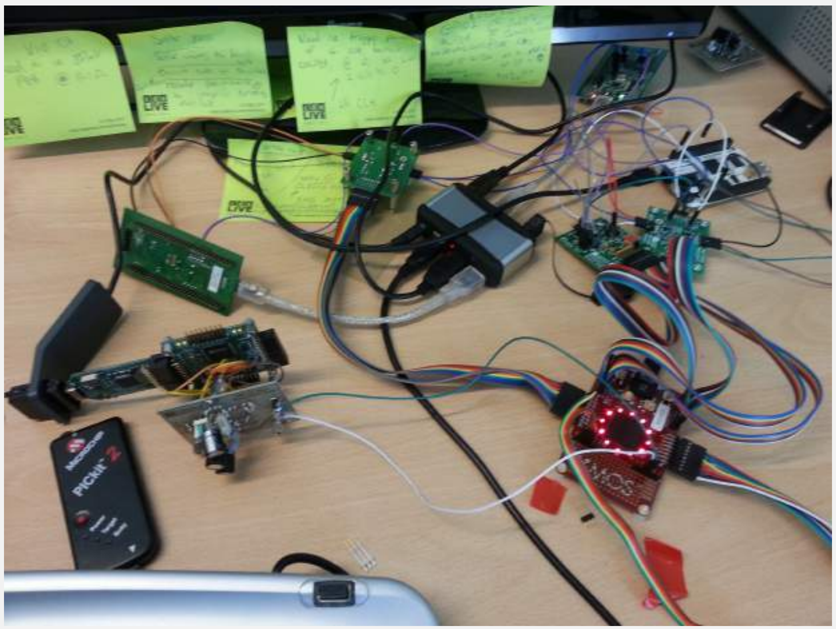
Scalability in Compiler Development

How to Get Testing and Optimization Done in a Reasonable Time

Jeremy Bennett

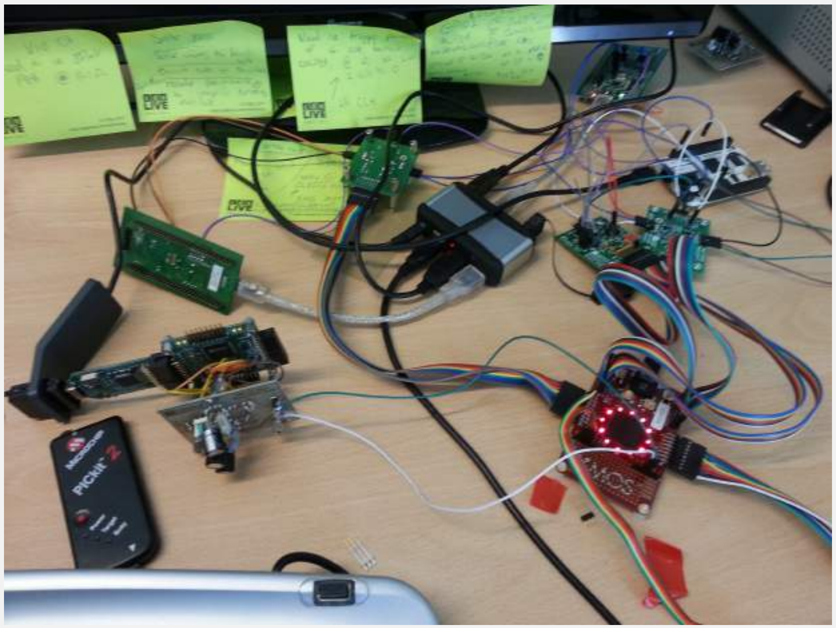
Machine Learning Compilers

Do Compilers Affect Energy?



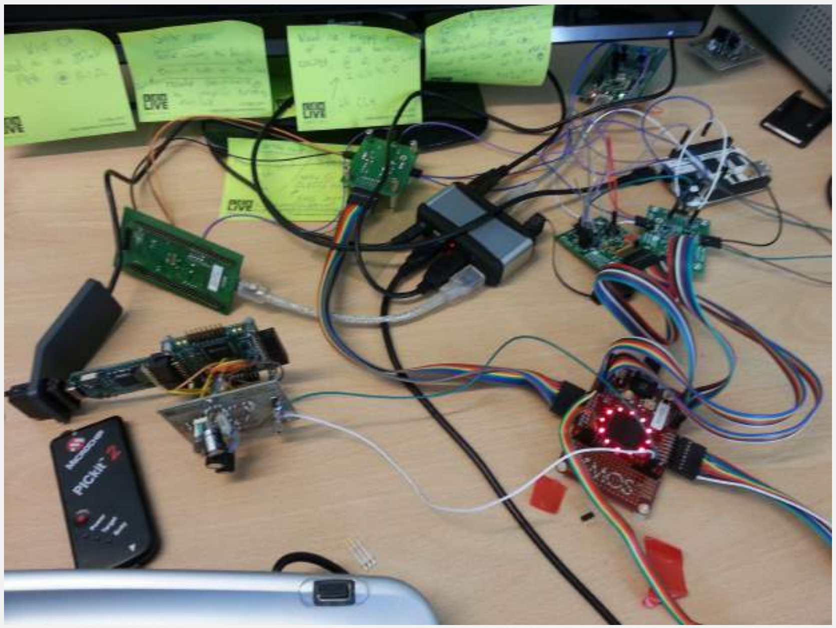
- Initial research in 2012 by Embecosm and Bristol University

Do Compilers Affect Energy?



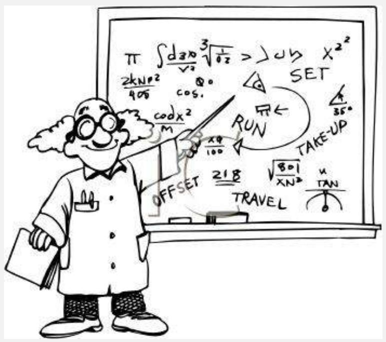
- Initial research in 2012 by Embecosm and Bristol University
- The answer is "yes"

Do Compilers Affect Energy?

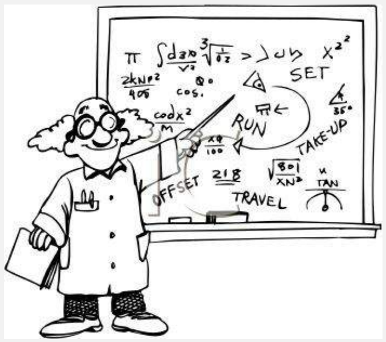


- Initial research in 2012 by Embecosm and Bristol University
- The answer is "yes"
- Now published *open access* in a peer-reviewed journal

Identifying Compiler Options to Minimize Energy Consumption for Embedded Platforms
James Pallister; Simon J. Hollis; Jeremy Bennett
The Computer Journal 2013; doi: 10.1093/comjnl/bxt129
http://comjnl.oxfordjournals.org/cgi/reprint/bxt129?ijkey=aA4RYlYQLNVgkE3



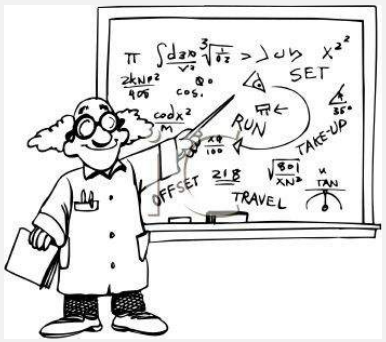
Research into feedback directed optimization



