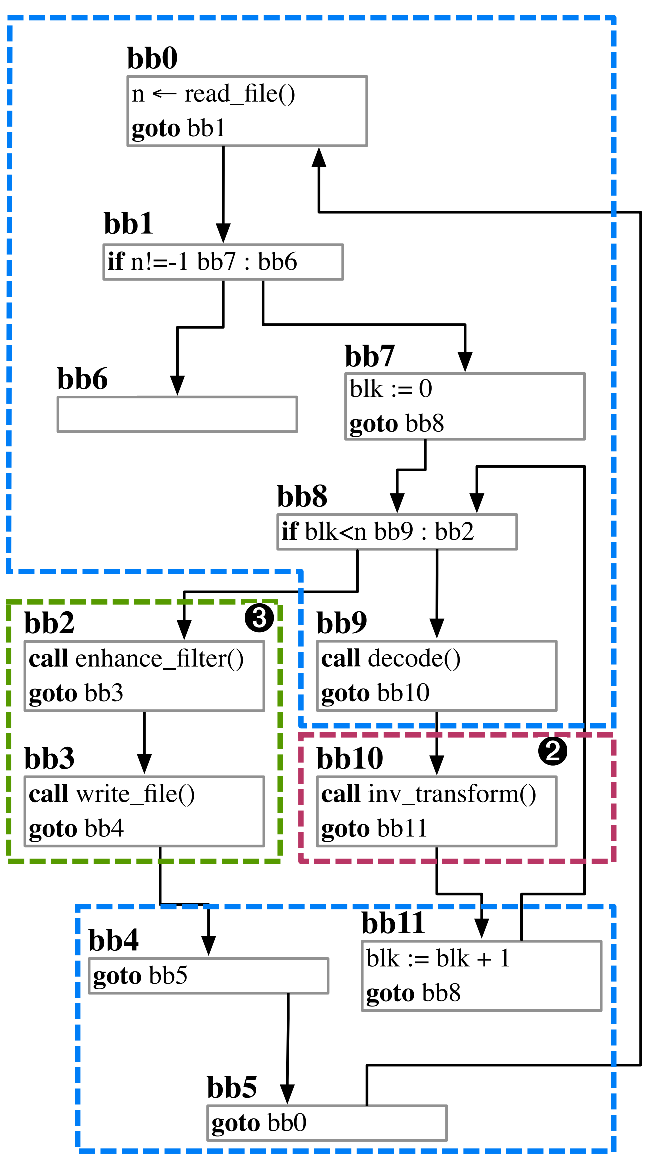
PDG Partitioning



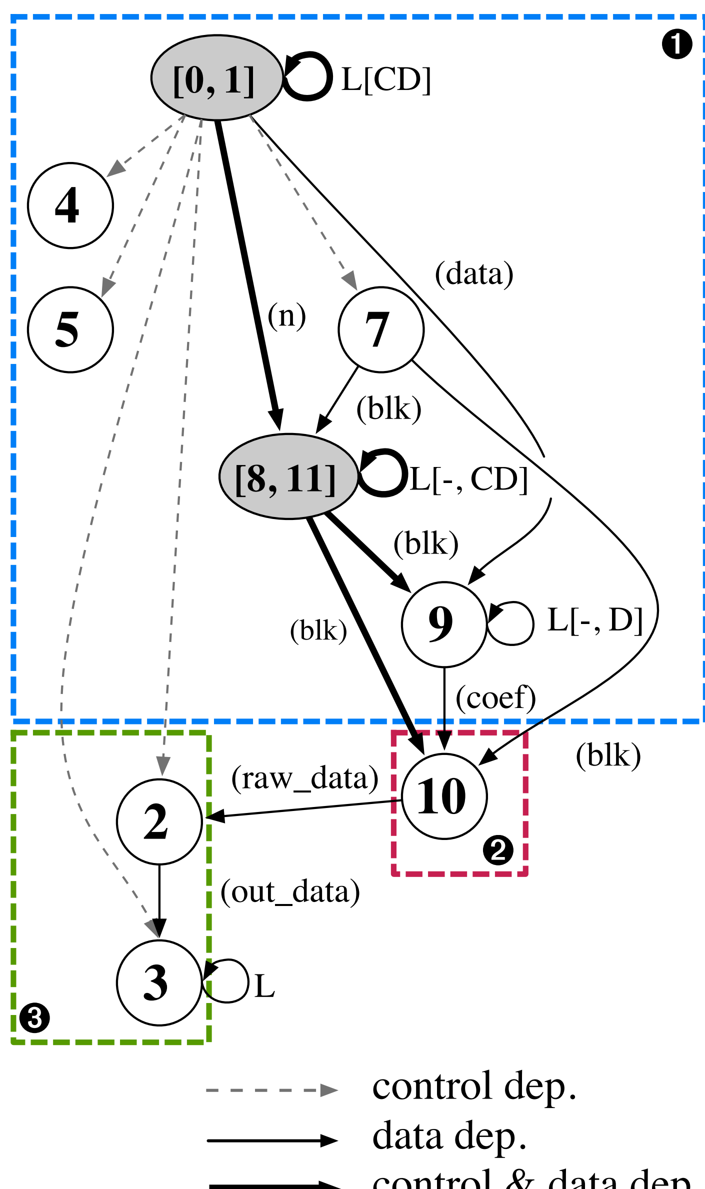
Parallel code generation

```
5%    while((n = read_file(inf, data)) != EOF) {           Stage 1
        for (blk=0; blk<n; blk++) {
20%       coef[blk] = decode(data, blk);
50%       raw_data[blk] = inv_transform(coef, blk);        Stage 2
        }
20%     out_data = enhance_filter(raw_data);               Stage 3
5%      write_file(outf, out_data);
      } /* while */
```

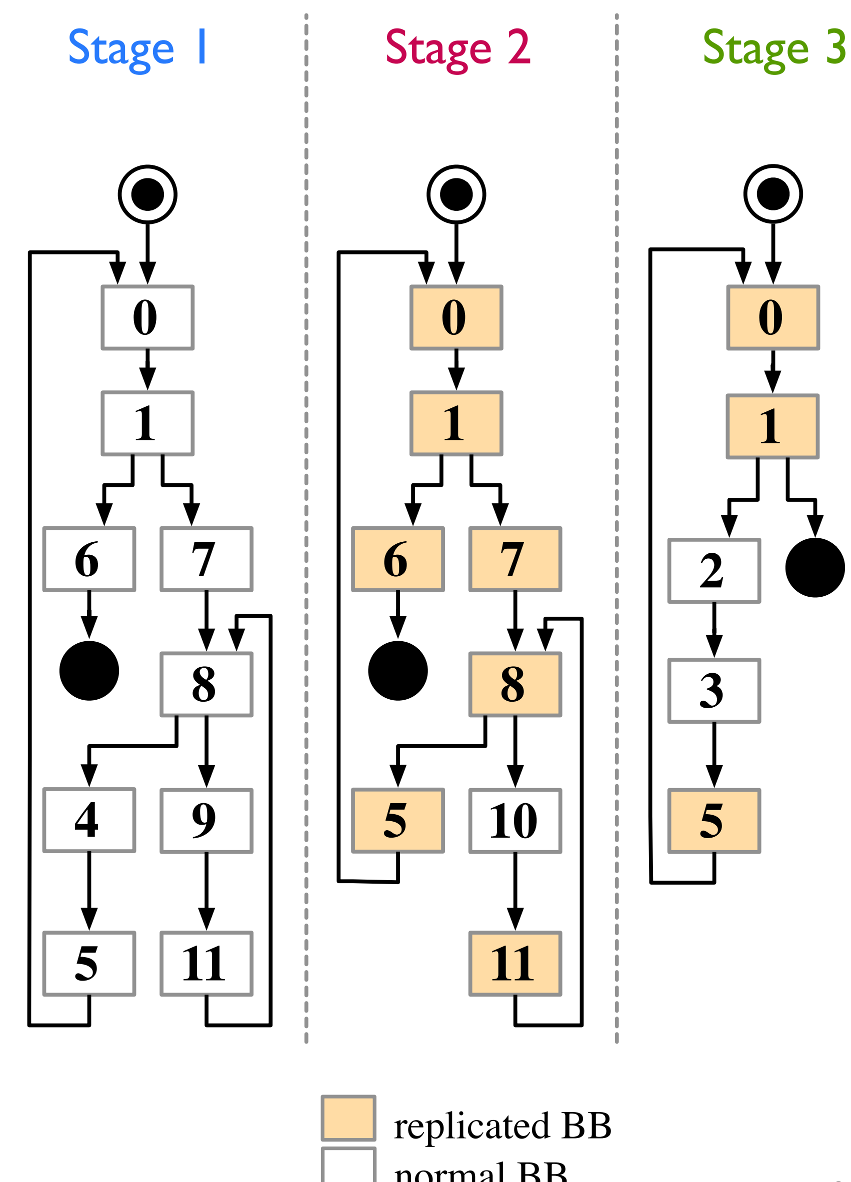
Sequential CFG



PDG Partitioning



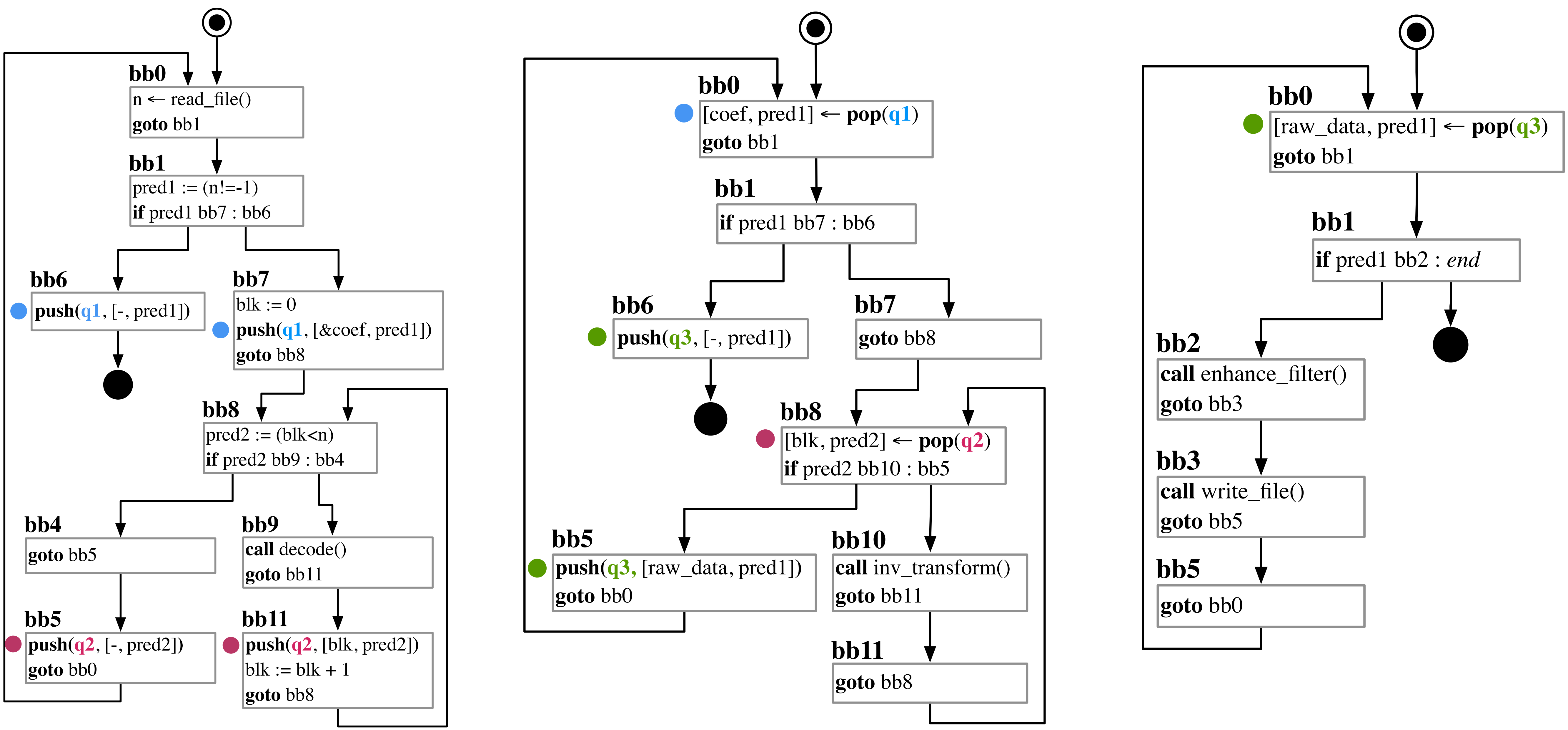
Sequentialization



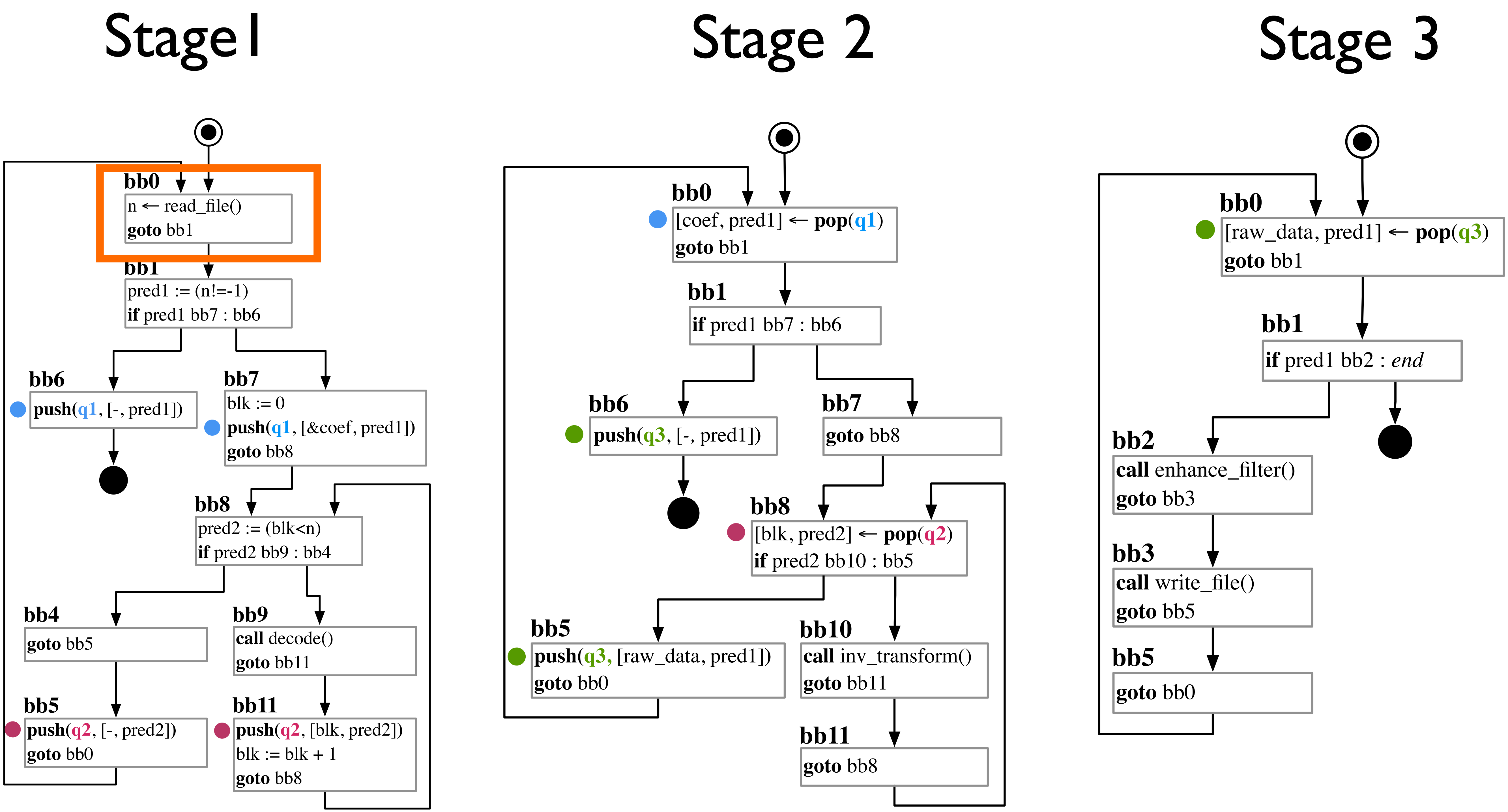
PIPELINES: COMMUNICATION

Communication

Stage 1 Stage 2 Stage 3

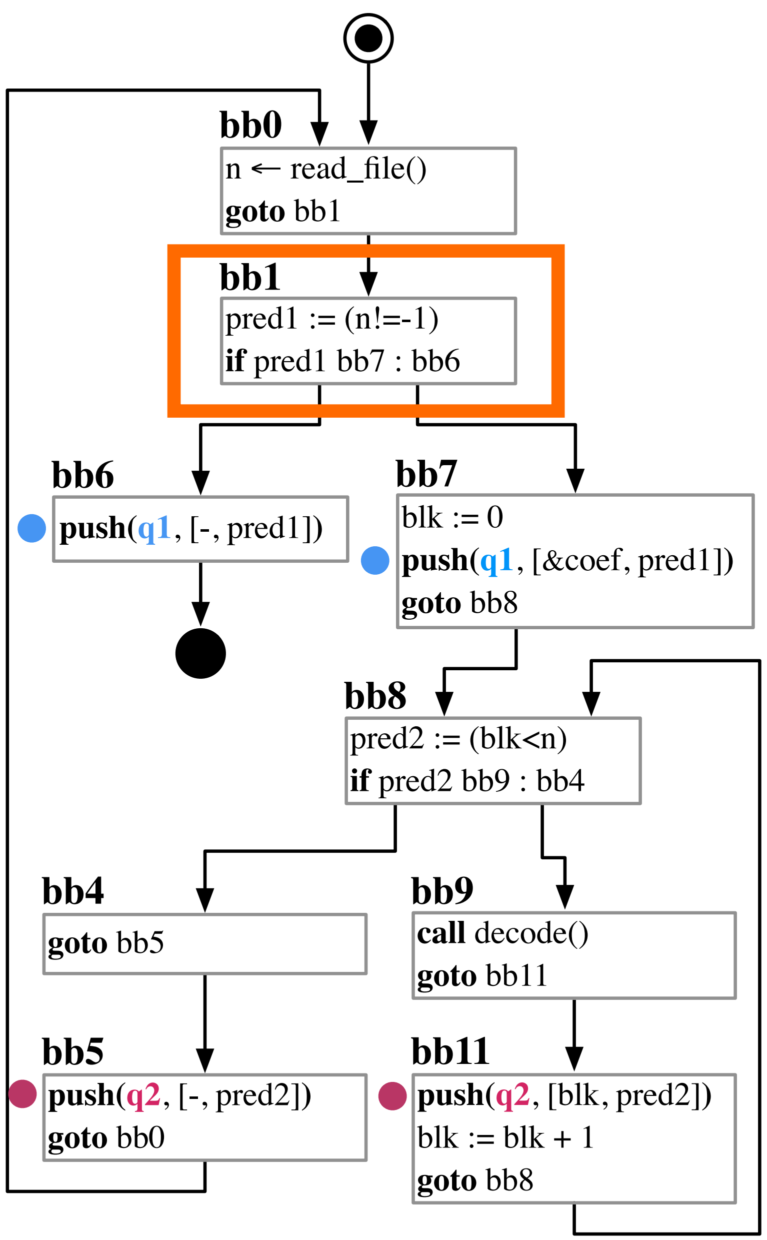


Communication

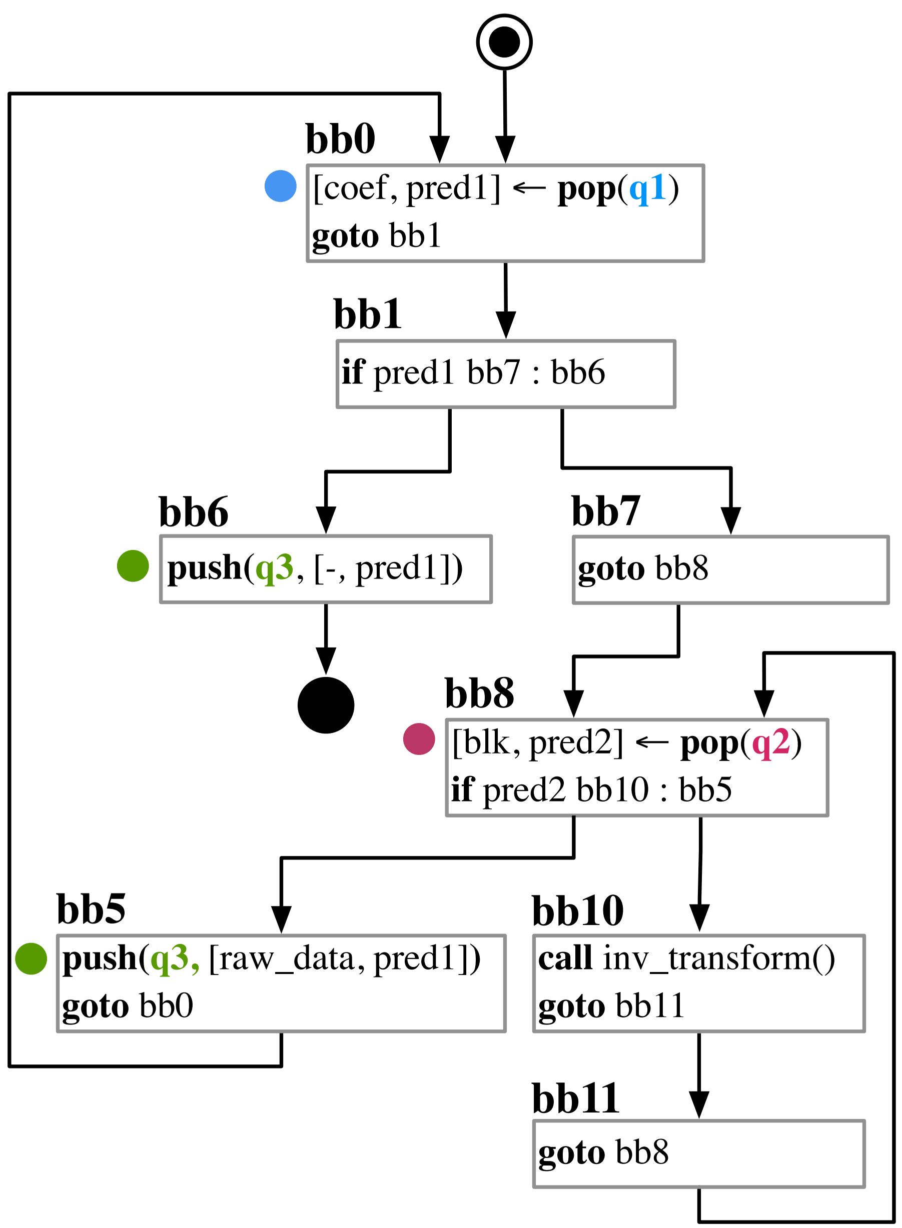


Communication

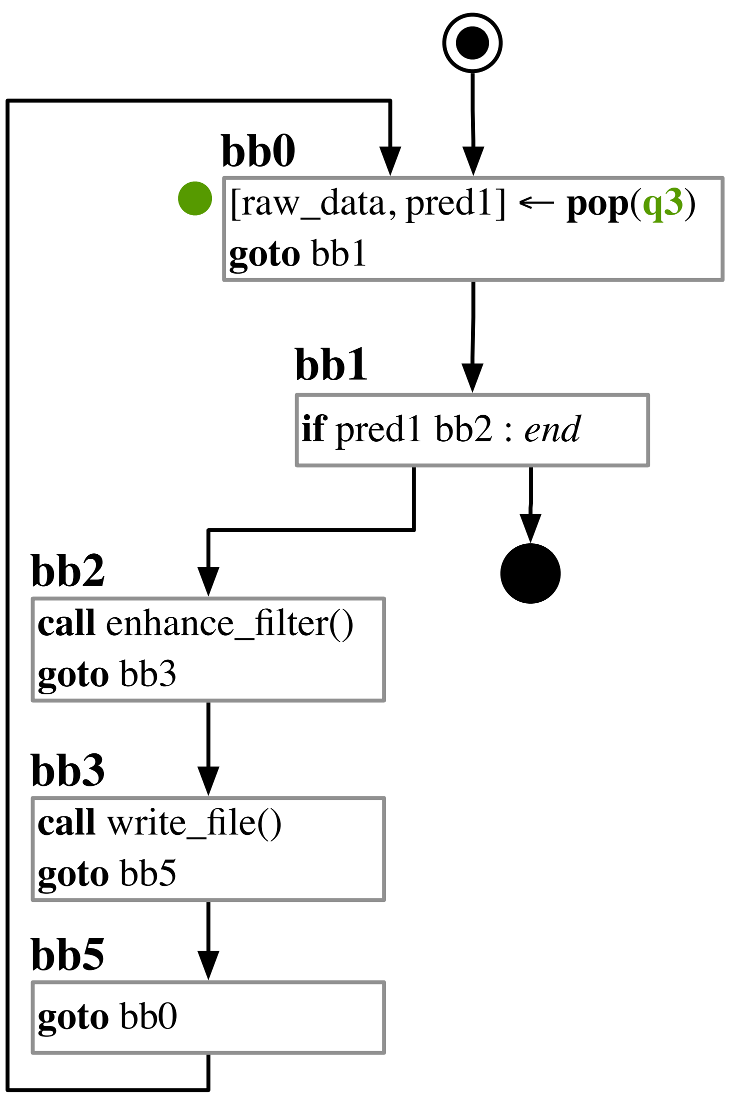
Stage 1



Stage 2

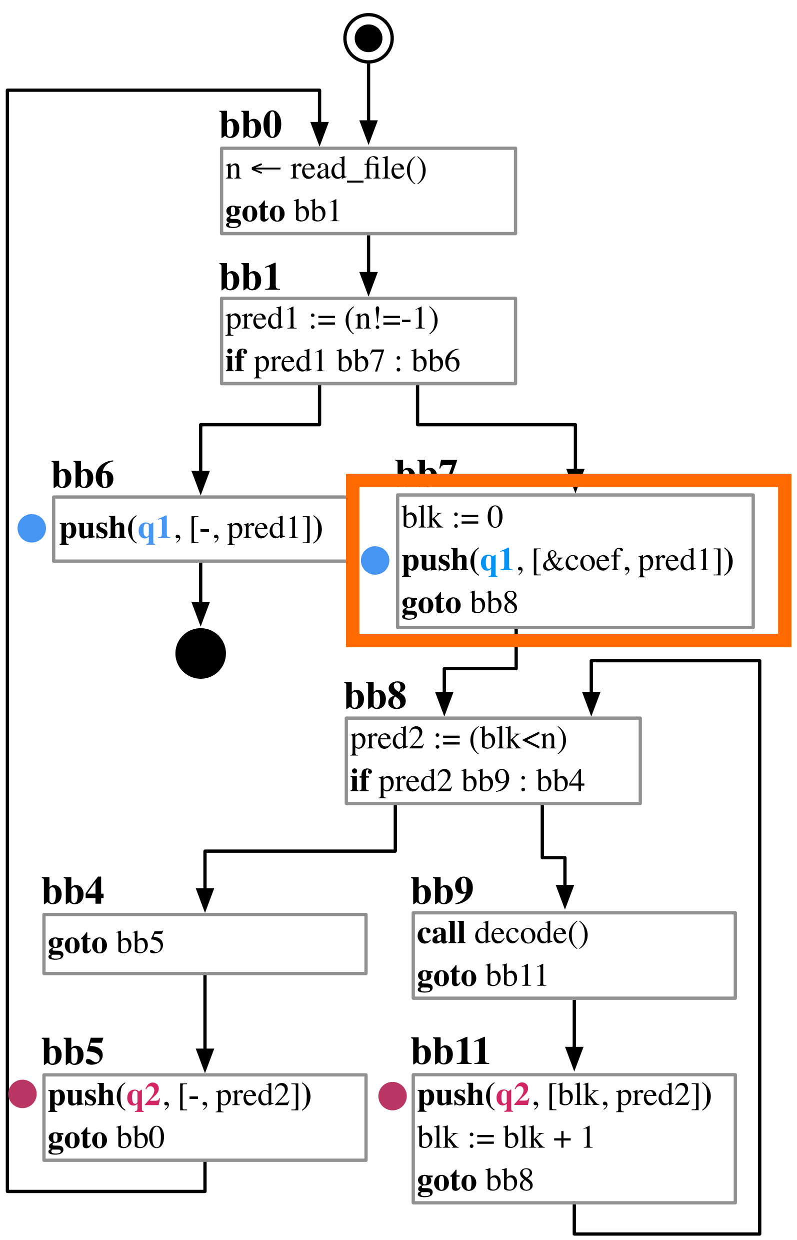