Research into feedback directed optimization

Research into modeling energy usage

MAGEEC: Machine Guided Energy Efficient Compilation

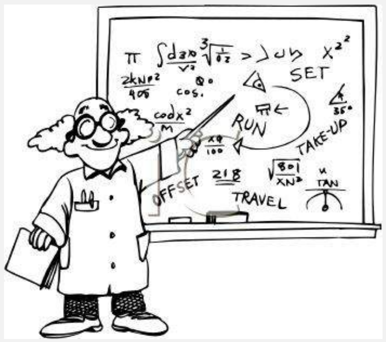


MILEPOST

Research into feedback directed optimization

Research into modeling energy usage

MAGEEC: Machine Guided Energy Efficient Compilation



MILEPOST

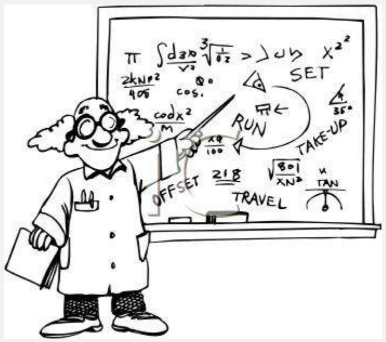
Research into feedback directed optimization

Research into modeling energy usage

Energy measurement

EMBECOSM®

MAGEEC: Machine Guided Energy Efficient Compilation



Research into feedback directed optimization

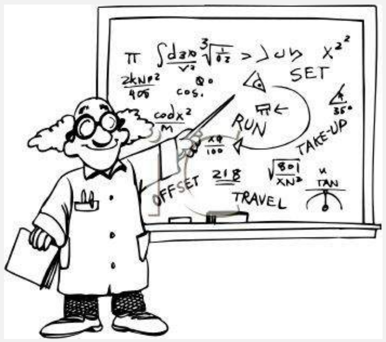
MILEPOST

Innovate UK
Technology Strategy Board

Research into modeling energy usage