Stage 3

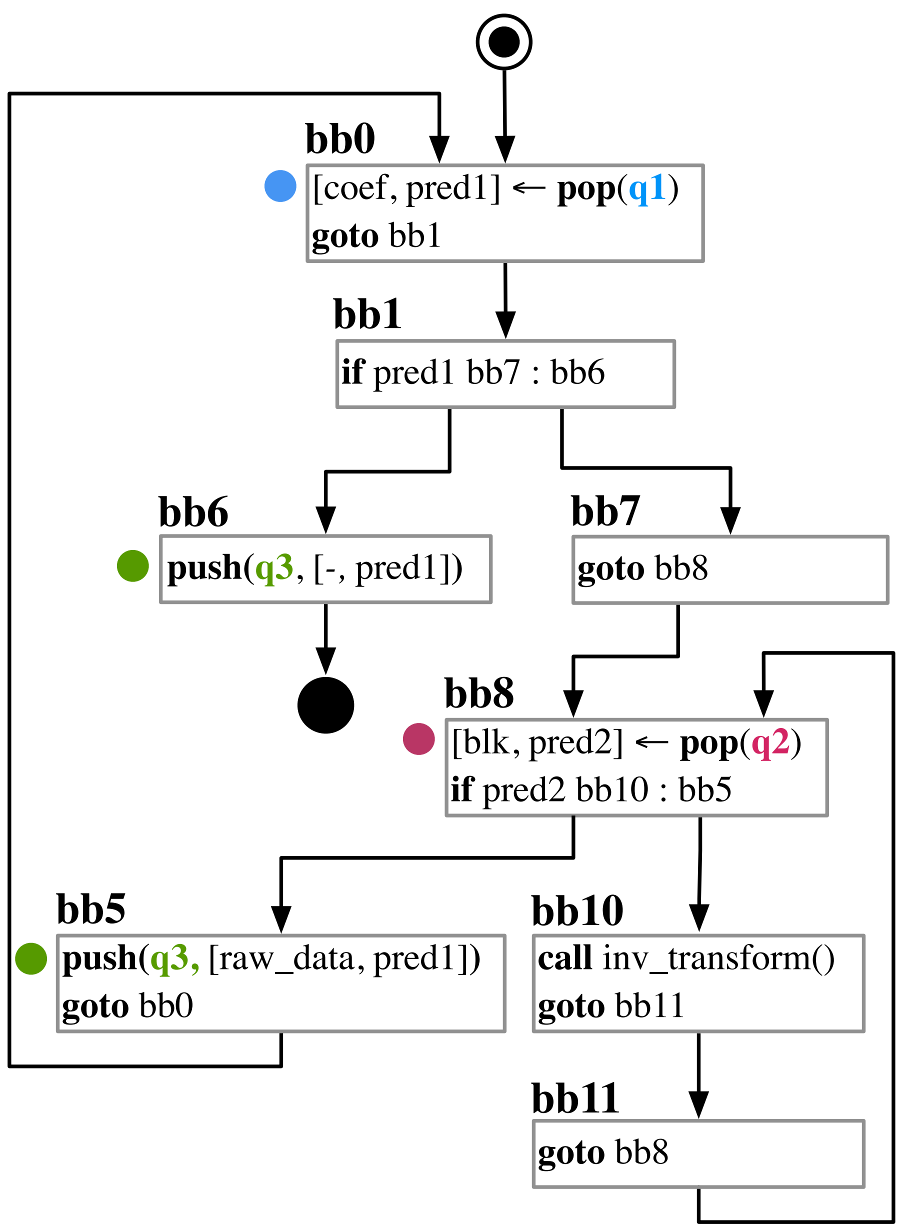


Communication

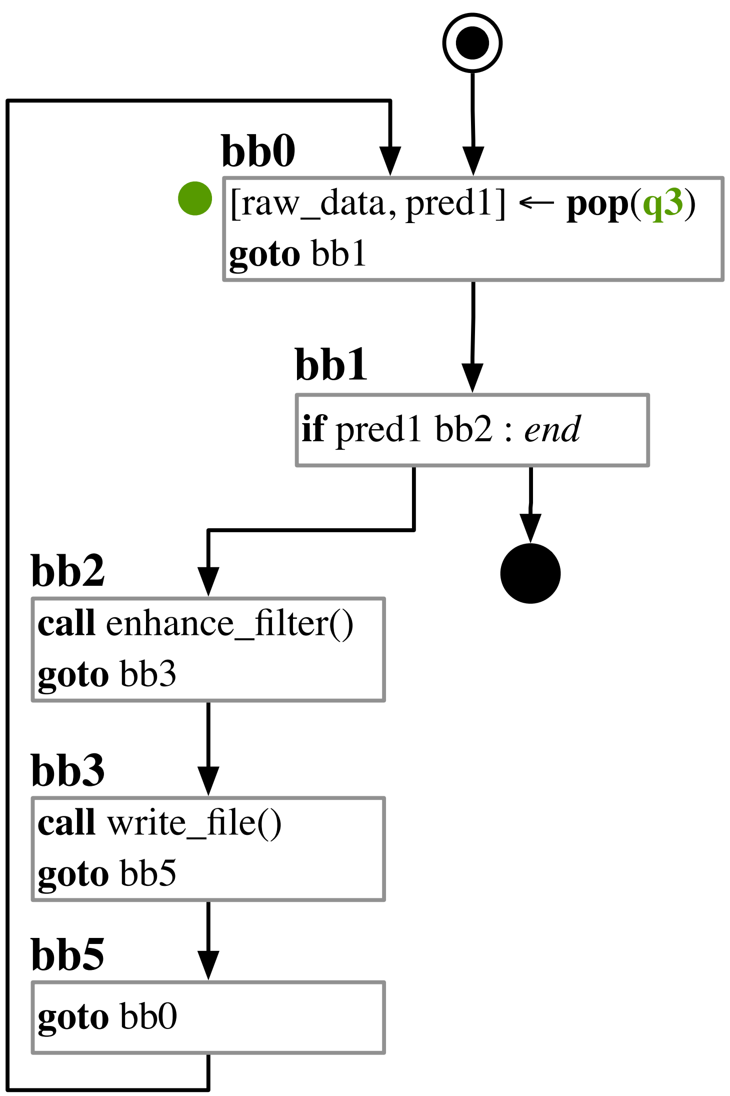
Stage 1



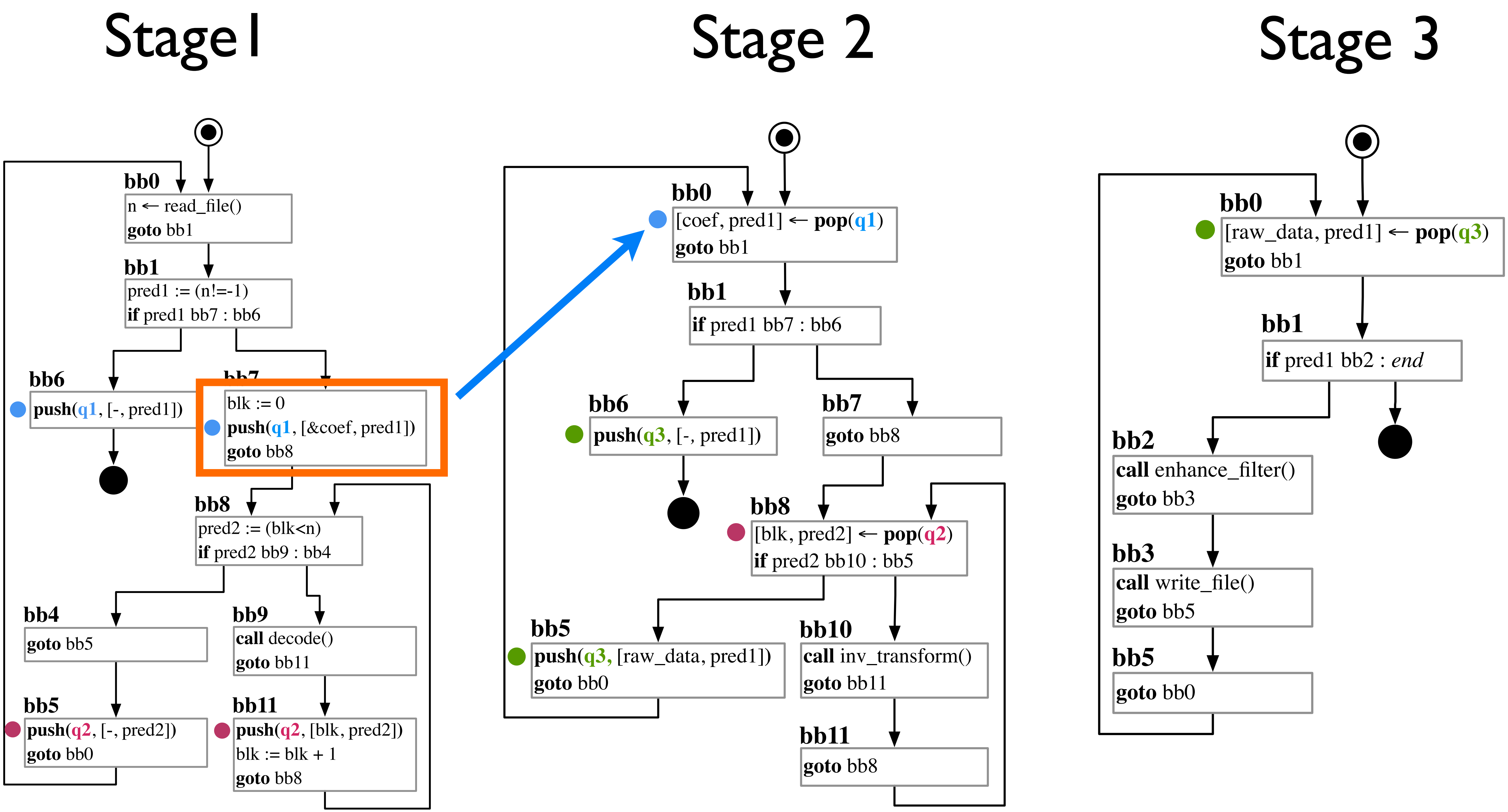
Stage 2



Stage 3

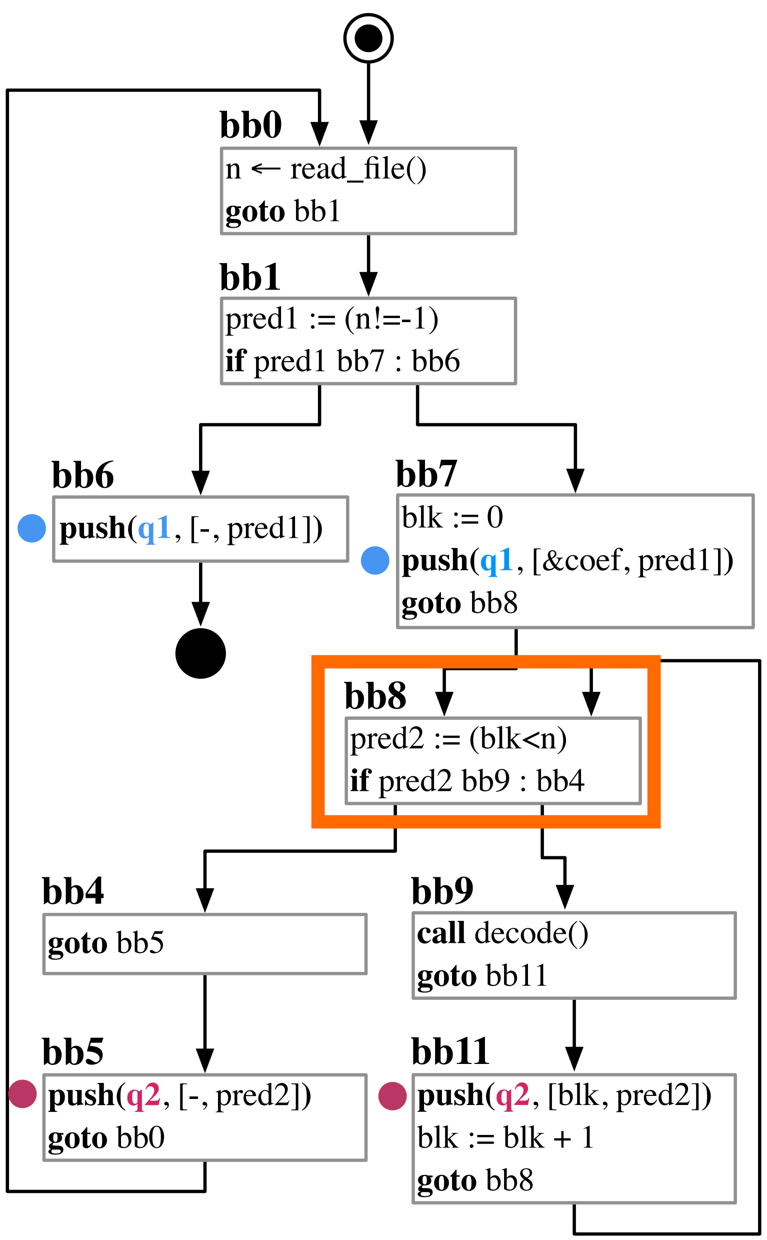


Communication

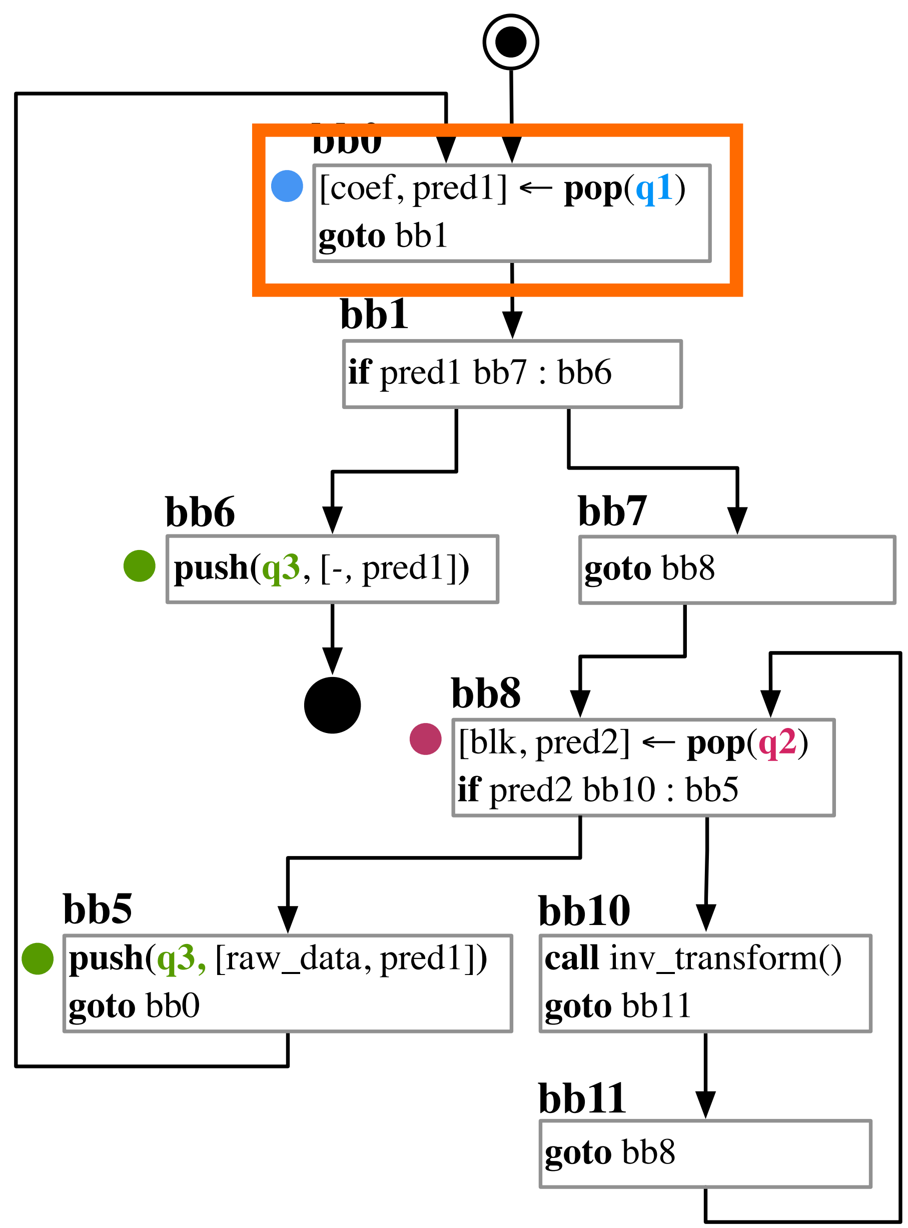


Communication

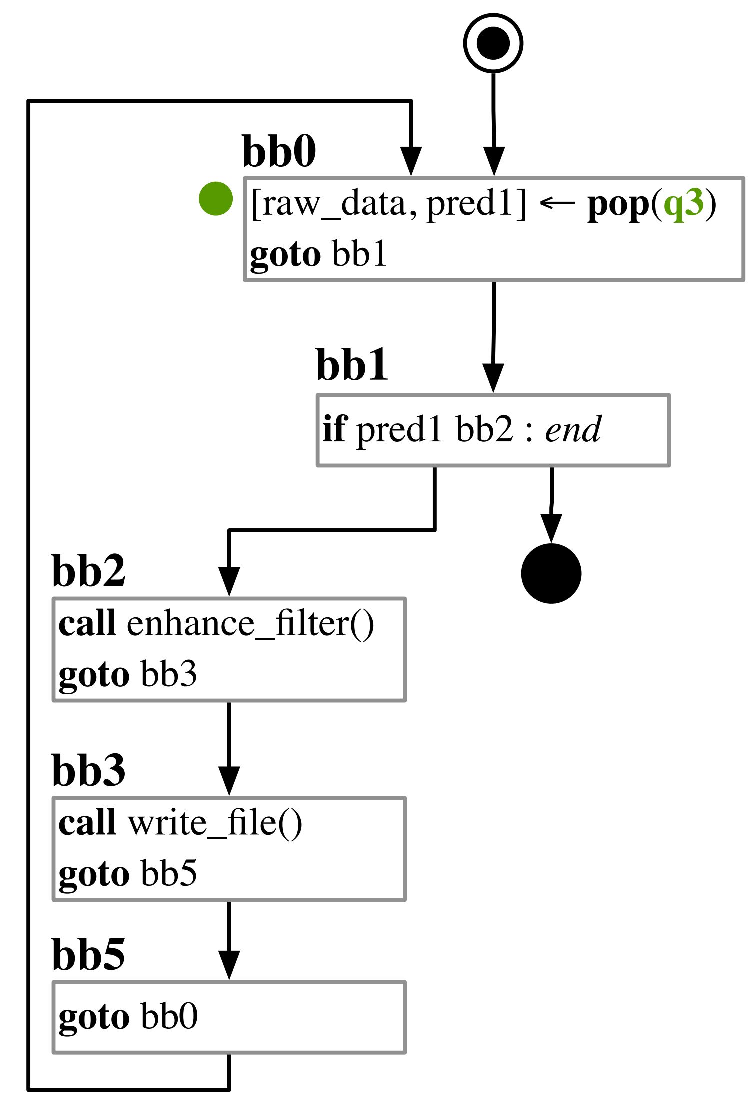
Stage 1



Stage 2

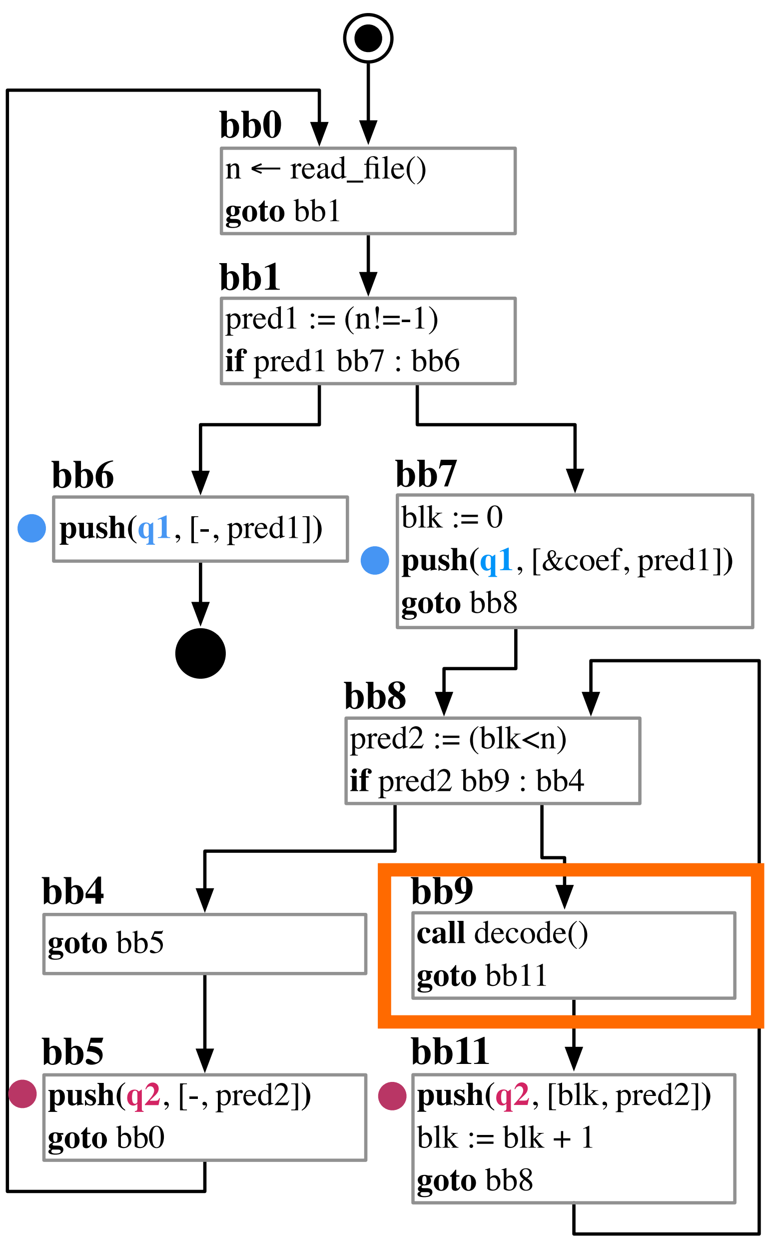


Stage 3

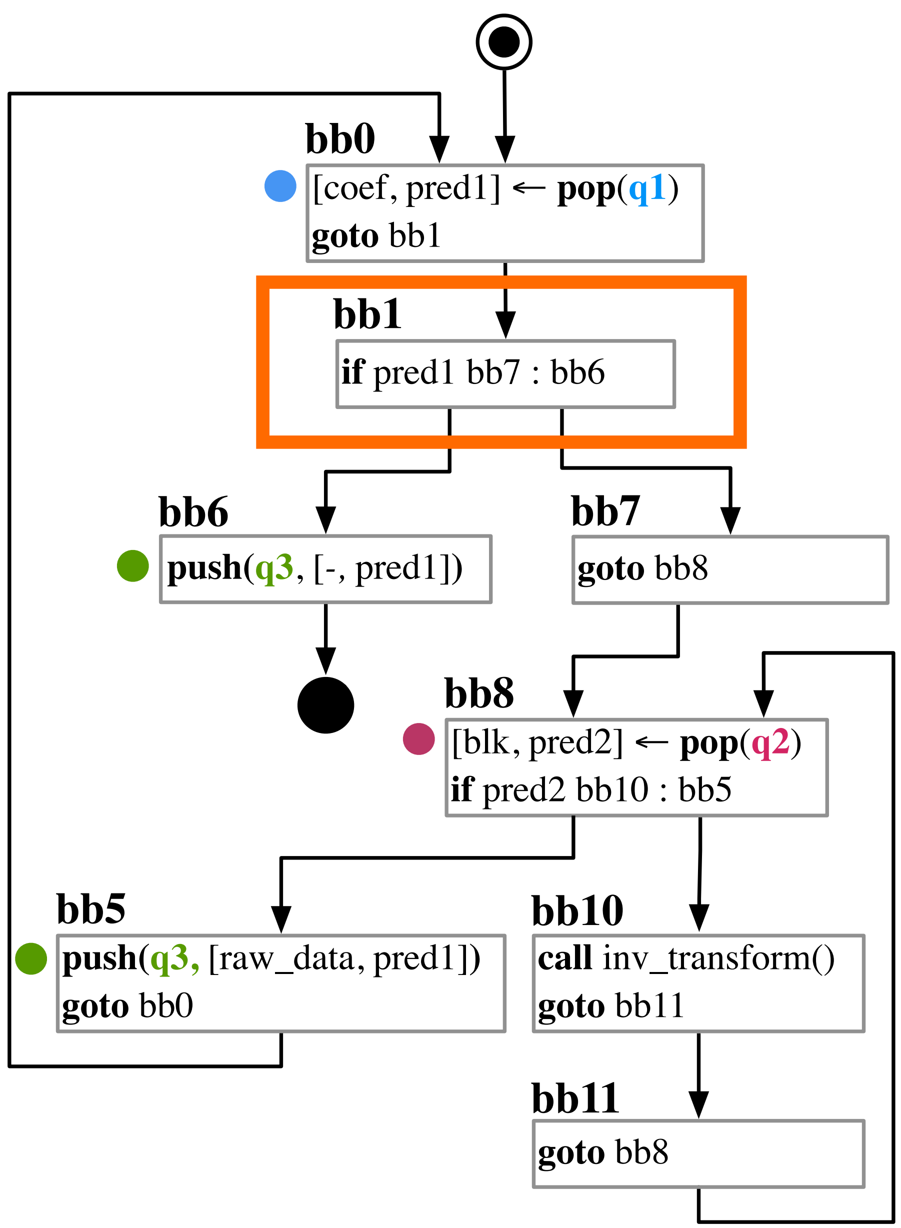


Communication

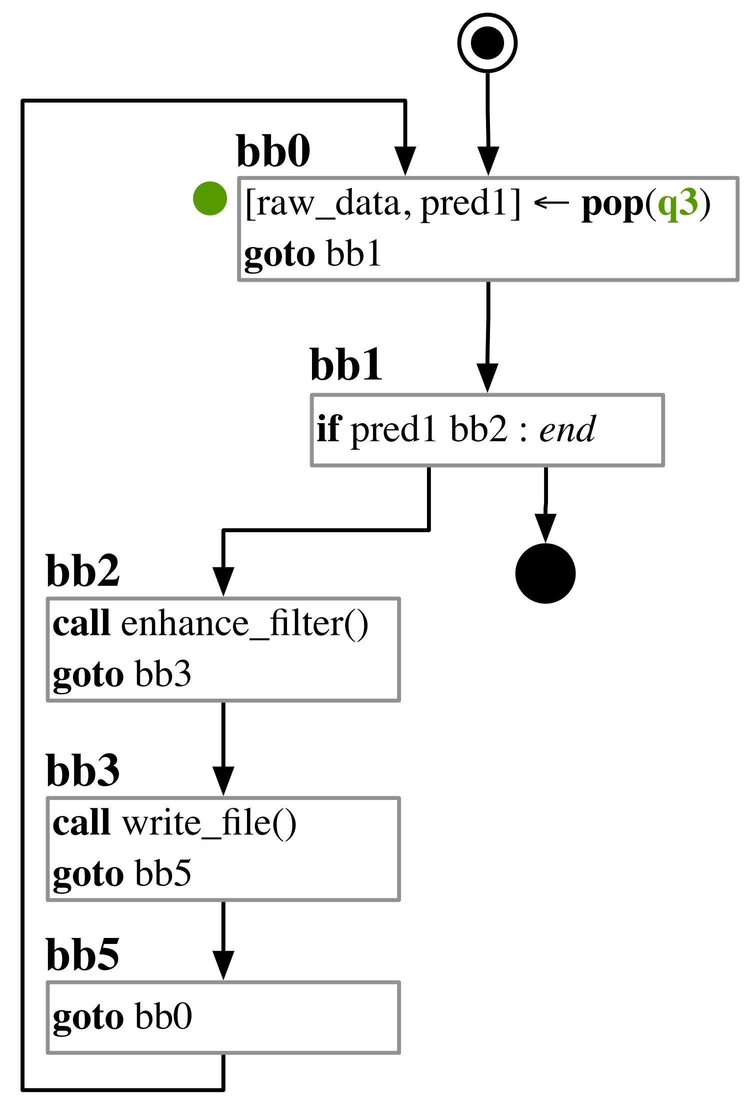
Stage 1



Stage 2

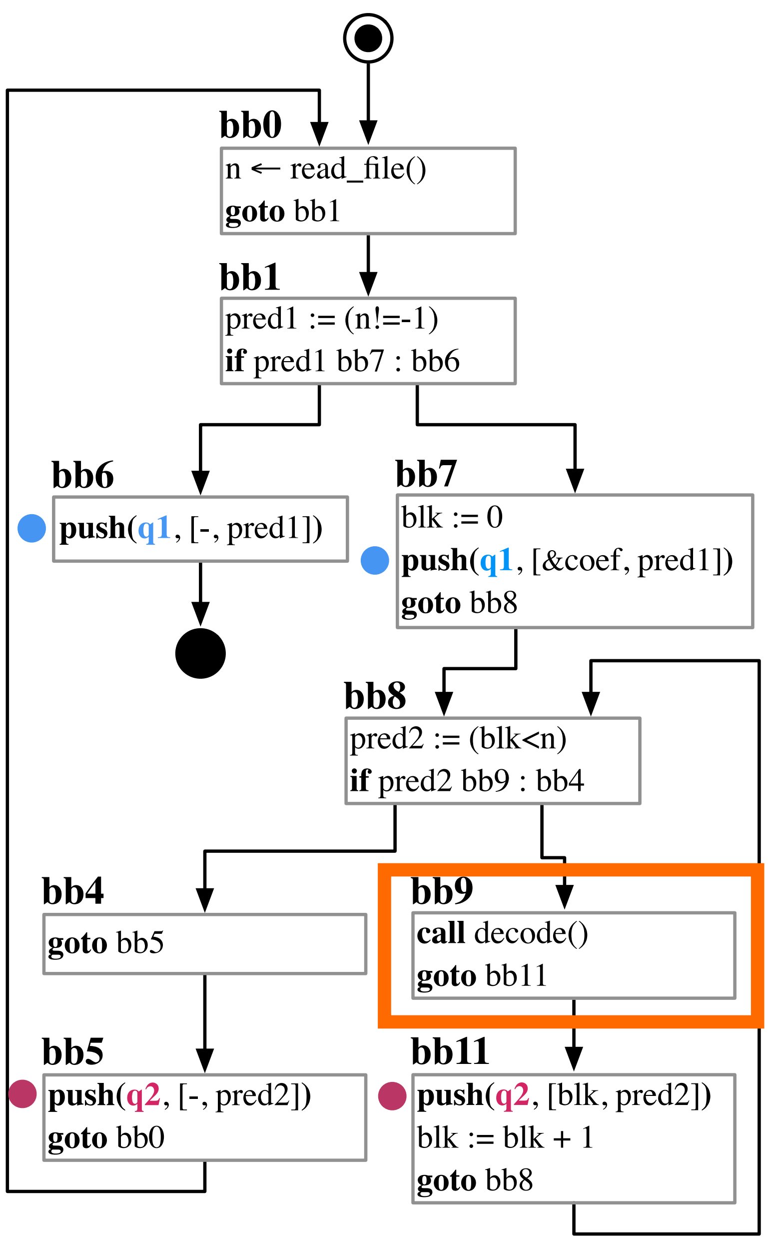


Stage 3

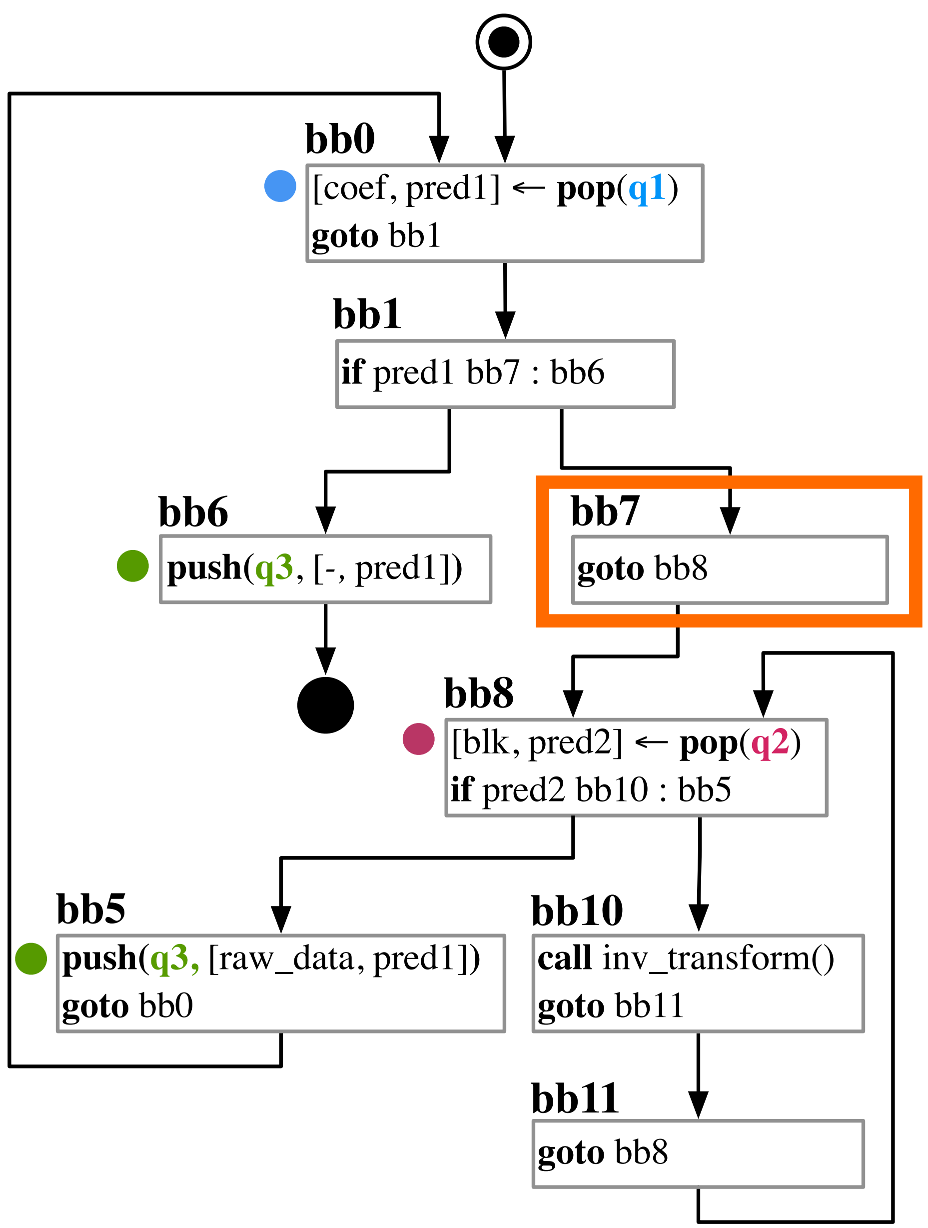


Communication

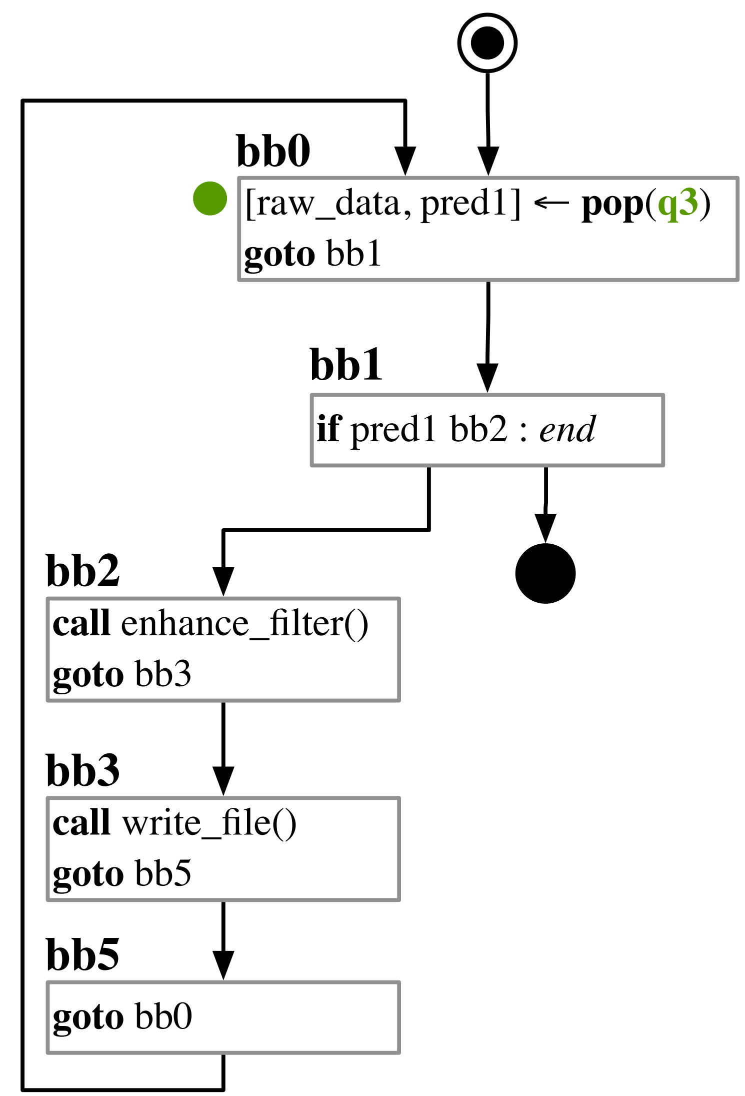
Stage 1



Stage 2

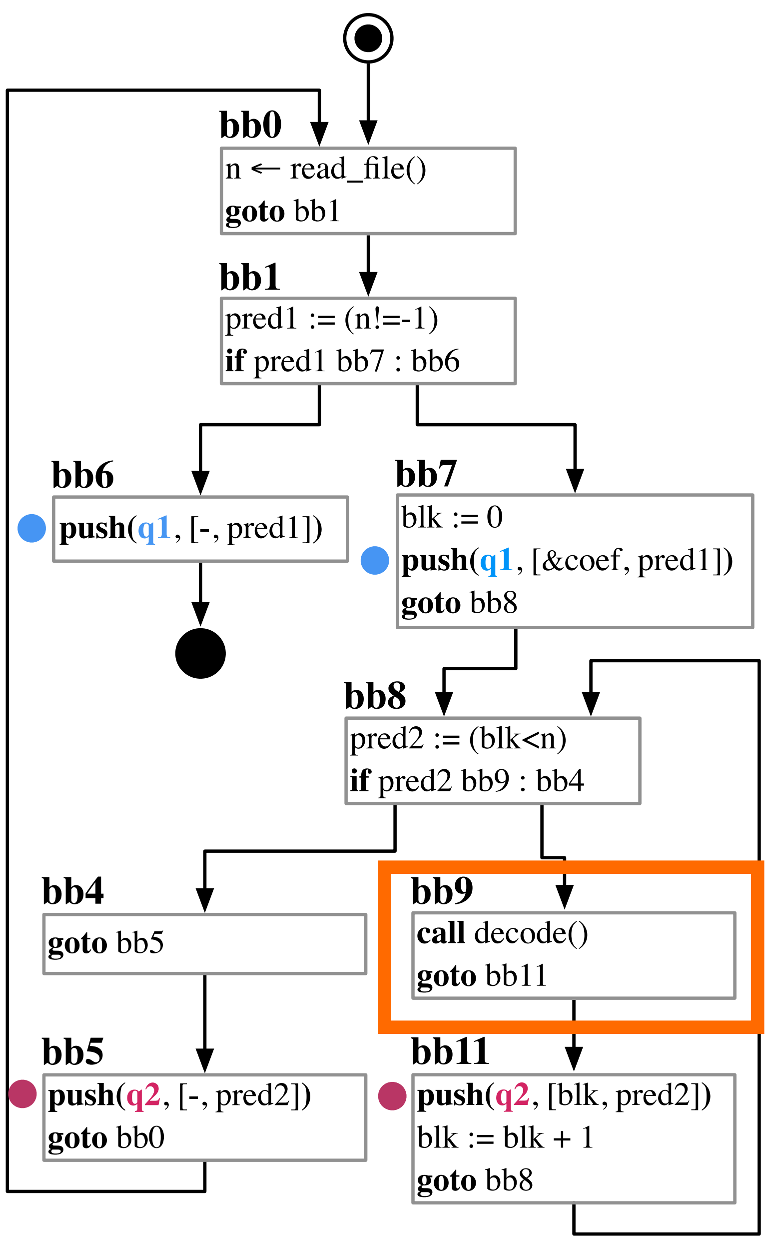


Stage 3

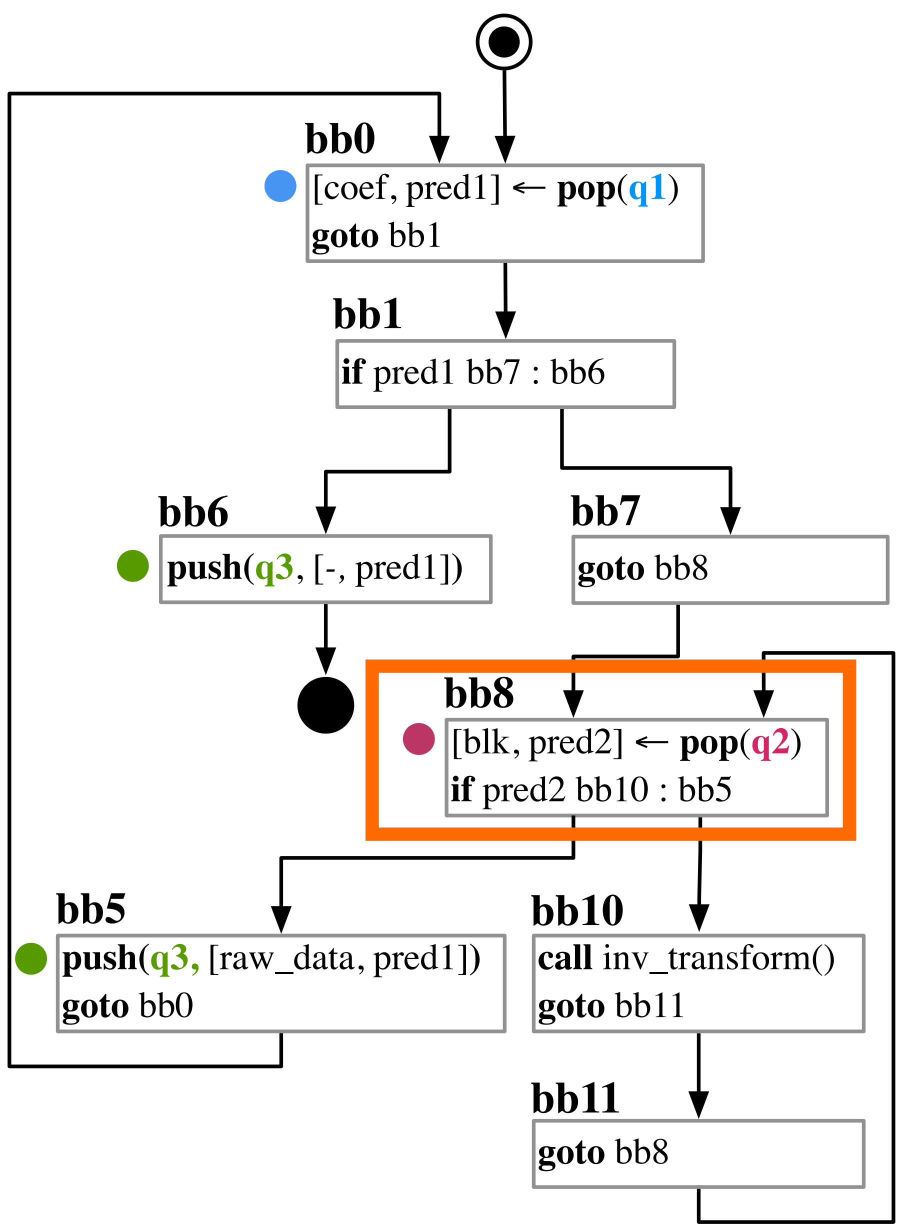


Communication

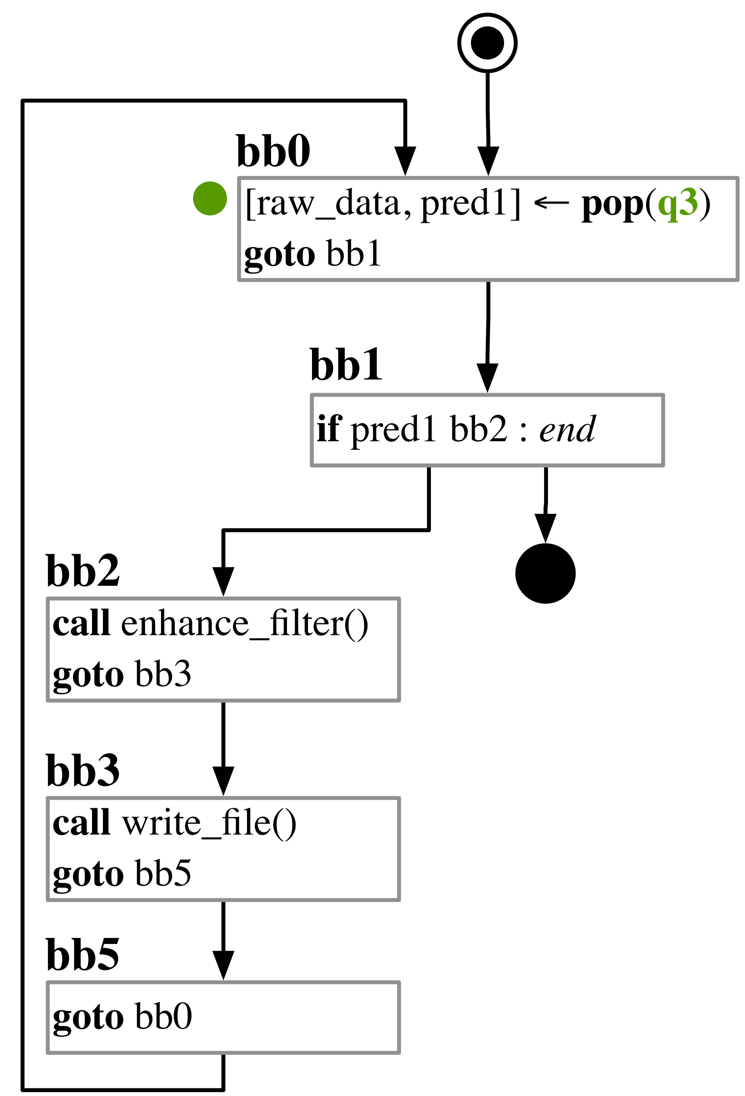
Stage 1



Stage 2

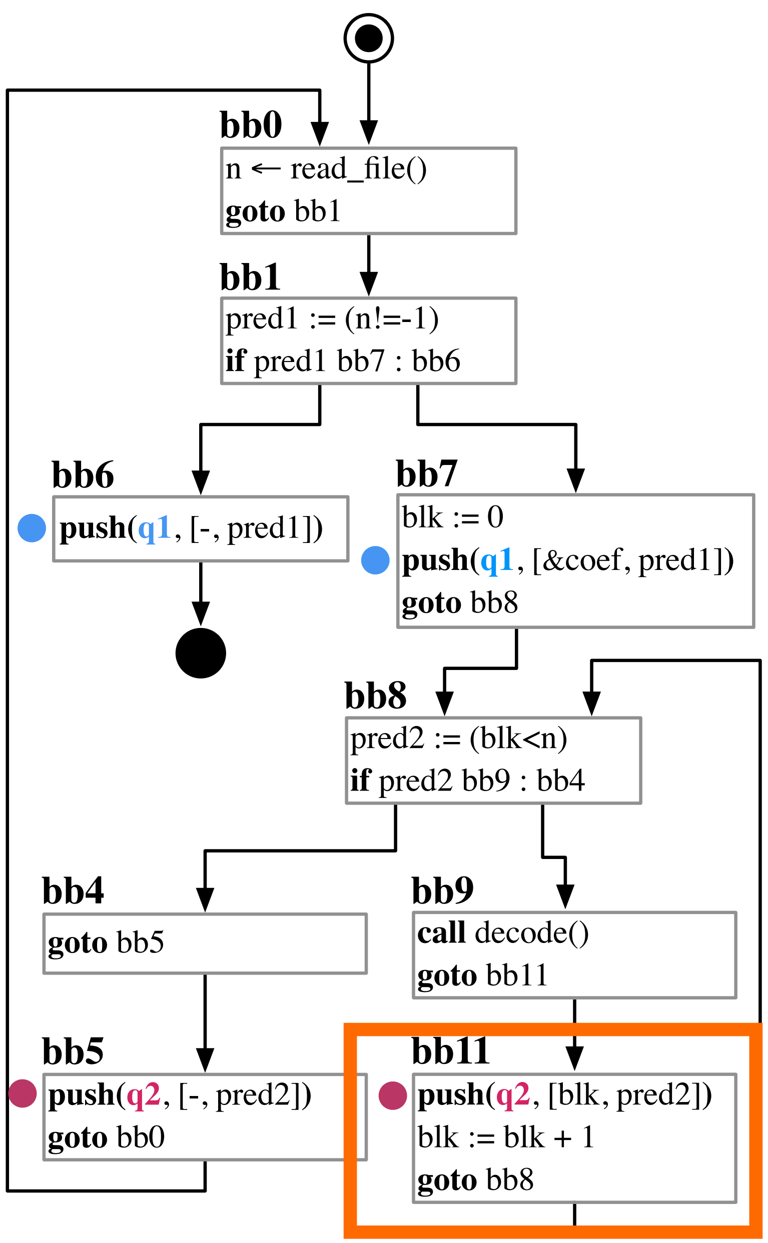