Energy measurement

MAGEEC: Machine Guided Energy Efficient Compilation



Research into feedback directed optimization

MILEPOST

Innovate UK

Technology Strategy Board

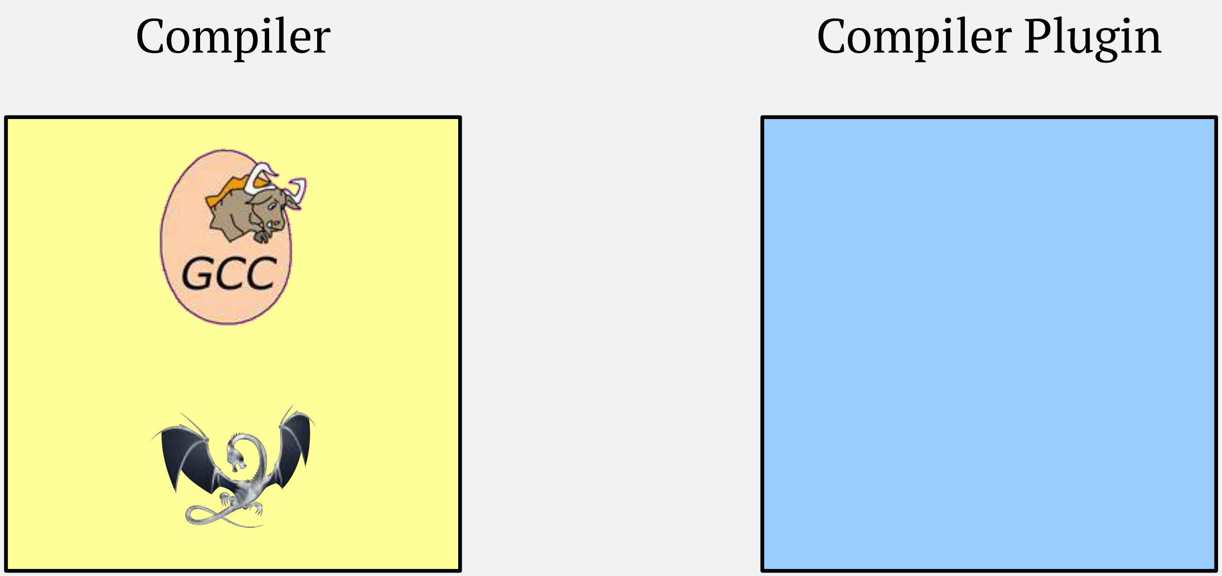
Research into modeling energy usage

Energy measurement

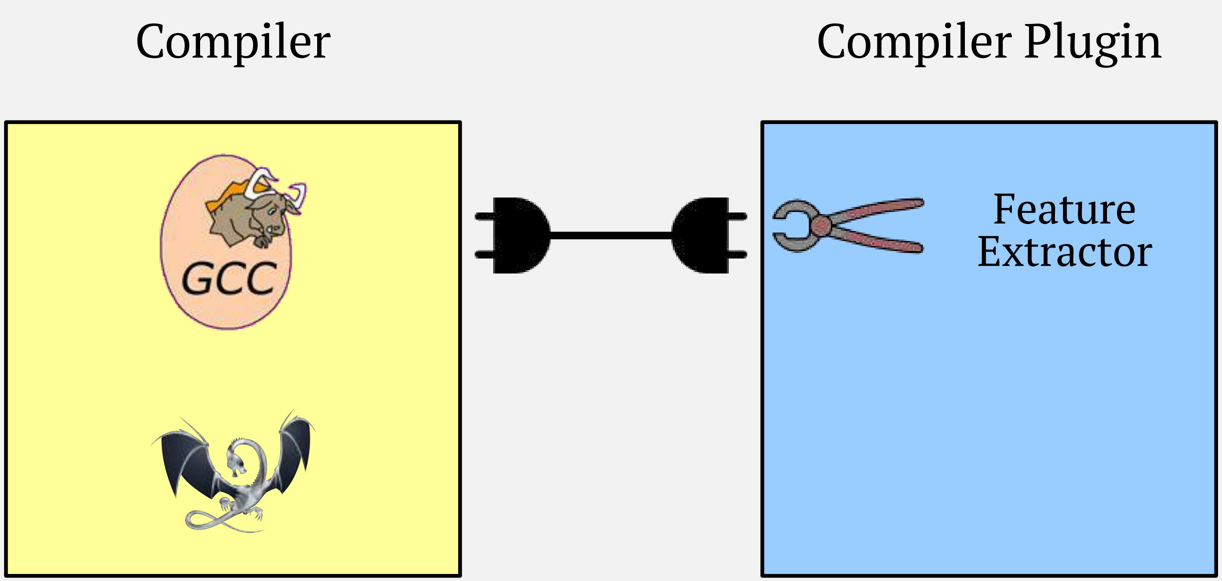
Overall Design

Compiler

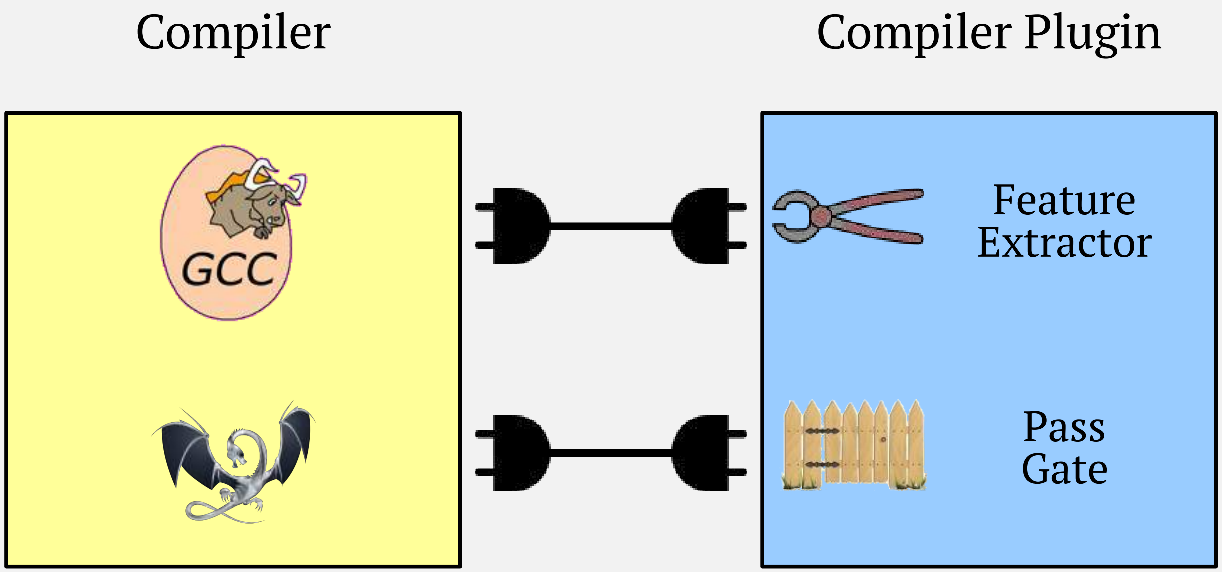