Stage 3

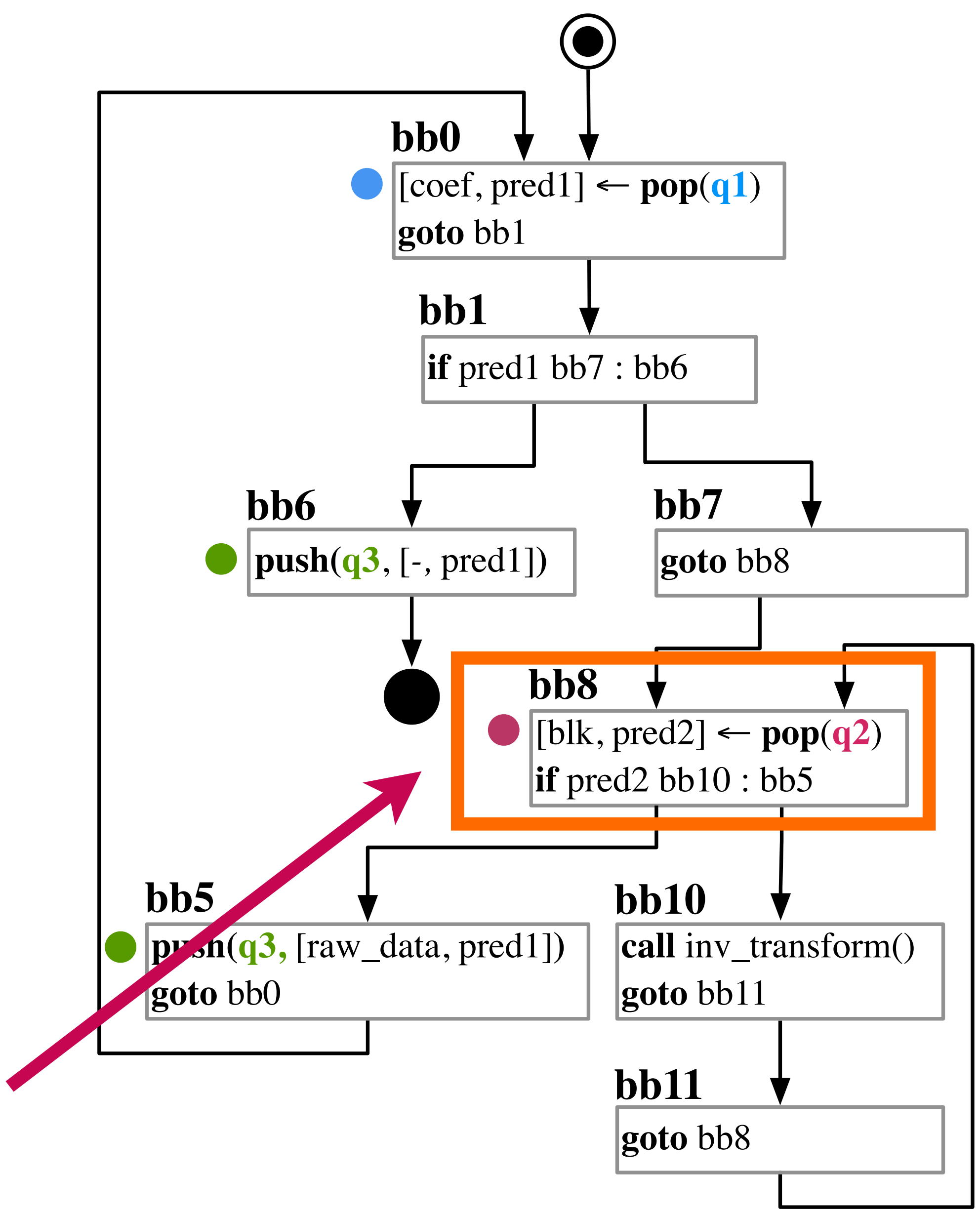


Communication

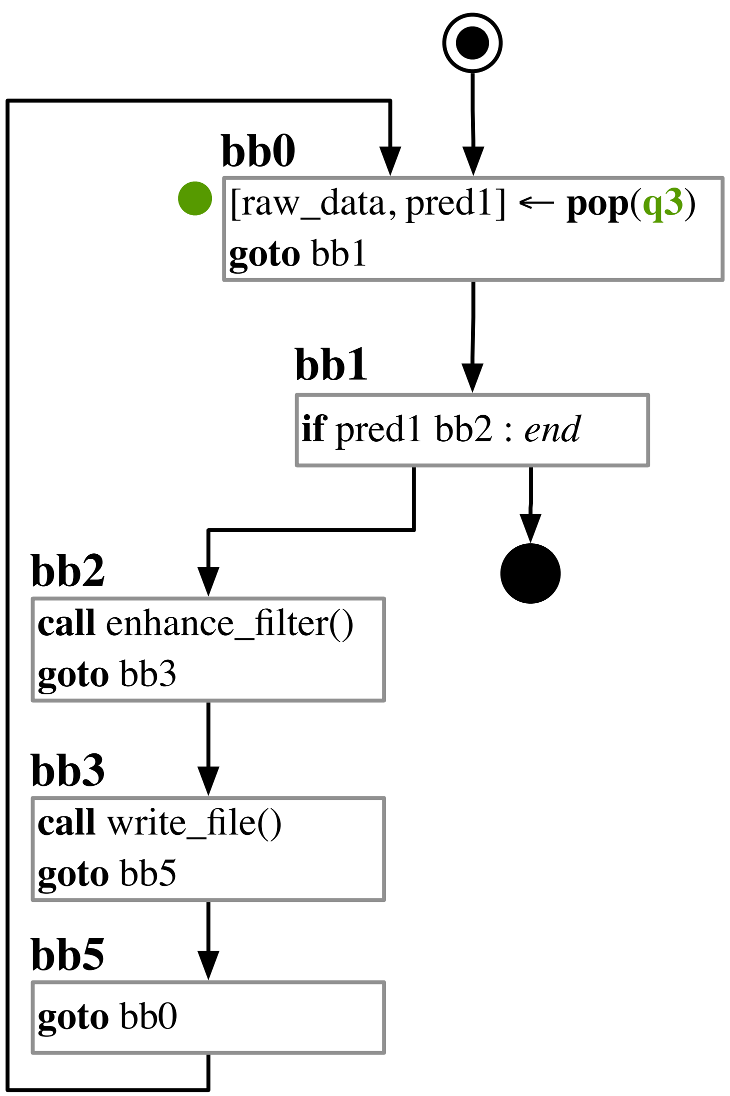
Stage 1



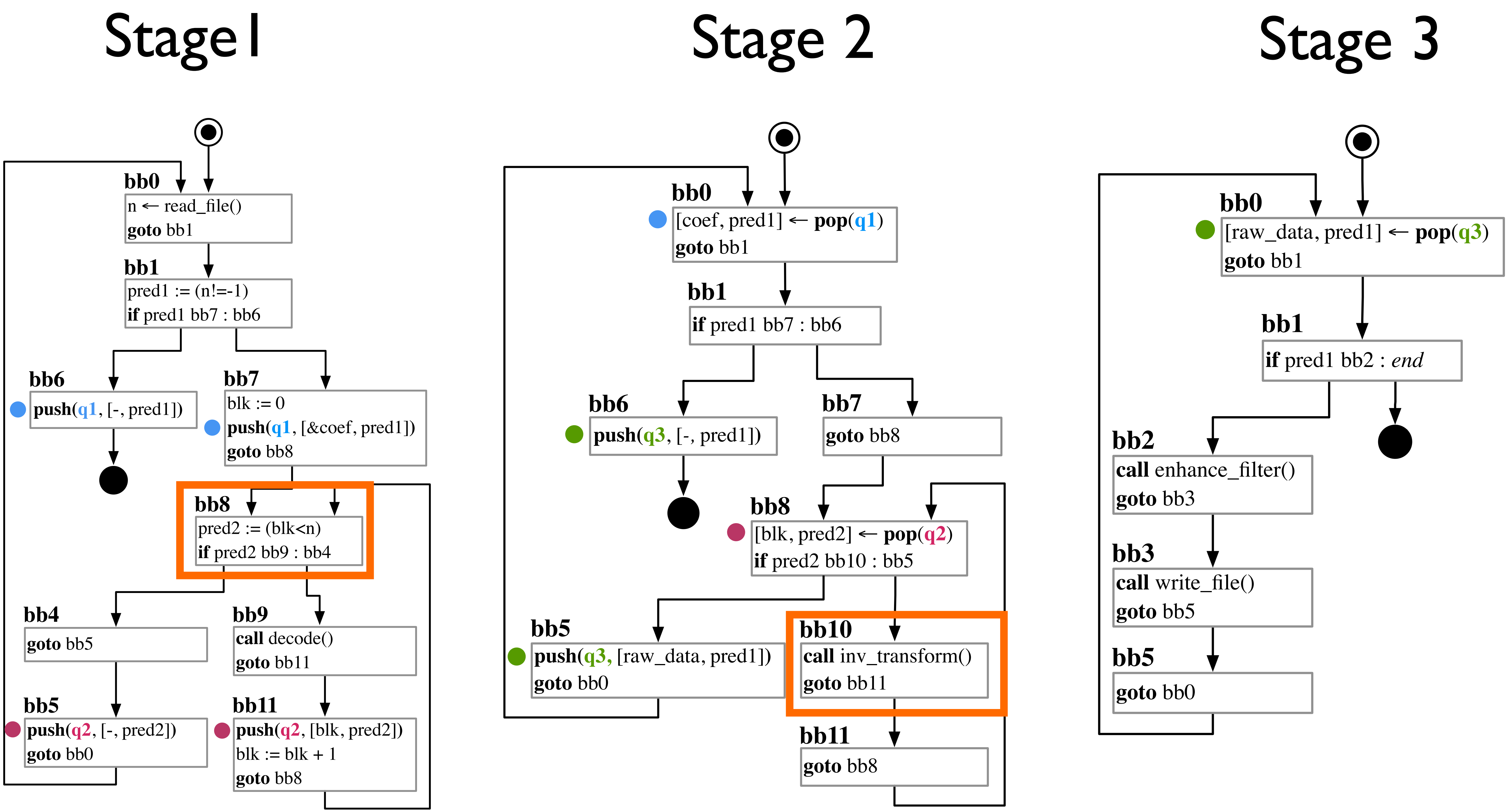
Stage 2



Stage 3

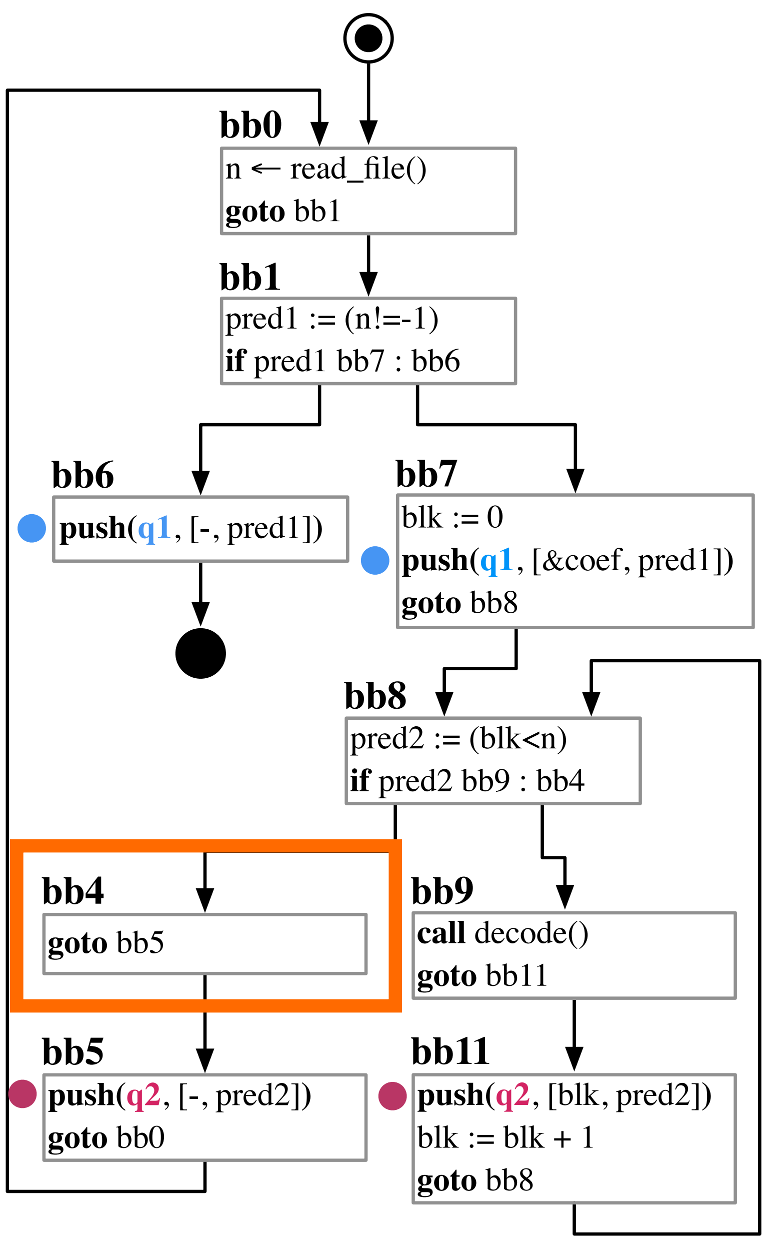


Communication

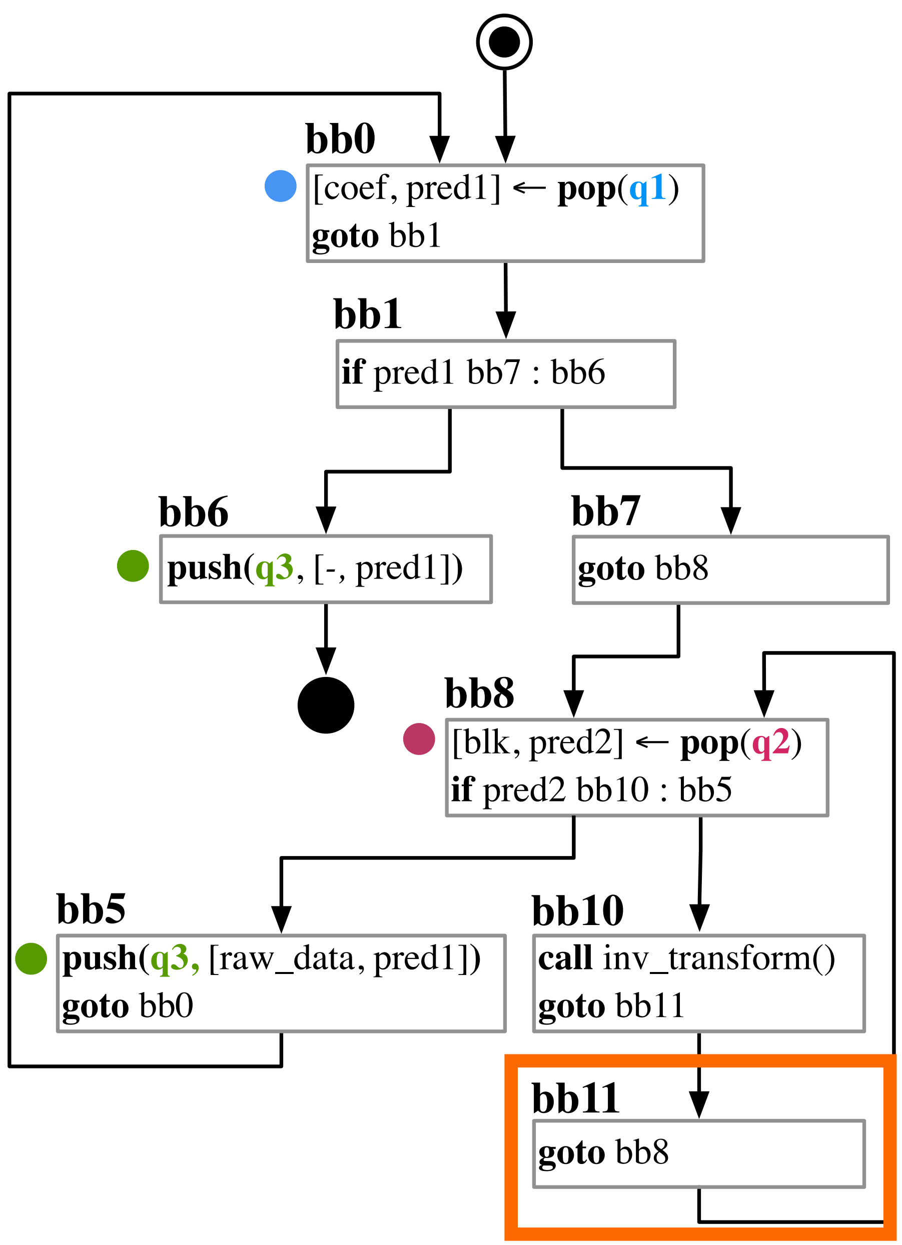


Communication

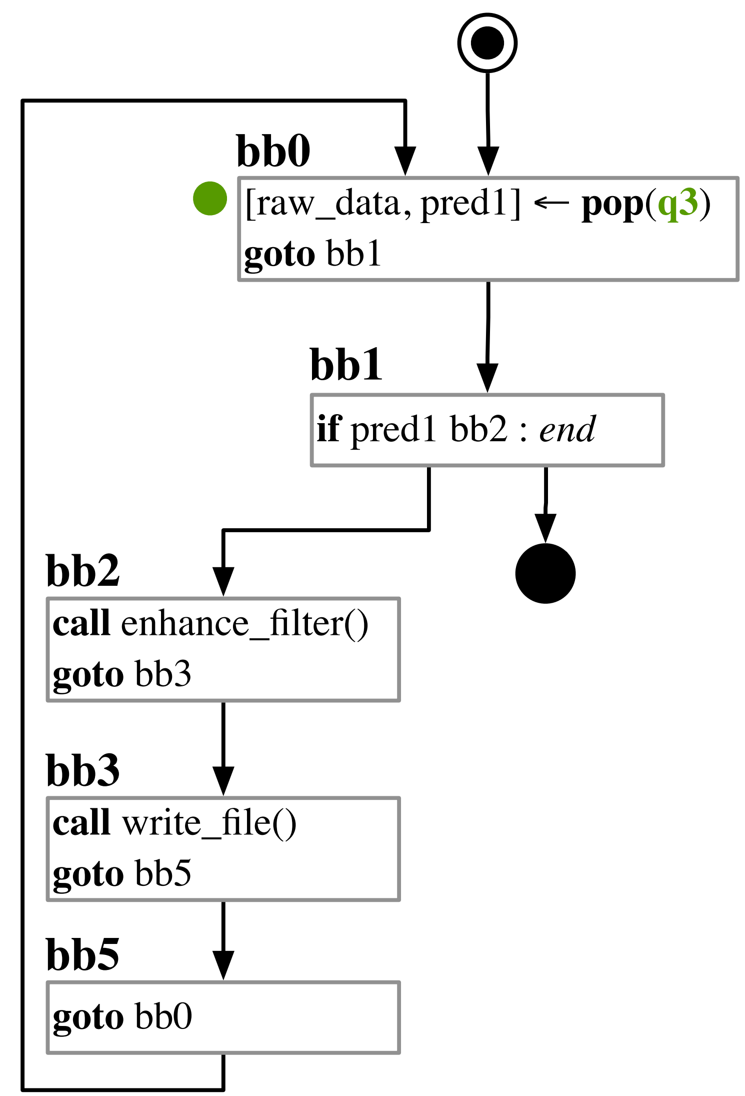
Stage 1



Stage 2

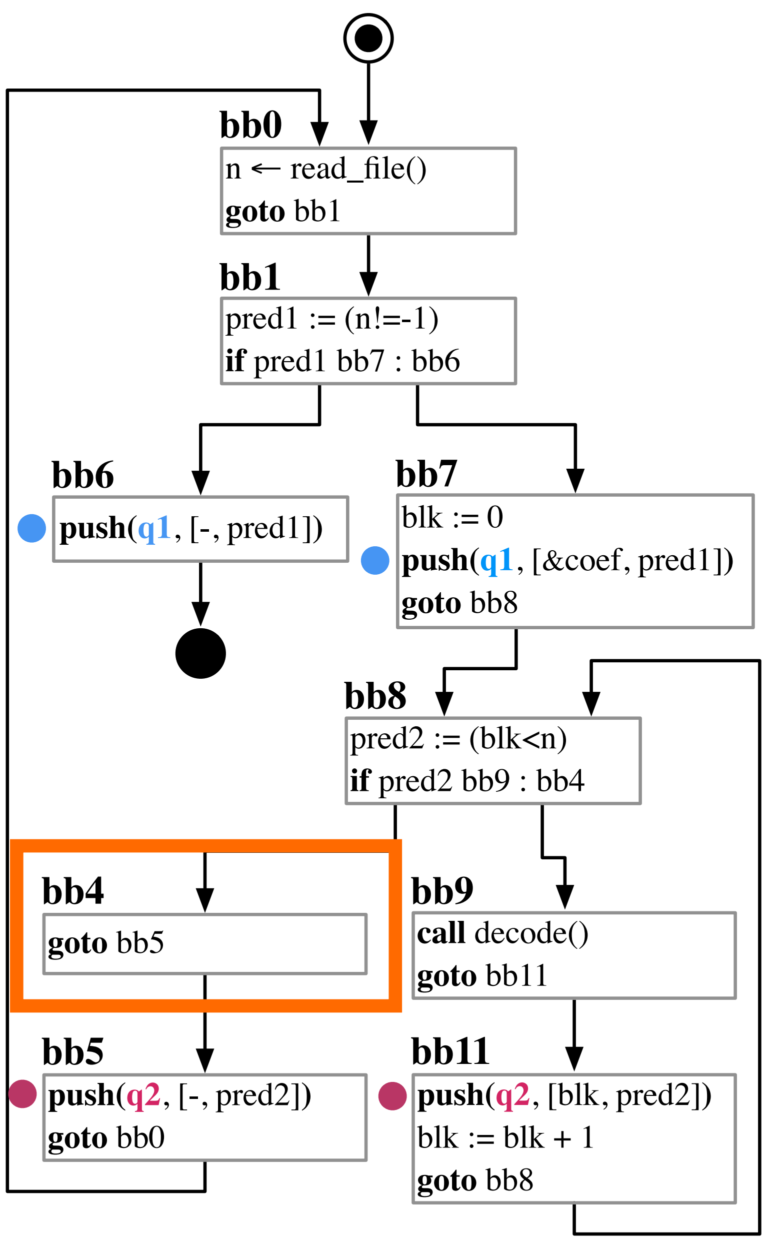


Stage 3

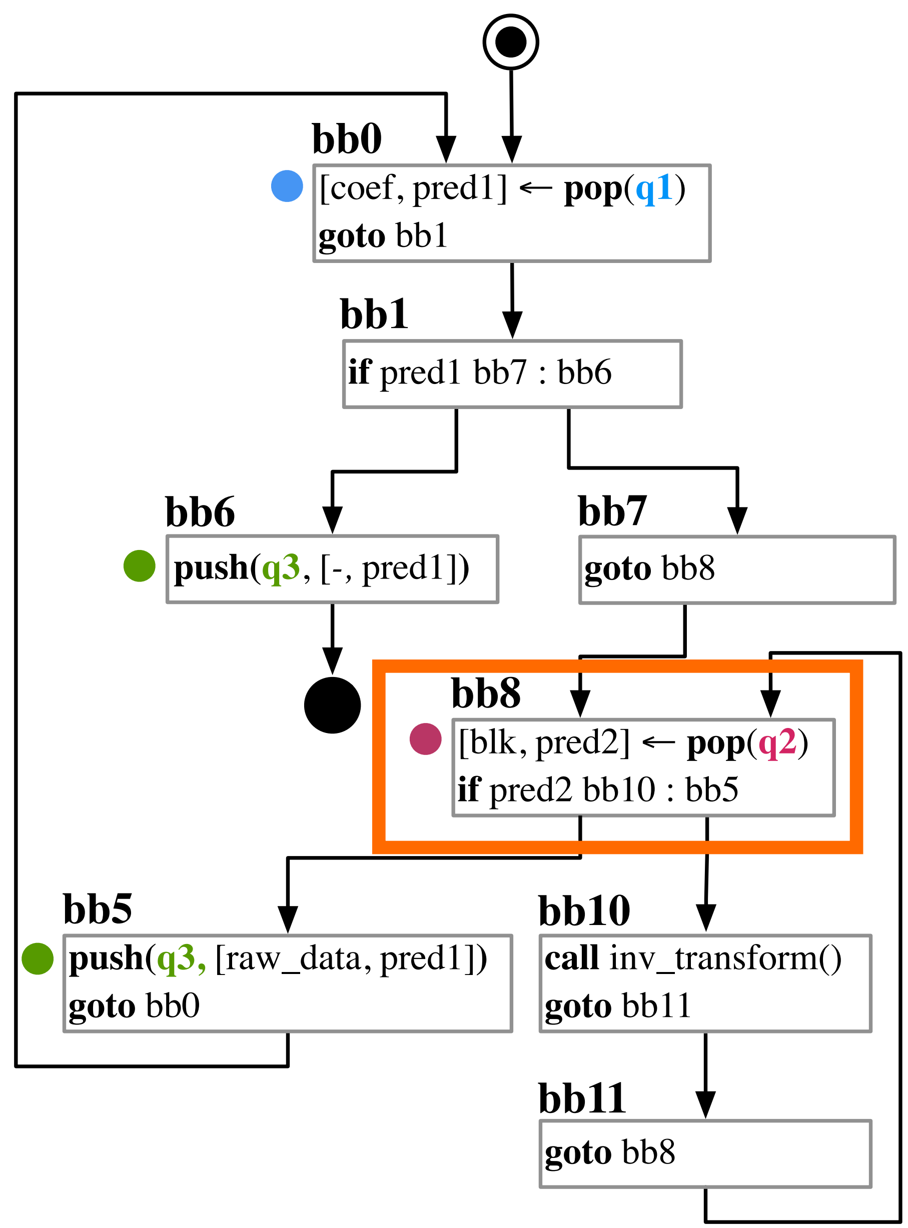


Communication

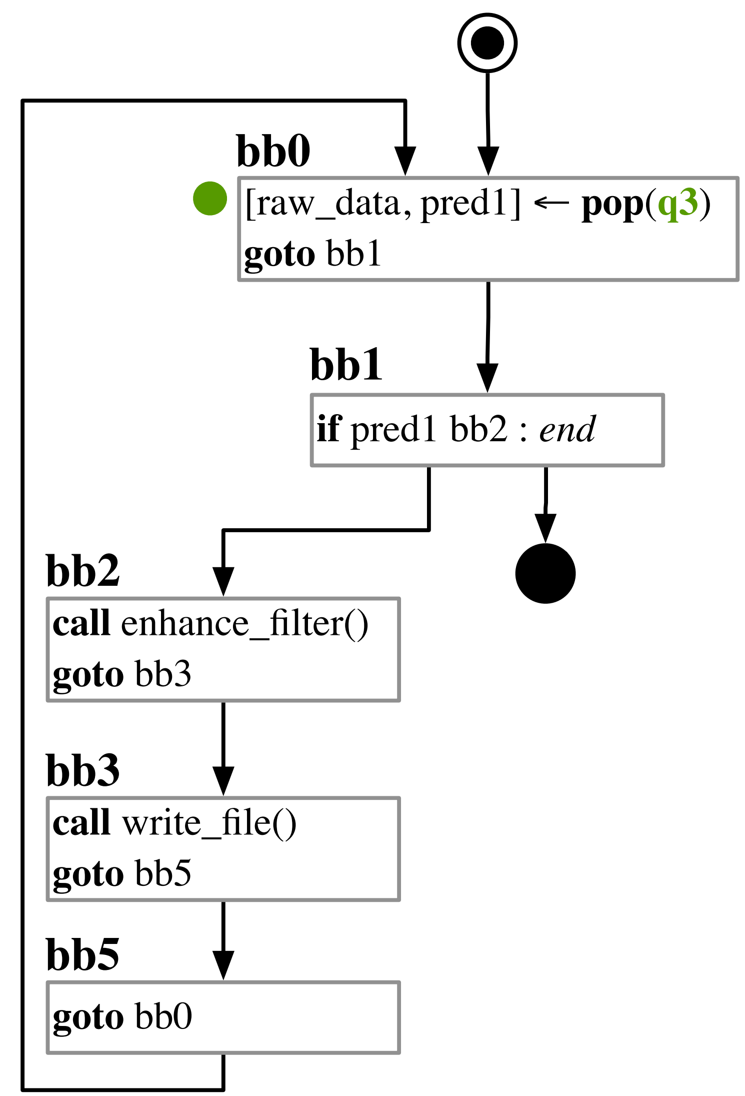
Stage 1



Stage 2

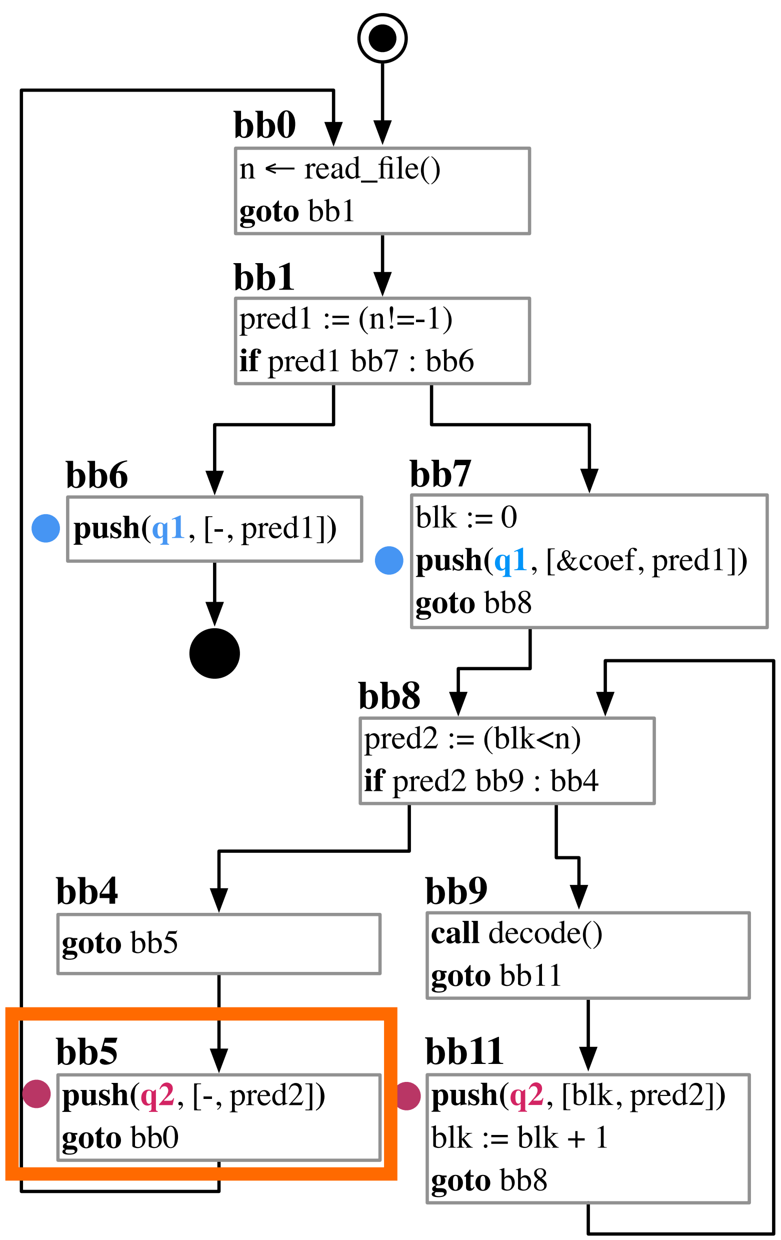


Stage 3

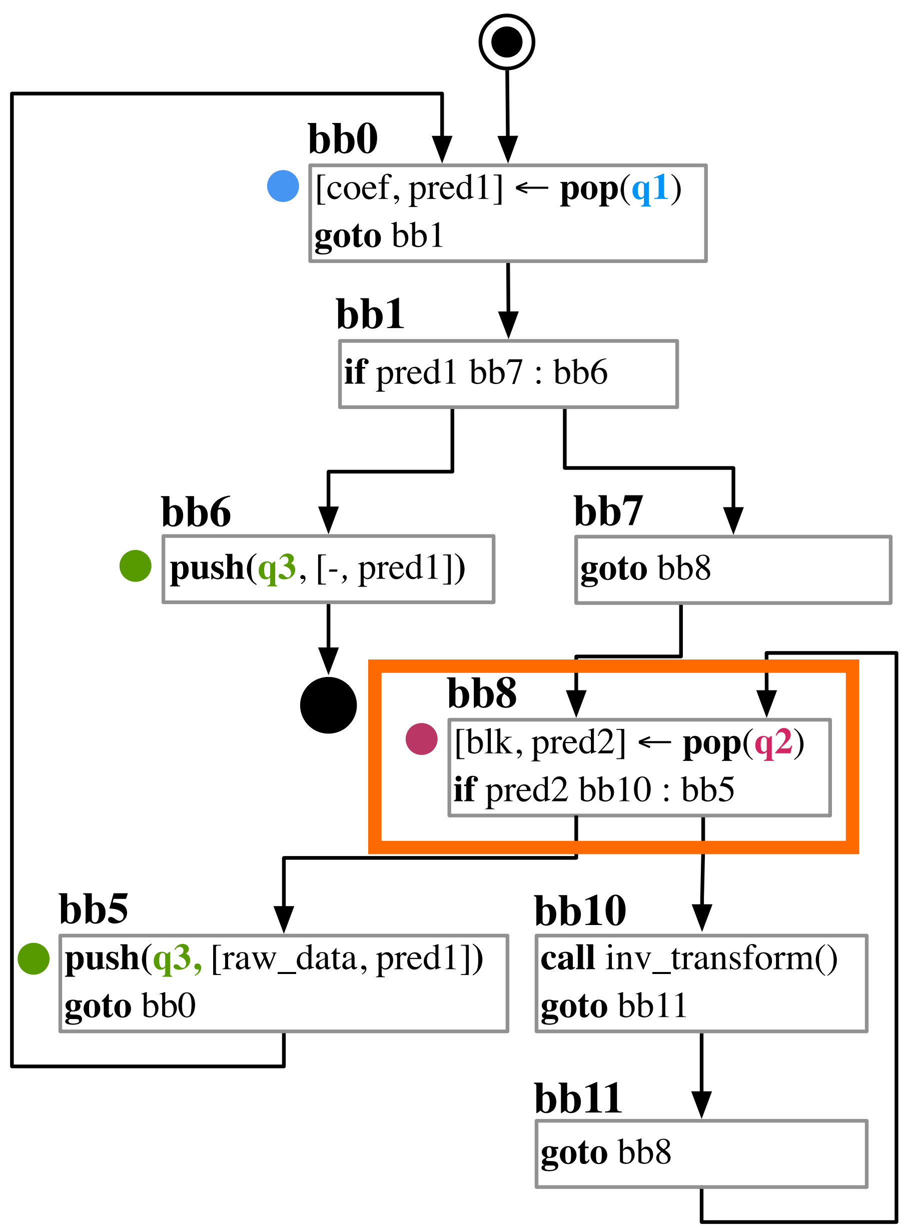


Communication

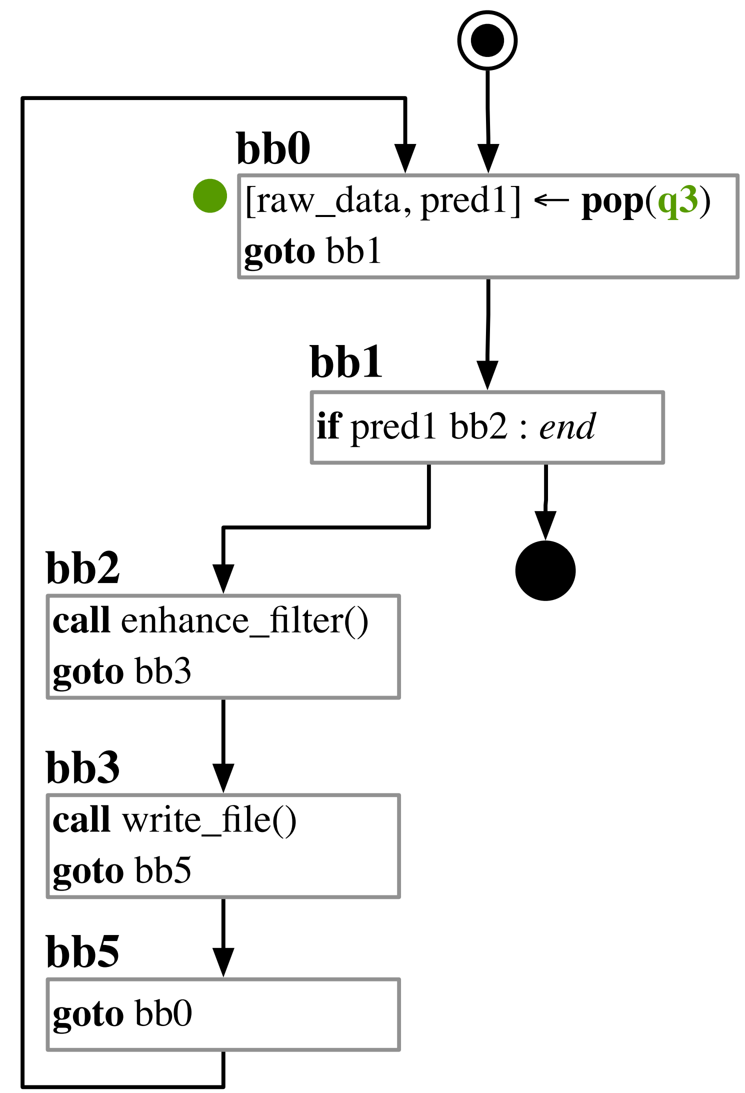
Stage 1



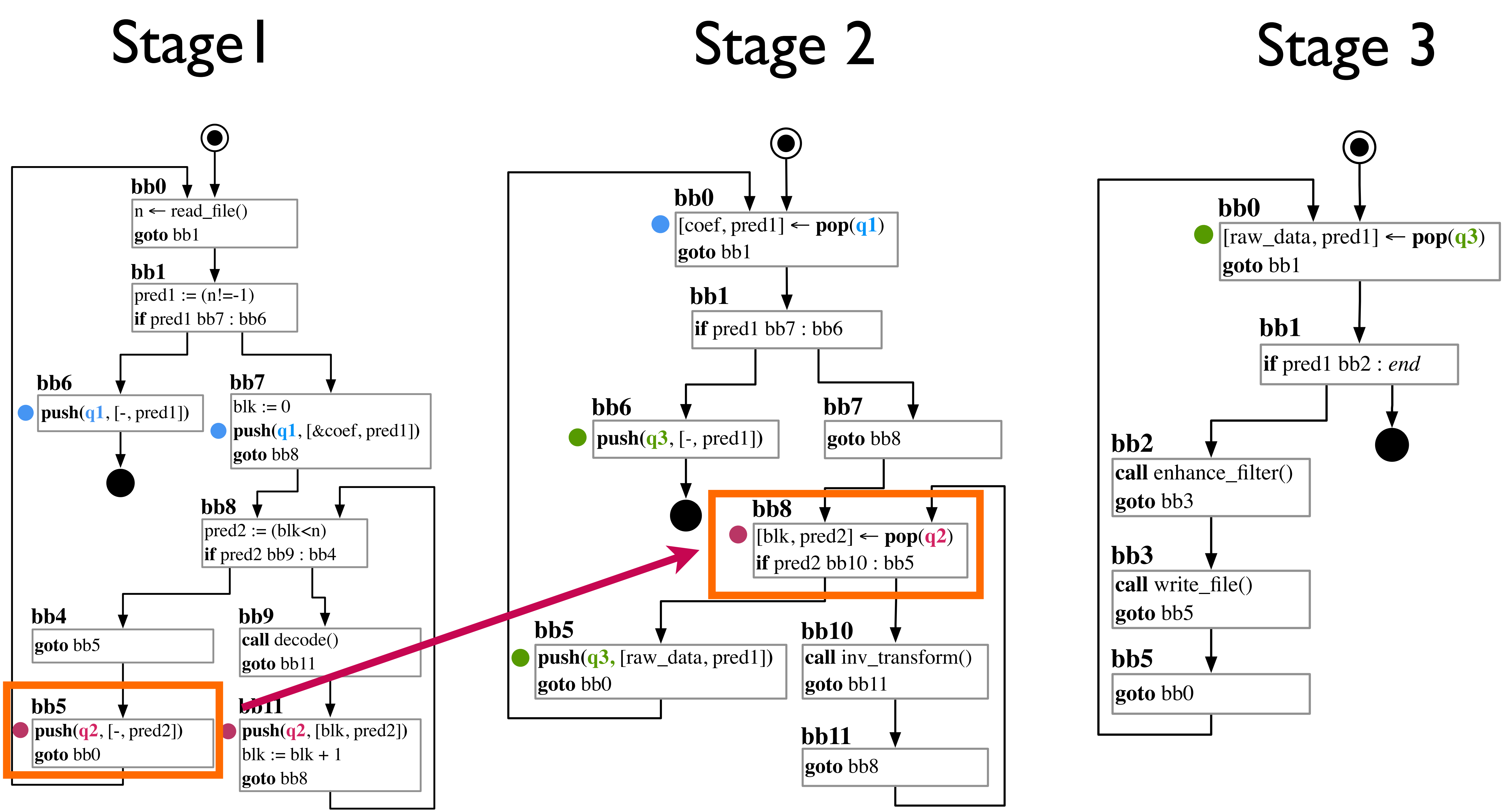
Stage 2



Stage 3

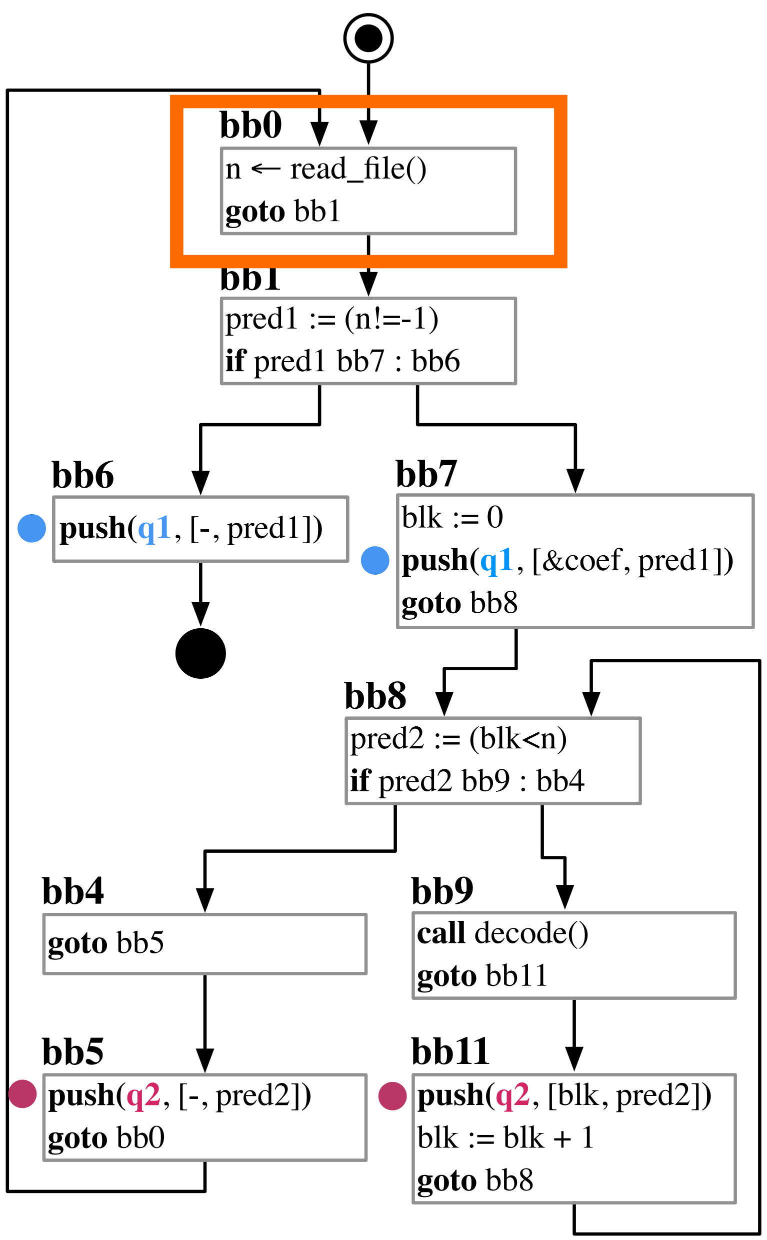


Communication

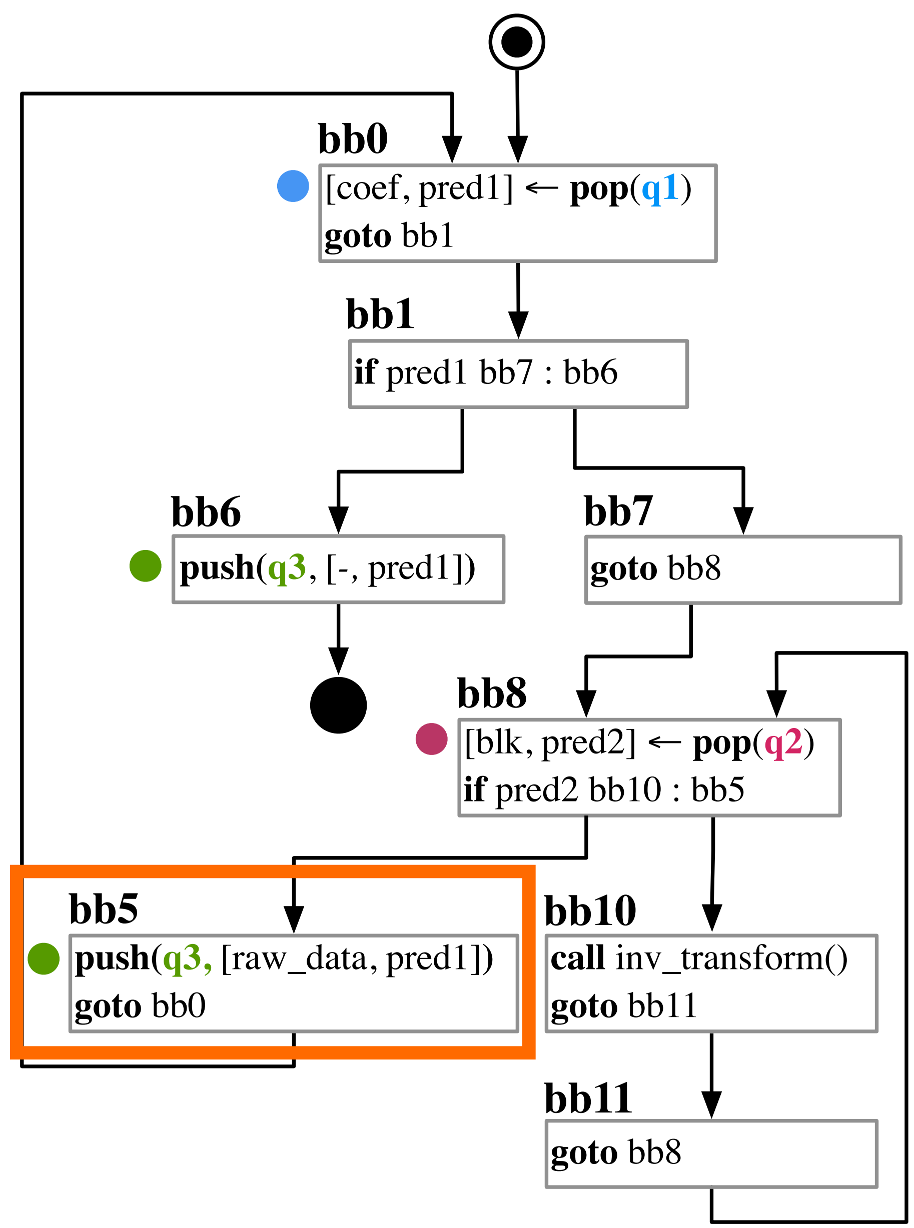


Communication

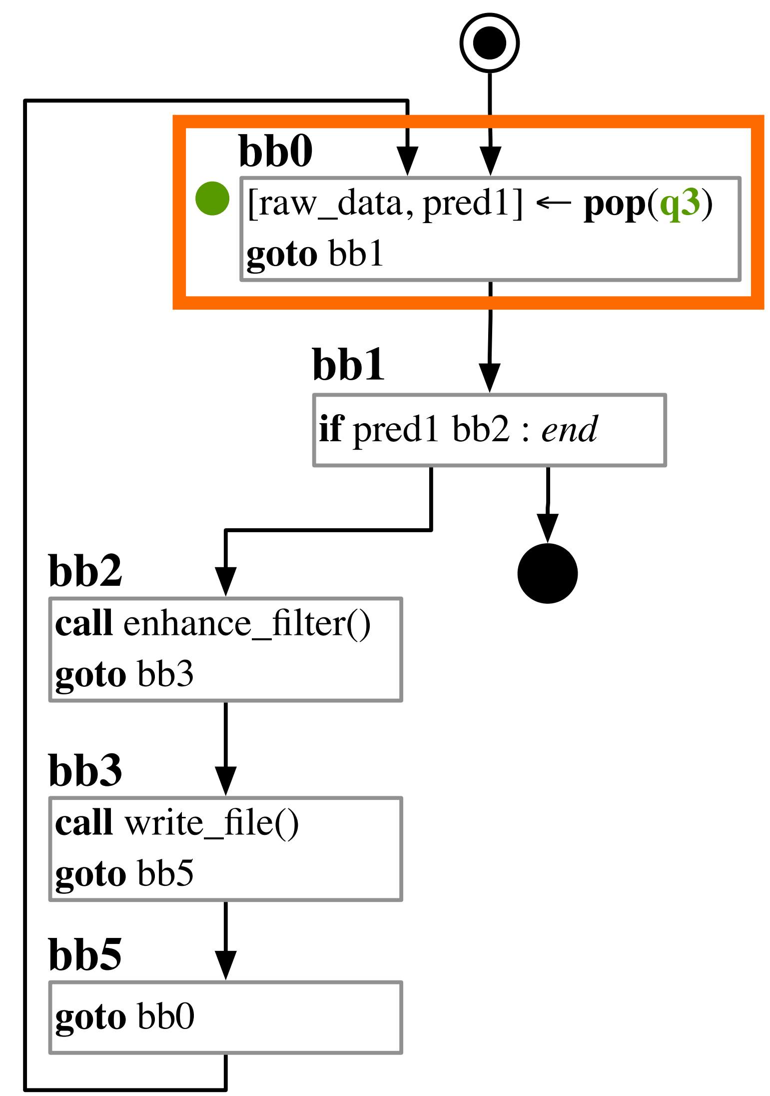
Stage 1