Overall Design



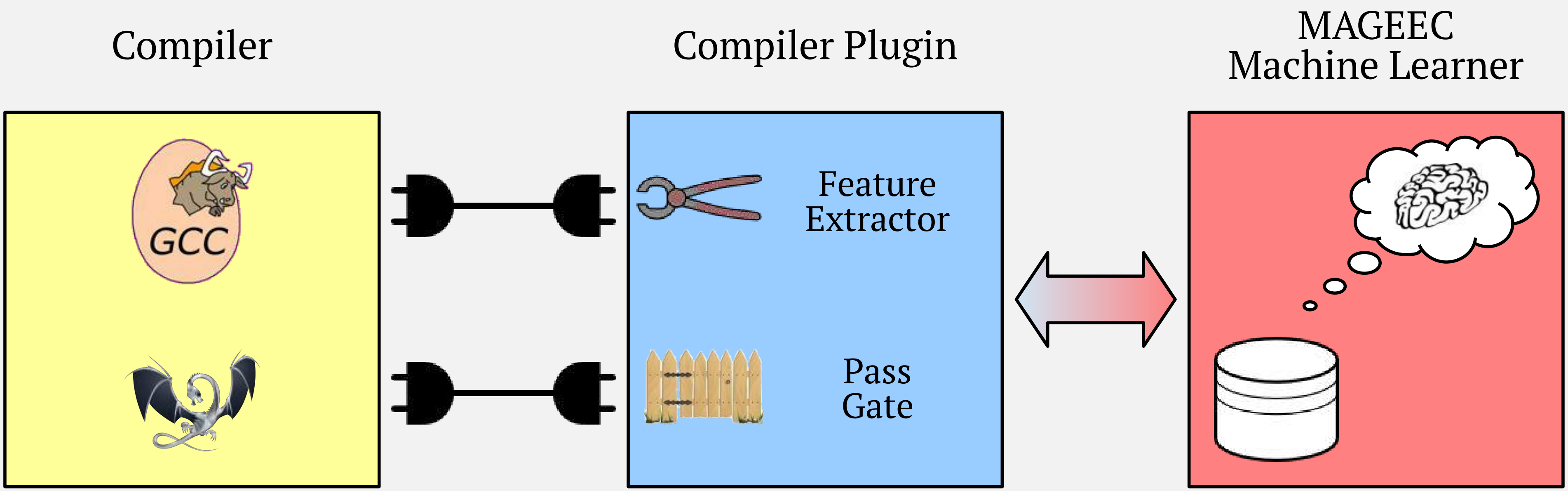
Overall Design



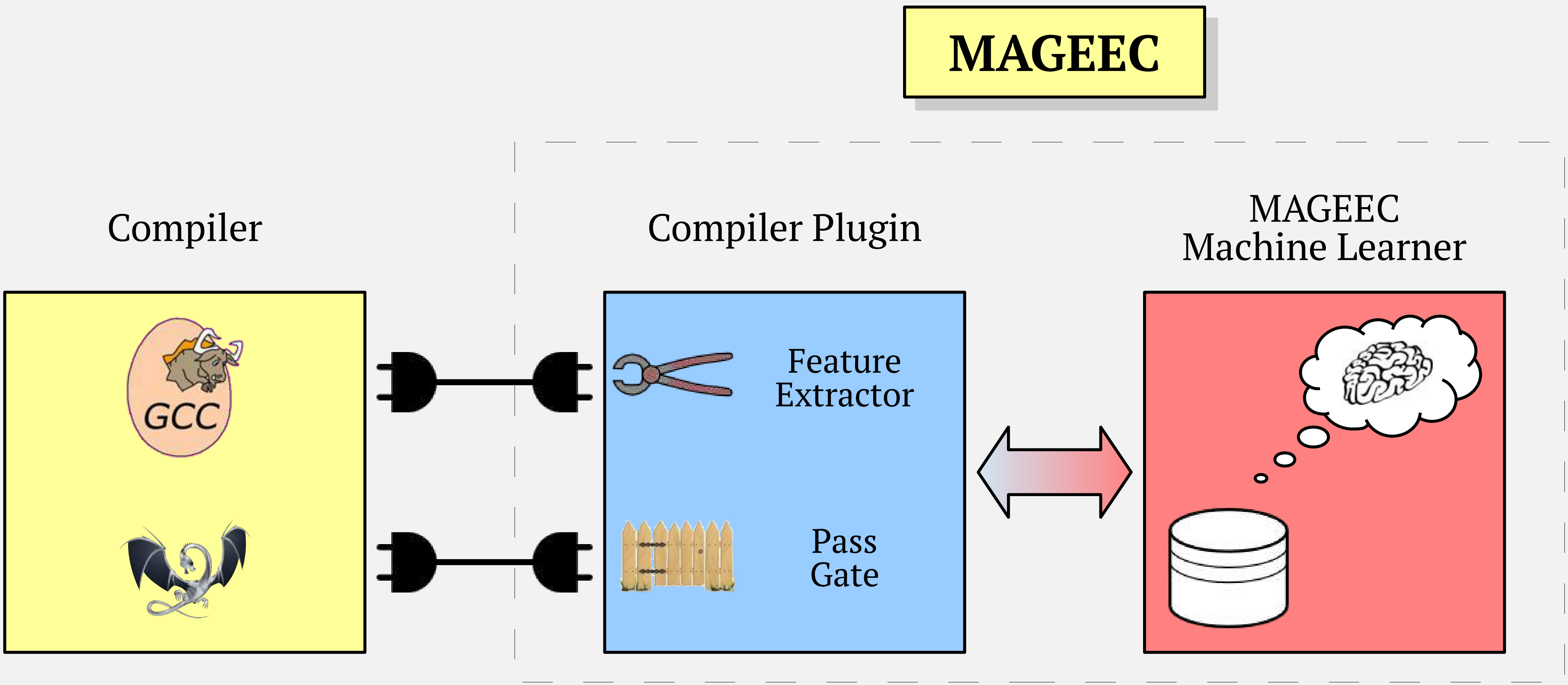
Overall Design



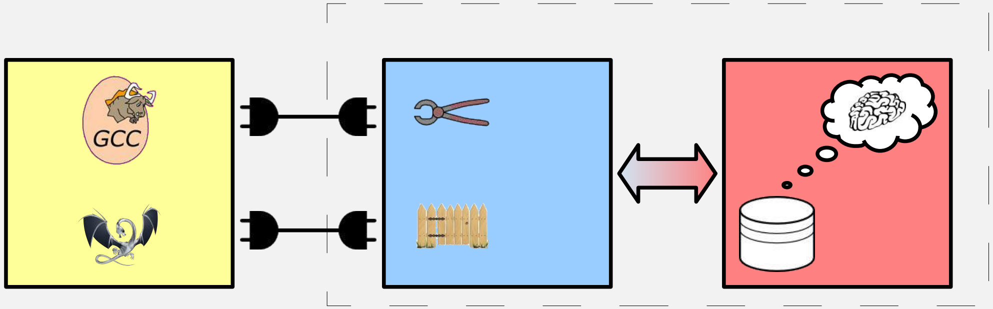
Overall Design



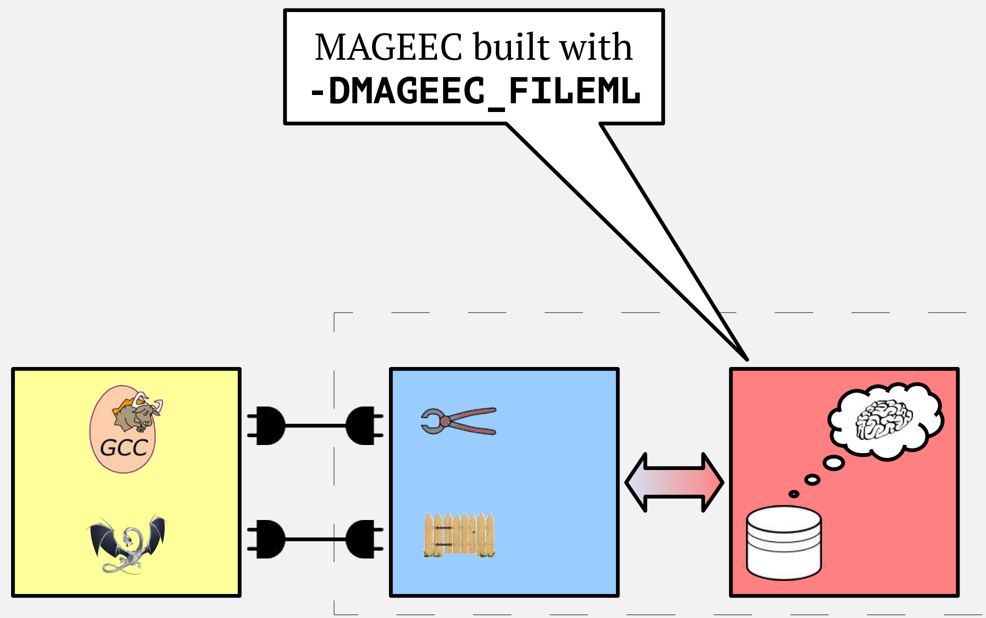
Overall Design



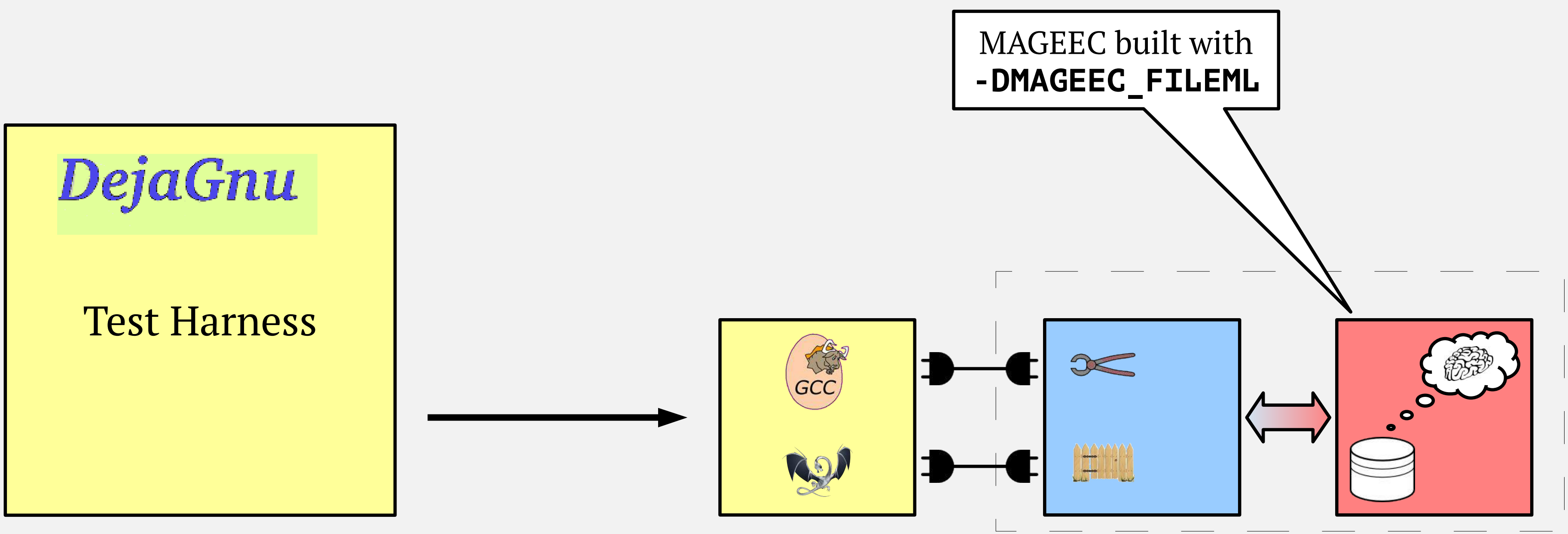
Building the Database



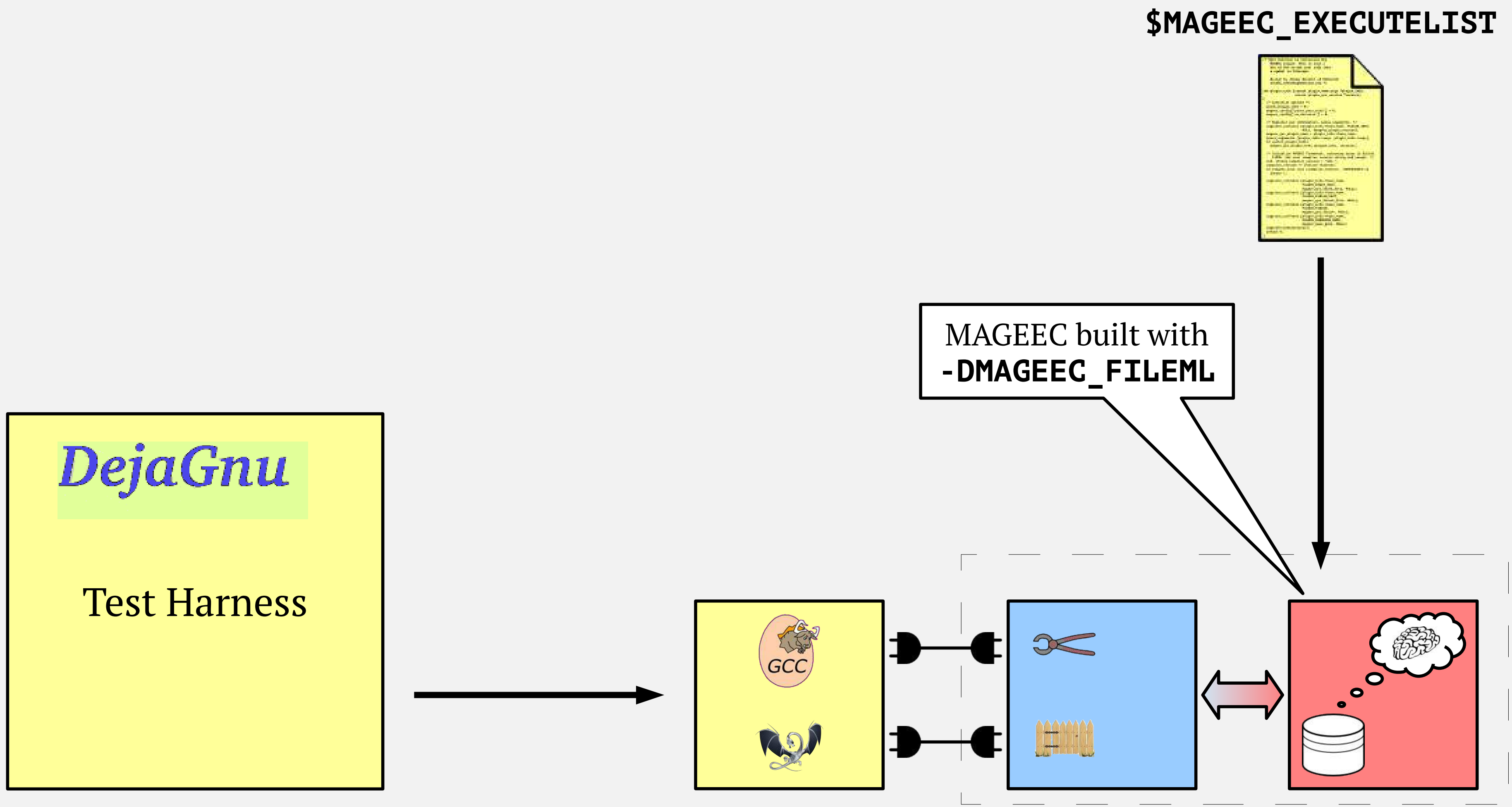