Stage 2

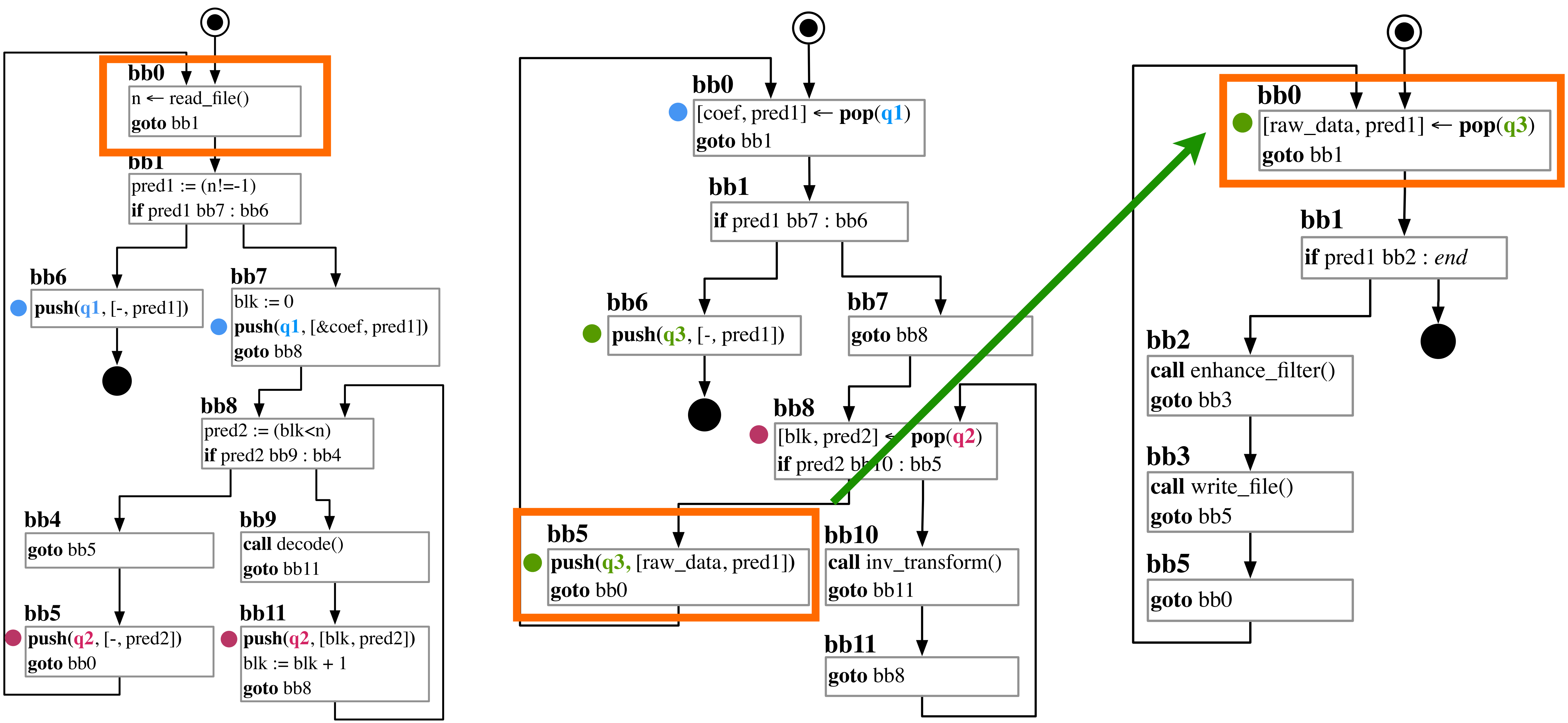


Stage 3

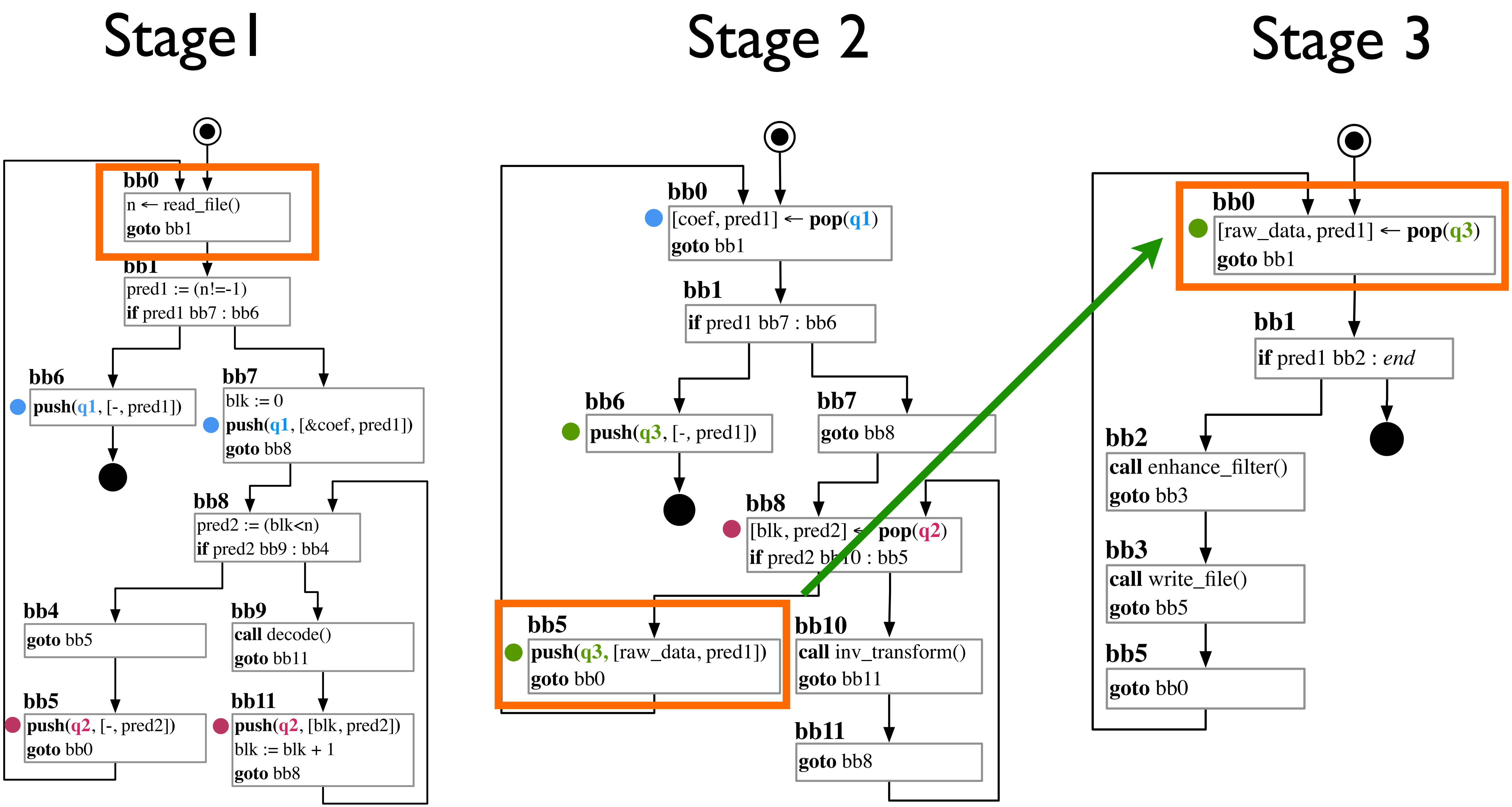


Communication

Stage 1 Stage 2 Stage 3

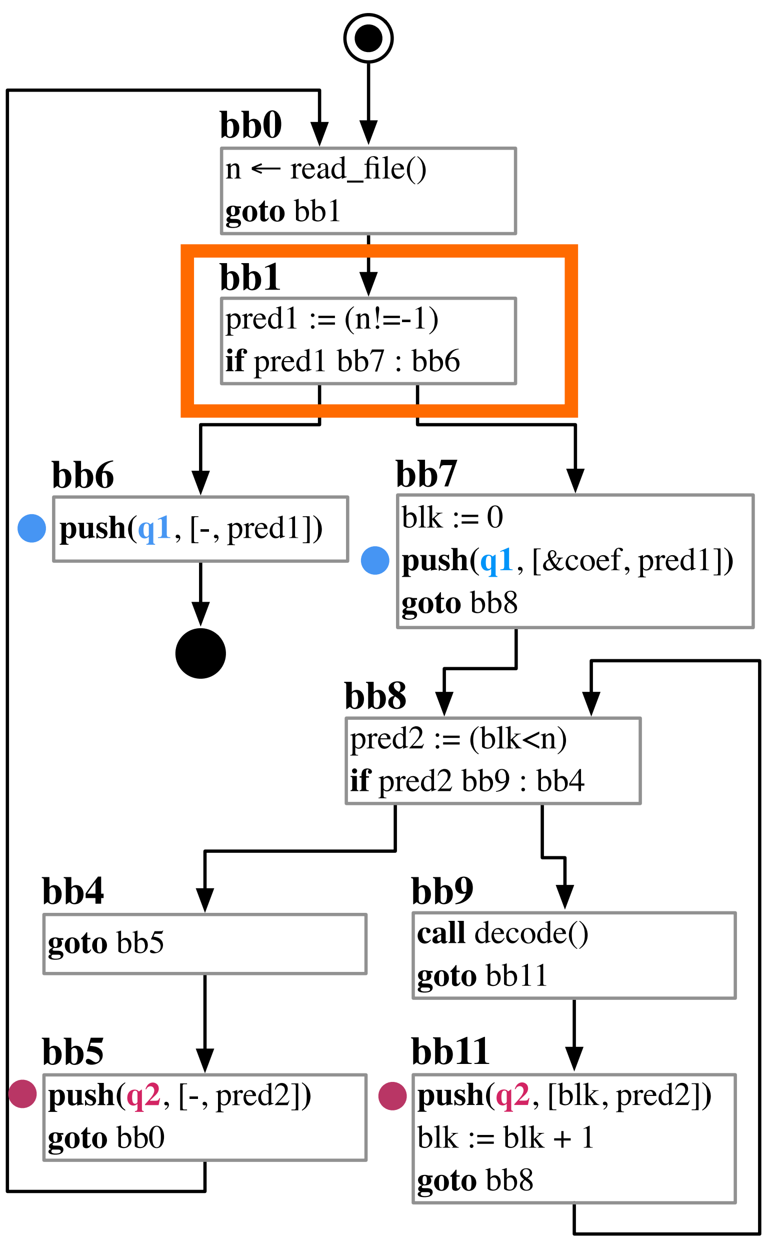


Communication

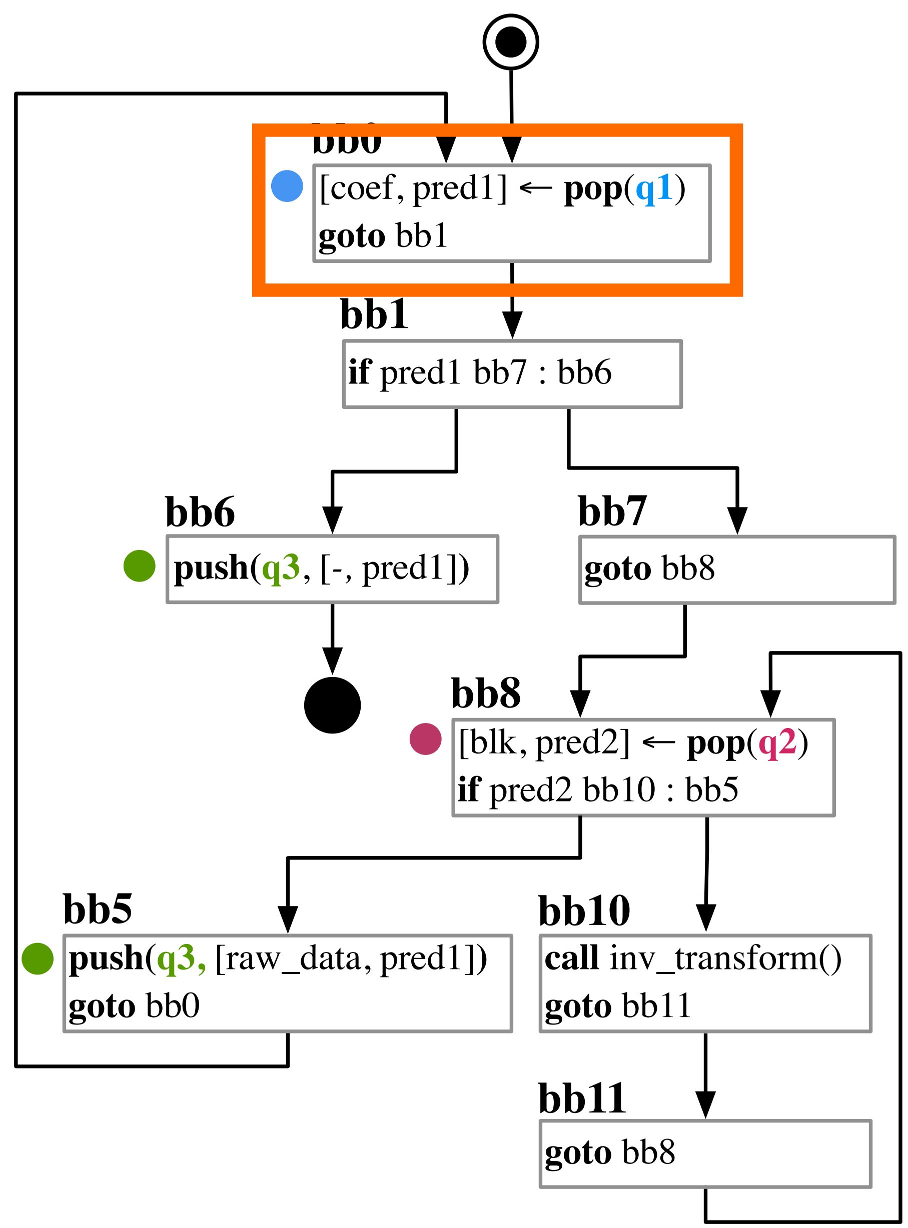


Communication

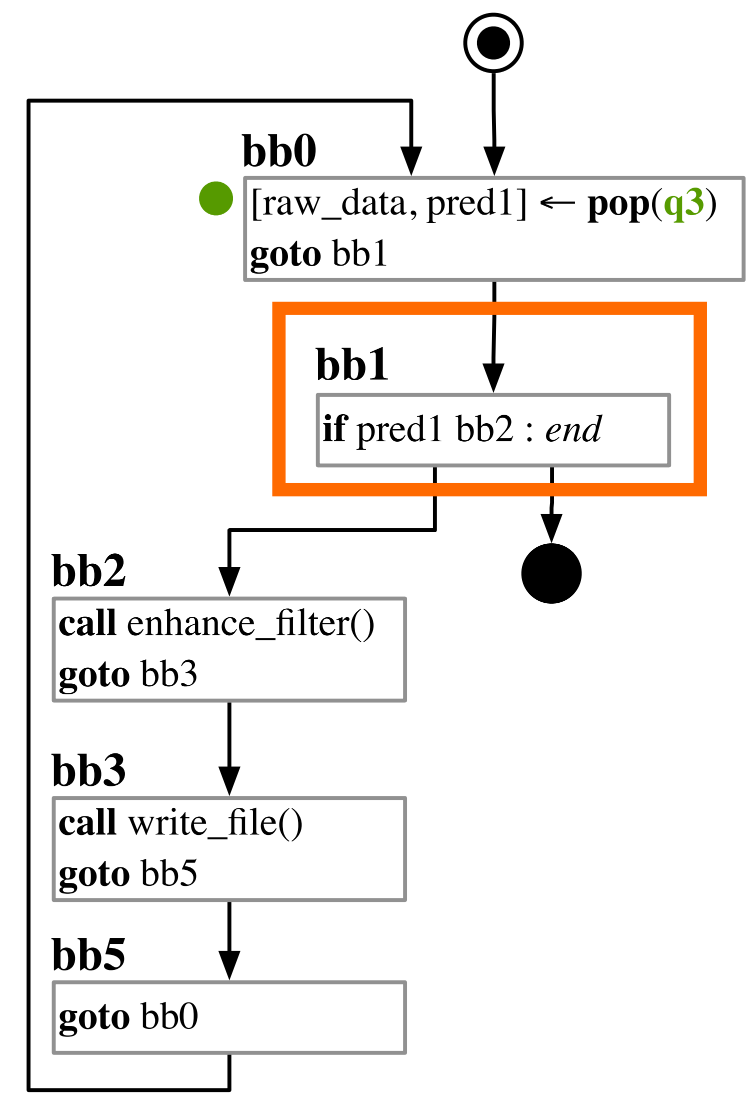
Stage 1



Stage 2

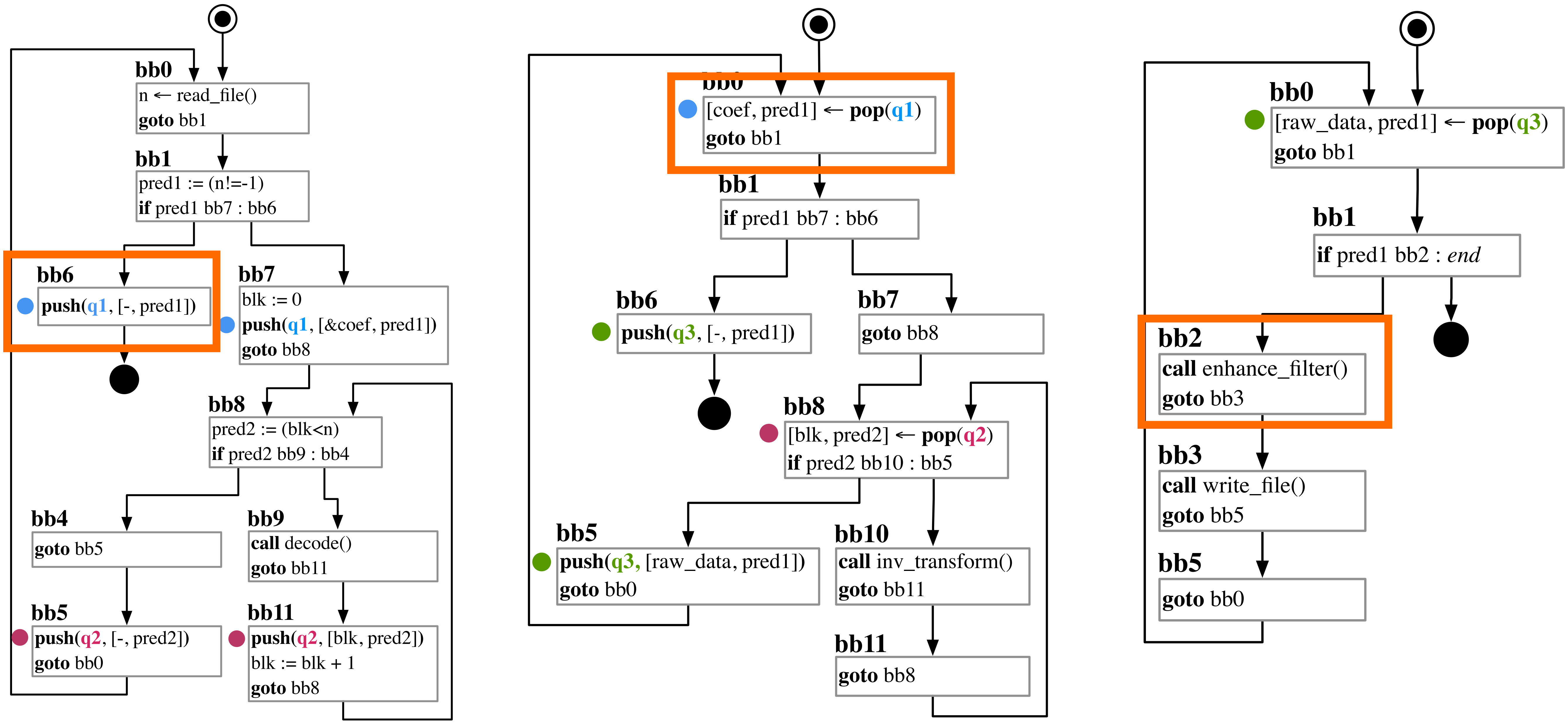


Stage 3

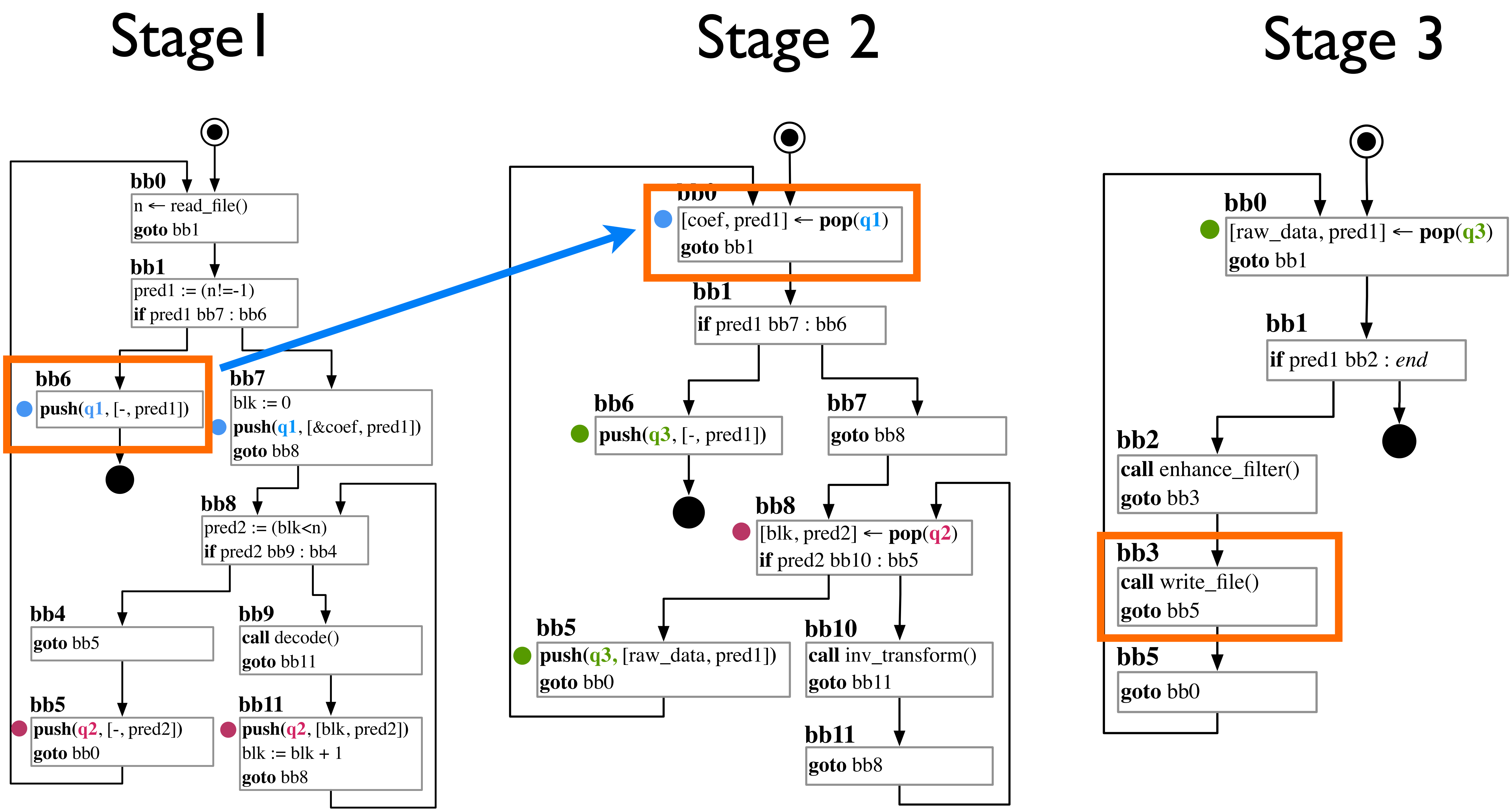


Communication

Stage 1 | Stage 2 | Stage 3

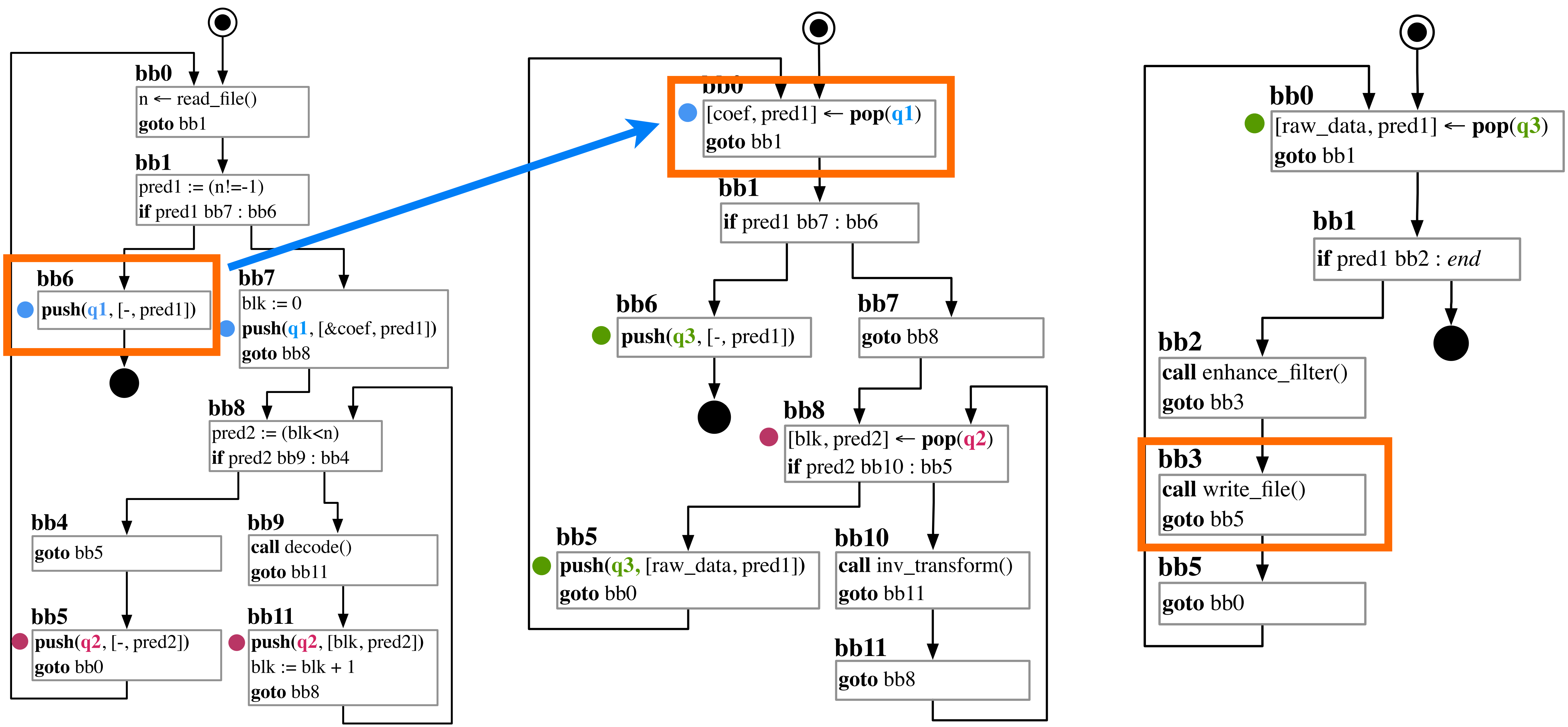


Communication



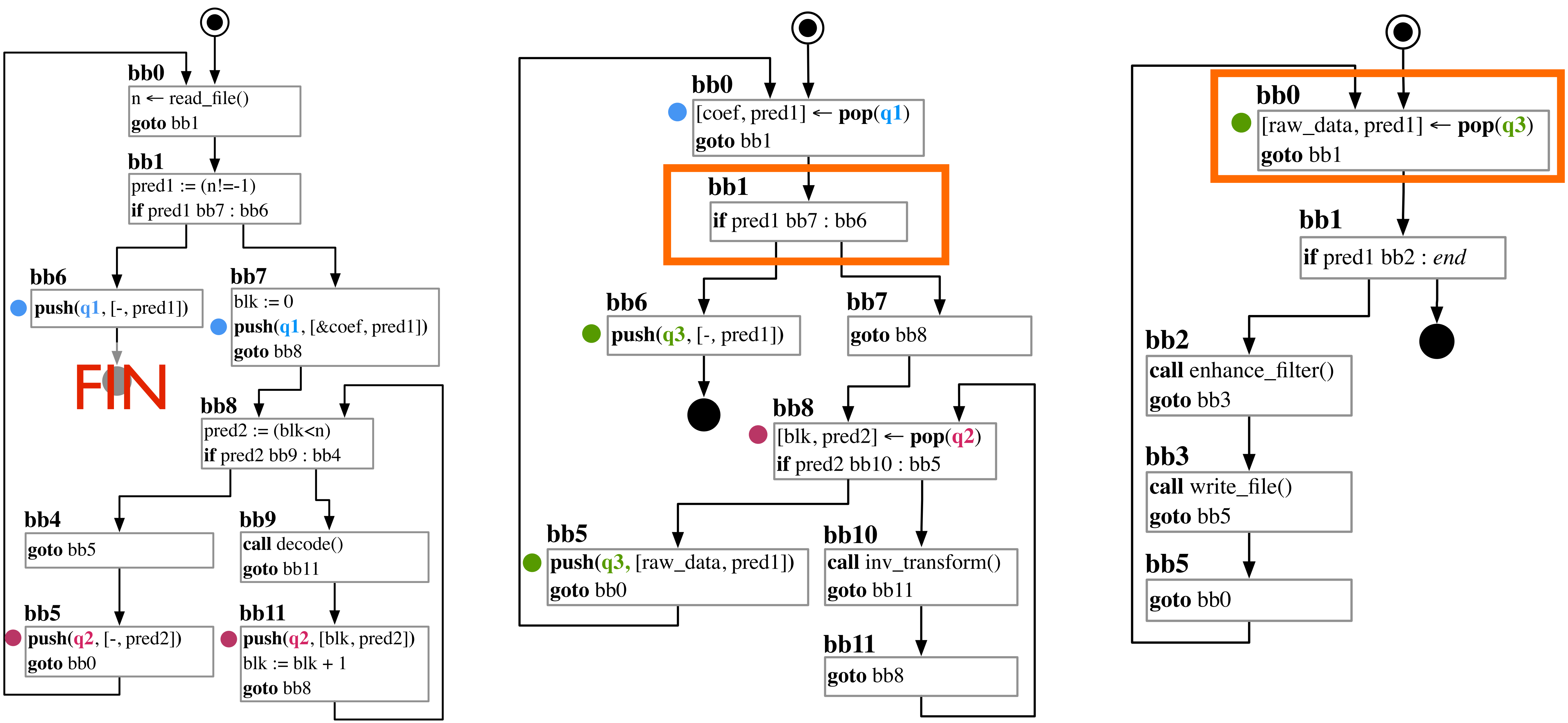
Communication

Stage 1 | Stage 2 | Stage 3

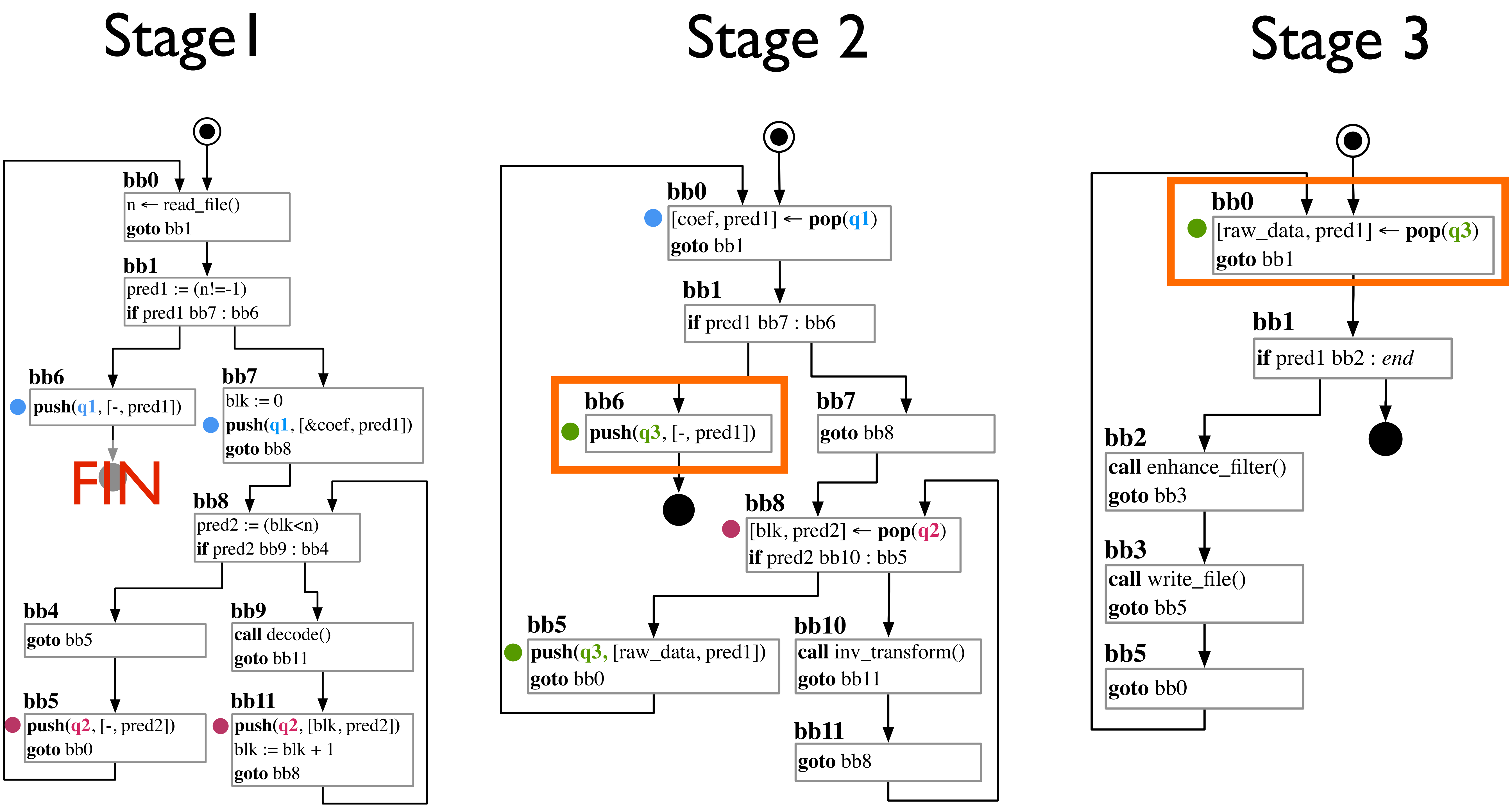


Communication

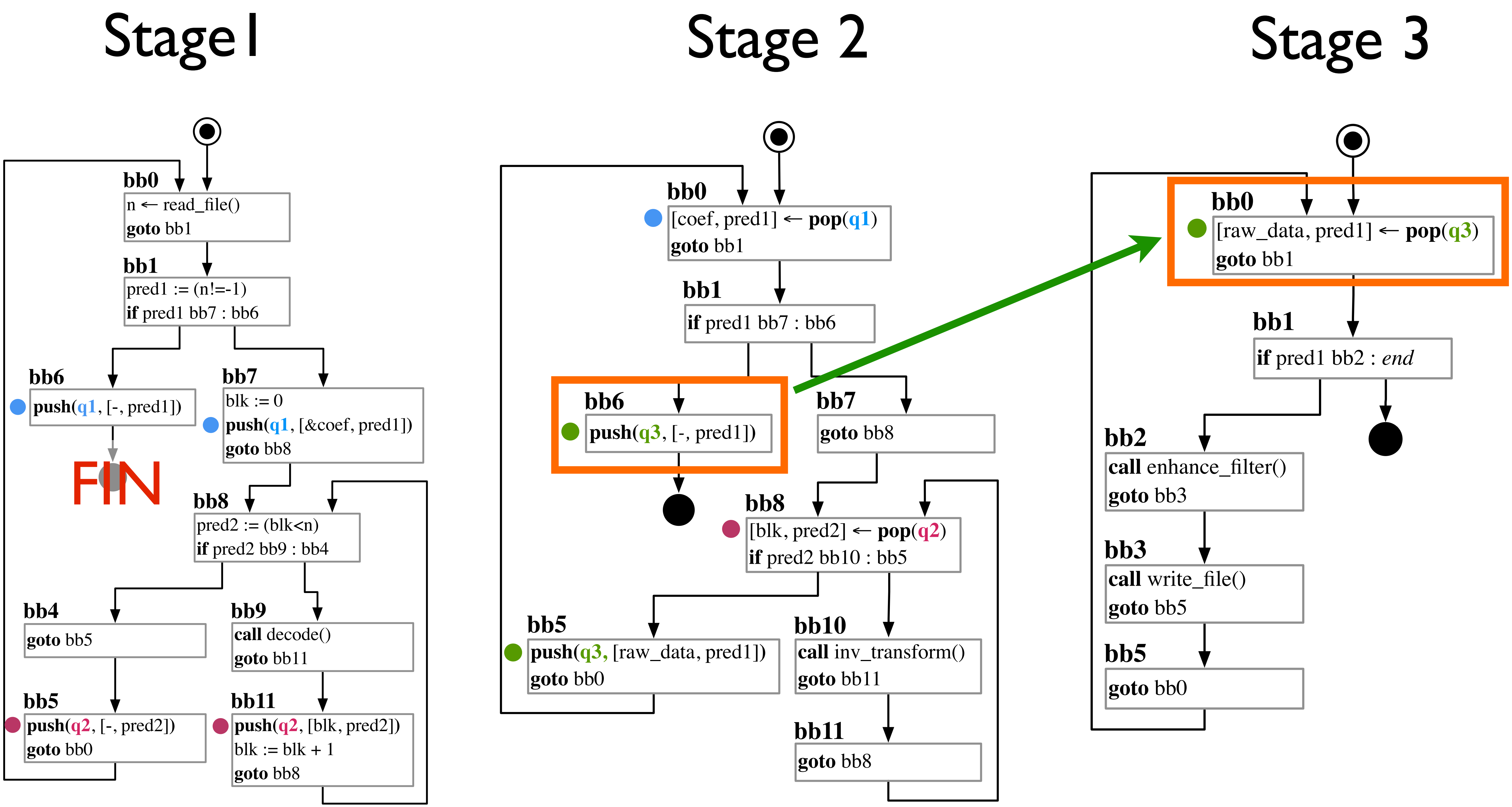
Stage 1 | Stage 2 | Stage 3



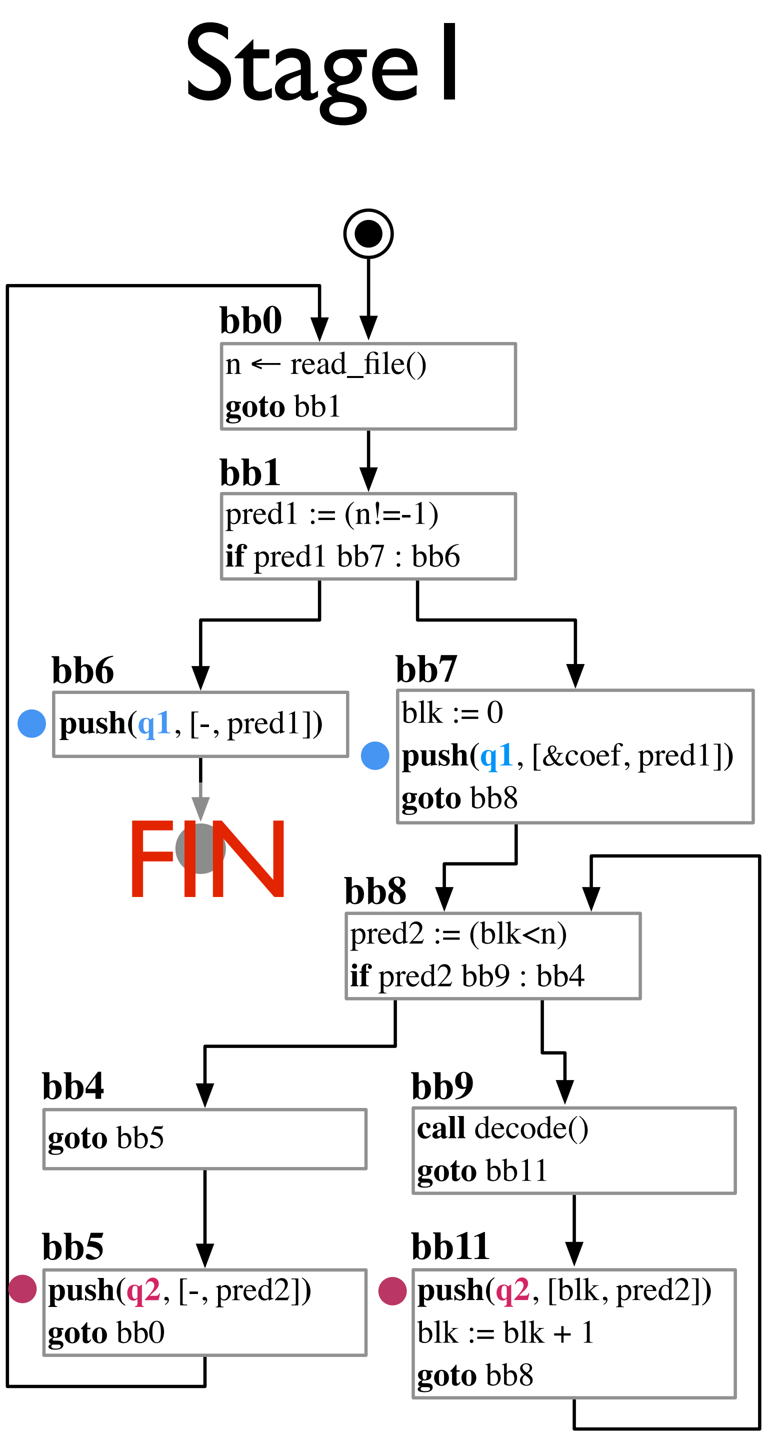
Communication

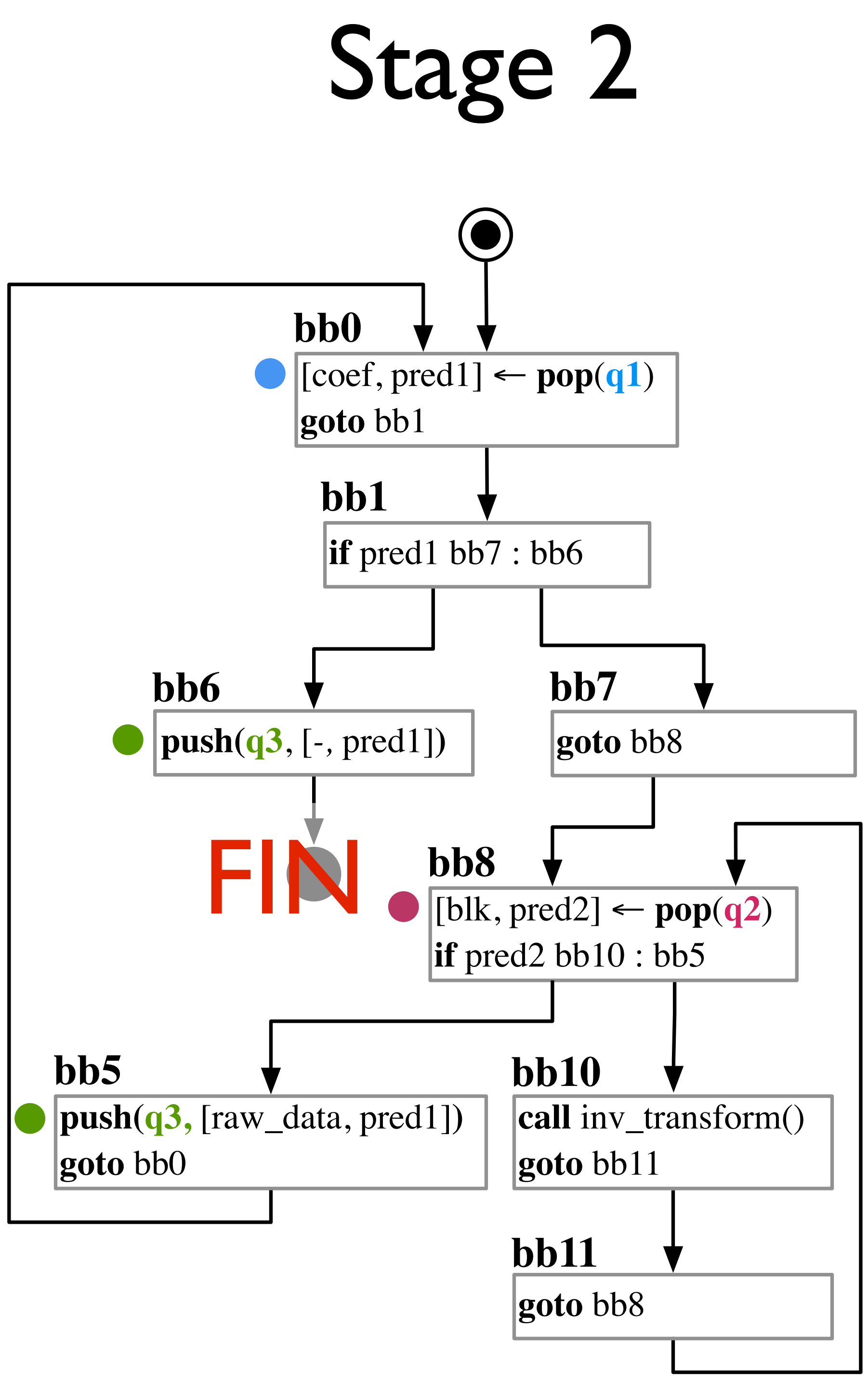


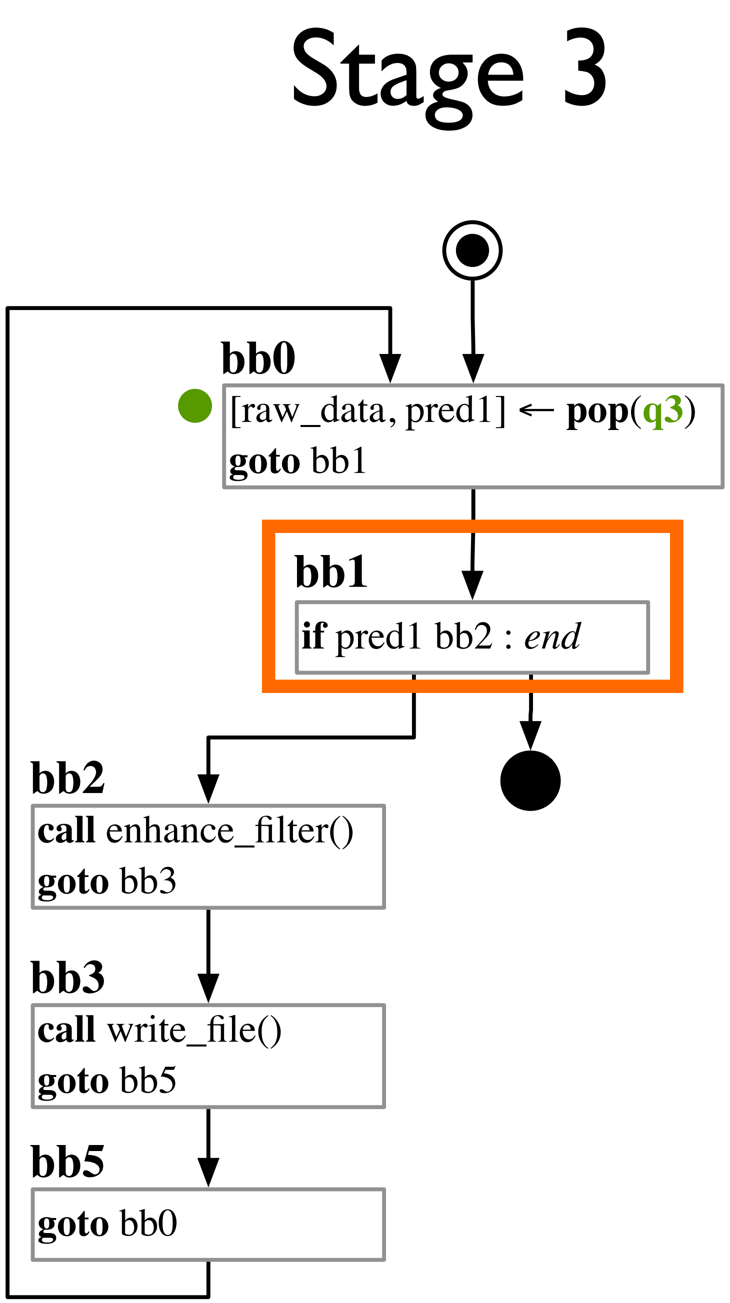
Communication



Communication







Communication