Building the Database



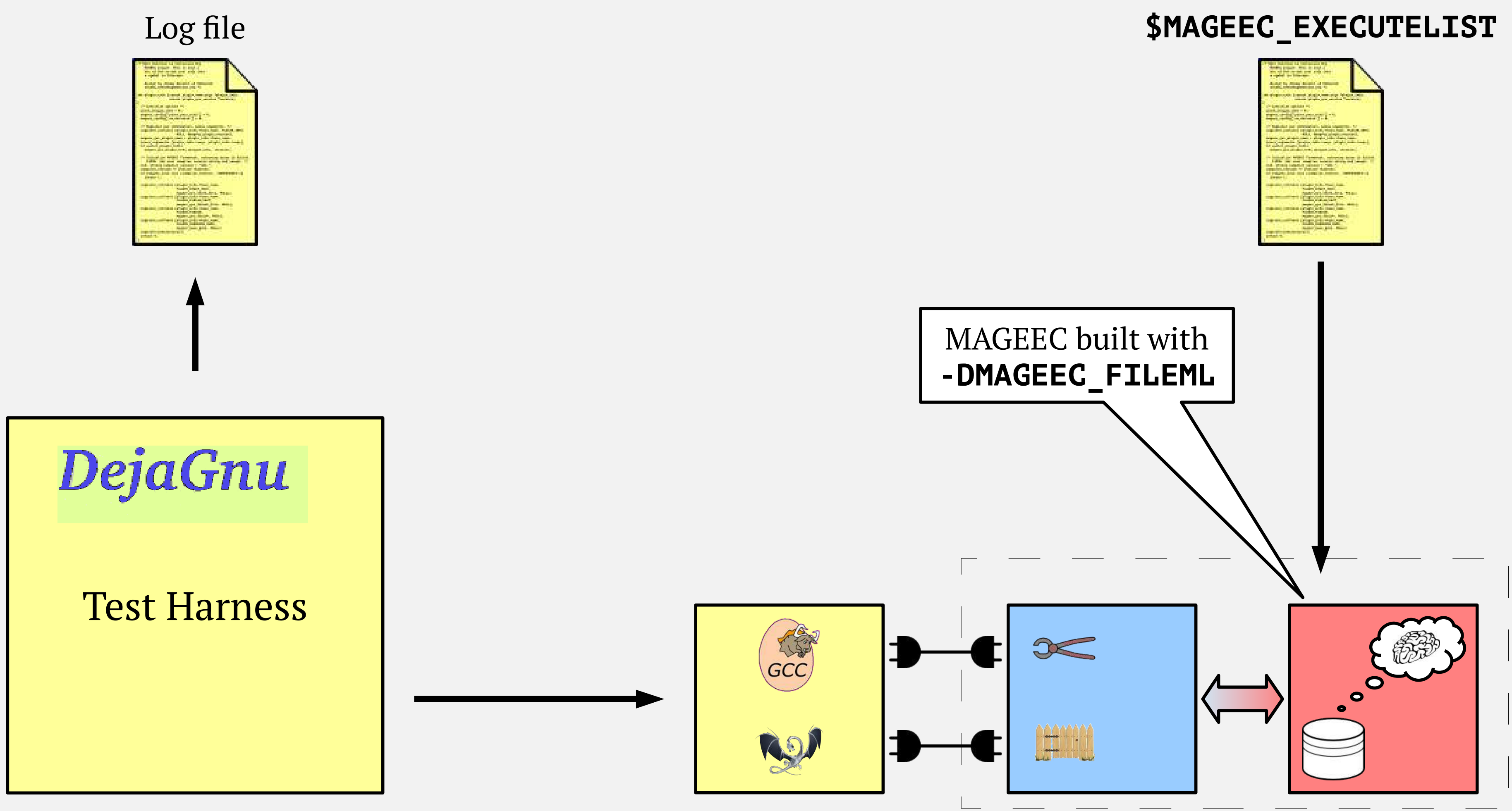
Building the Database



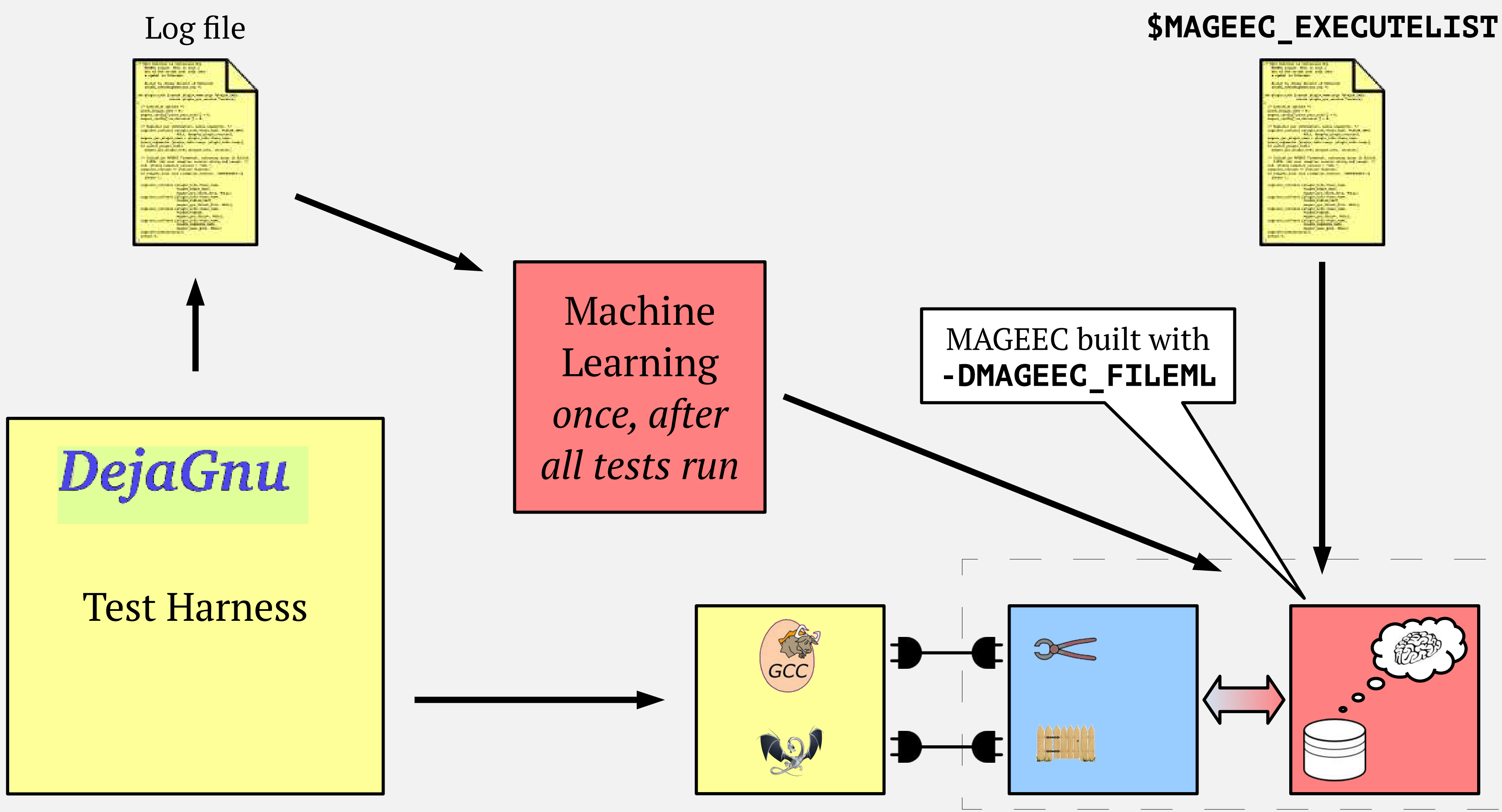
Building the Database



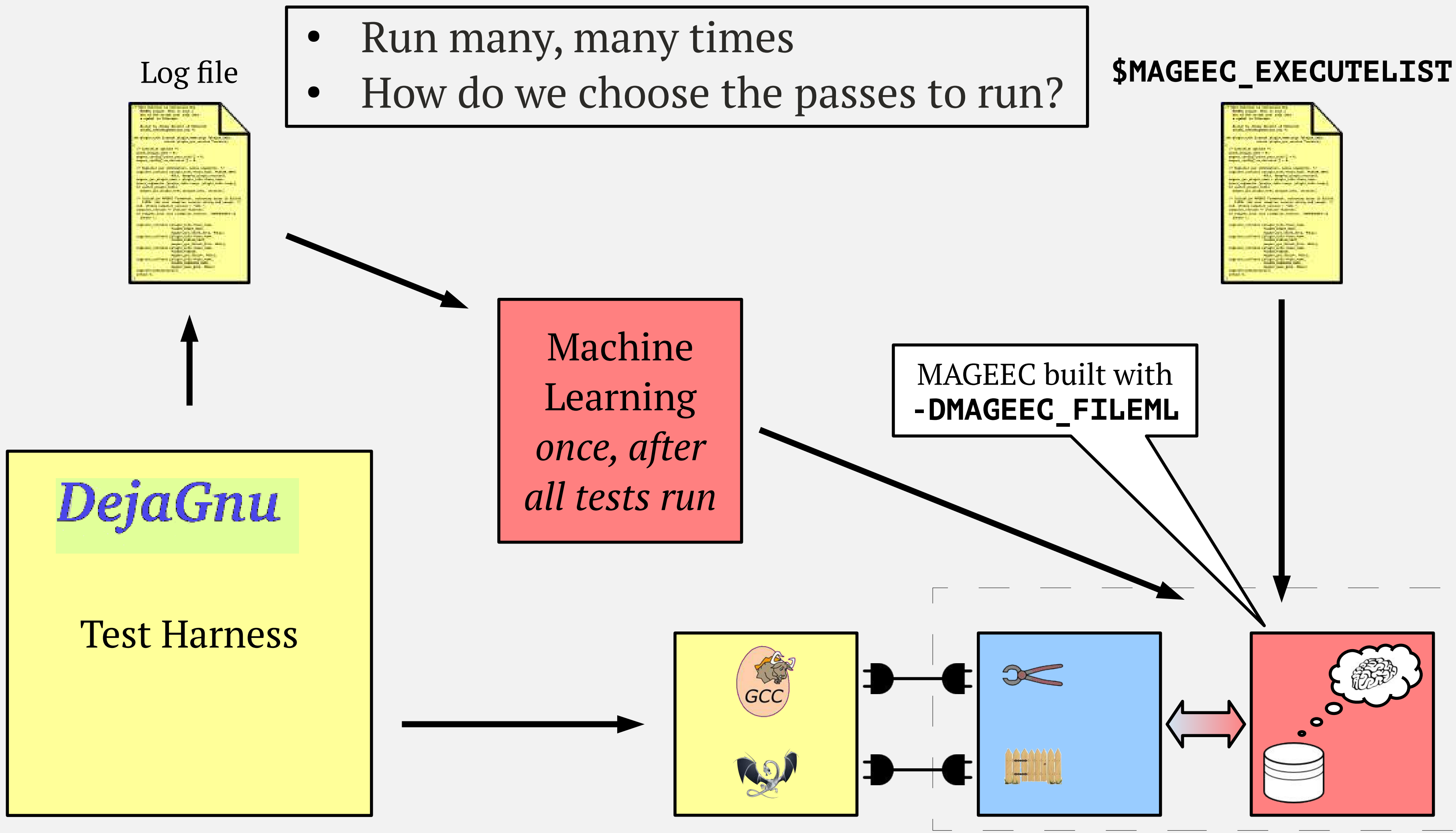
Building the Database



Building the Database



Building the Database



Full Factorial Design

- From all combinations, we can find the impact of one option.

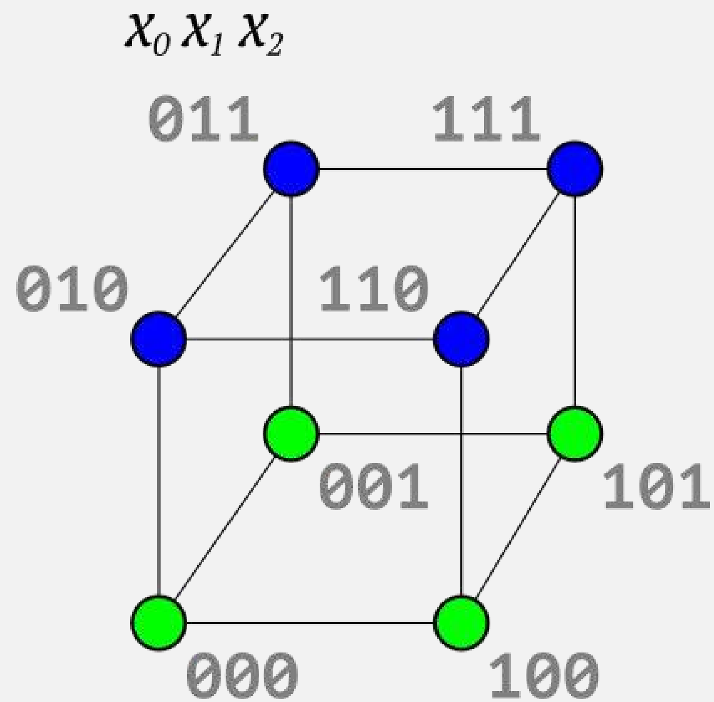
Full Factorial Design

- From all combinations, we can find the impact of one option.
  - example with three options, x_0, x_1 and x_2.

Full Factorial Design

- From all combinations, we can find the impact of one option.
  - example with three options, x_0, x_1 and x_2.

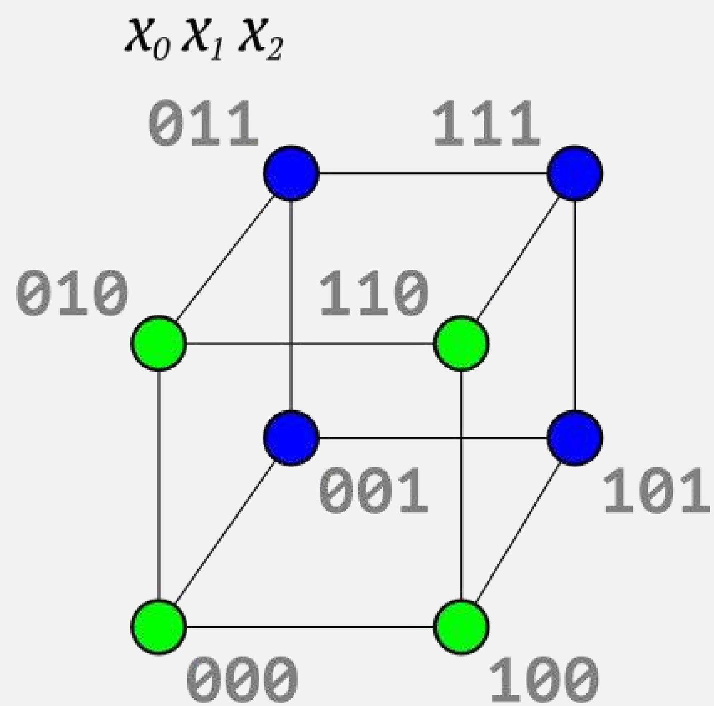
$$x_1 = \frac{\sum \textcolor{blue}{\bullet}}{4} - \frac{\sum \textcolor{green}{\bullet}}{4}$$



Full Factorial Design

- The same data give us the other options as well

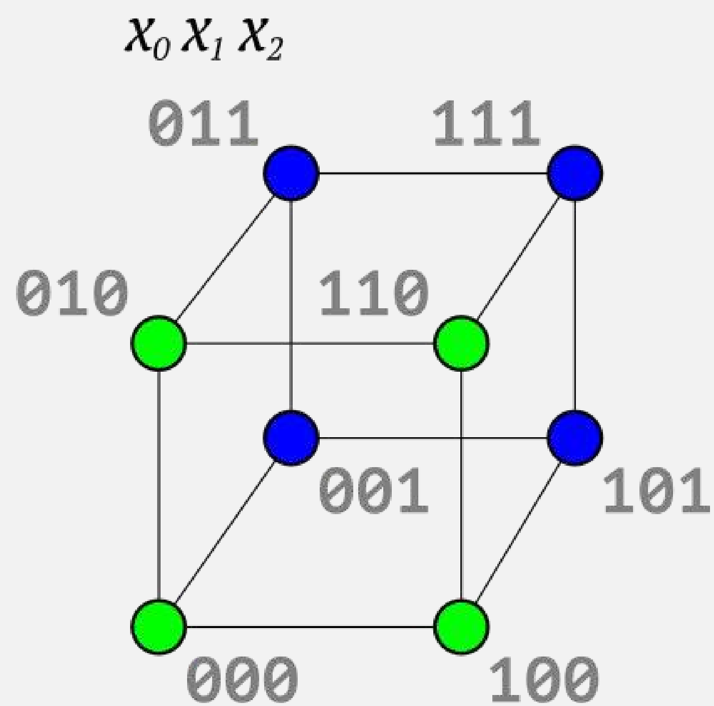
$$x_2 = \frac{\sum {\color{blue}\bullet}}{4} - \frac{\sum {\color{green}\bullet}}{4}$$



Full Factorial Design

- The same data give us the other options as well

$$x_2 = \frac{\Sigma \bullet}{4} - \frac{\Sigma \bullet}{4}$$

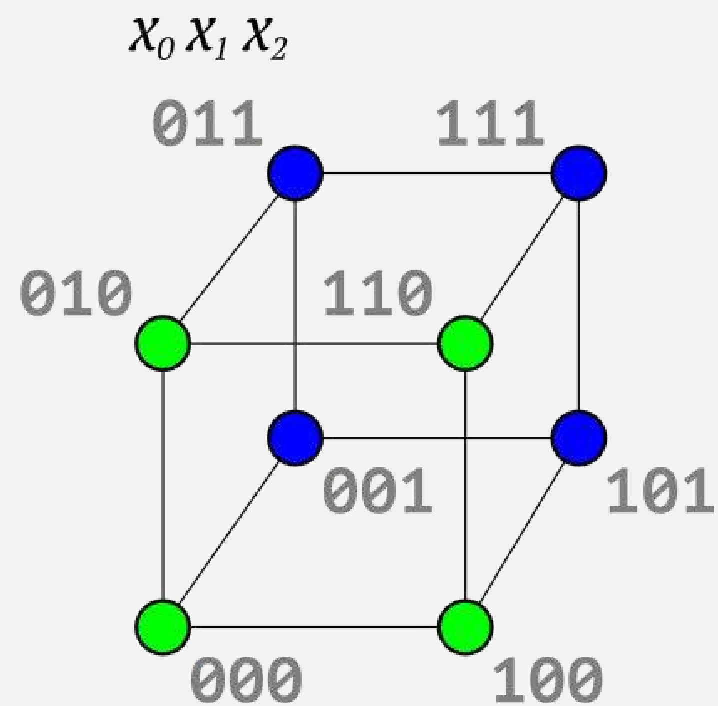


- We need a total of 8 runs

Full Factorial Design

- The same data give us the other options as well

$$x_2 = \frac{\sum \bullet}{4} - \frac{\sum \bullet}{4}$$



- We need a total of 8 runs
  - but what if we had 250 options?

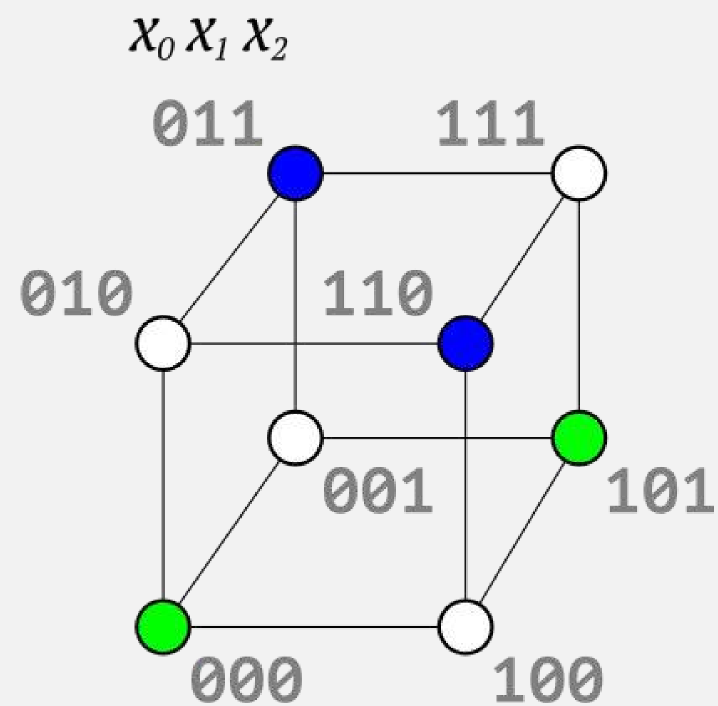
- From a subset, we can find the impact of one option.

- From a subset, we can find the impact of one option.
  - same example with three options, x_0, x_1 and x_2.

Fractional Factorial Design

- From a subset, we can find the impact of one option.
  - same example with three options, x_0, x_1 and x_2.

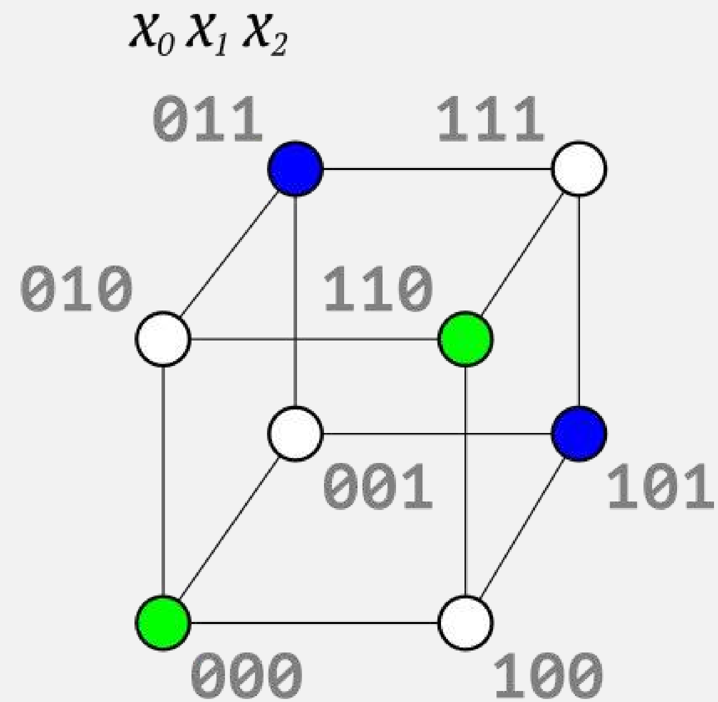
$$x_1 = \frac{\sum {\color{blue}\bullet}}{2} - \frac{\sum {\color{green}\bullet}}{2}$$



Fractional Factorial Design

- The same data give us all the options.
  - by choosing a different combination of data points

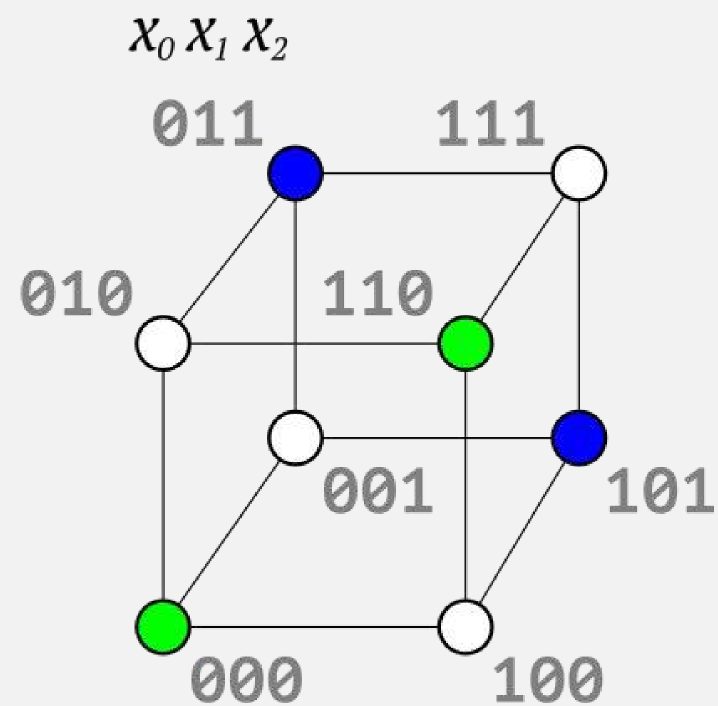
$$x_2 = \frac{\sum \bullet}{2} - \frac{\sum \bullet}{2}$$



Fractional Factorial Design

- The same data give us all the options.
  - by choosing a different combination of data points

$$x_2 = \frac{\Sigma {\color{blue}\bullet}}{2} - \frac{\Sigma {\color{green}\bullet}}{2}$$

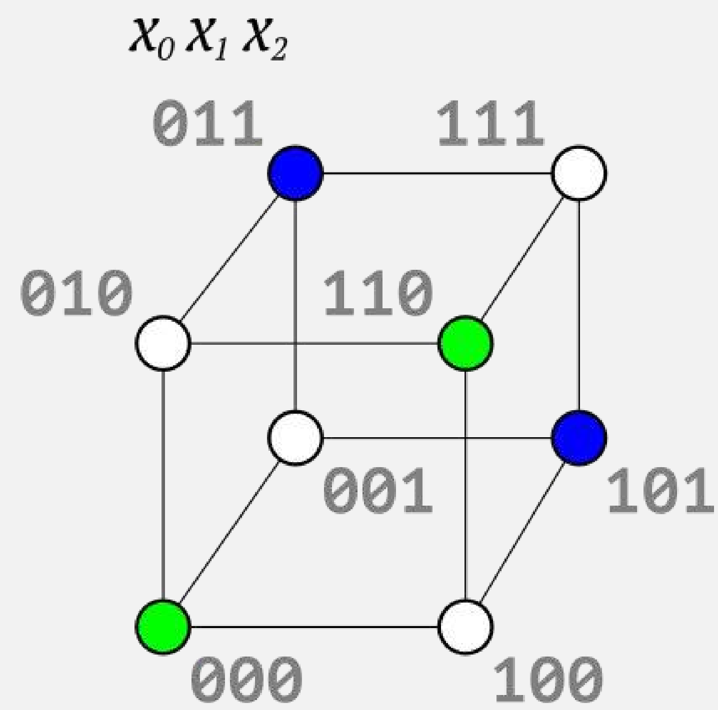


- We need a total of 4 runs

Fractional Factorial Design

- The same data give us all the options.
  - by choosing a different combination of data points

$$x_2 = \frac{\Sigma \bullet}{2} - \frac{\Sigma \bullet}{2}$$

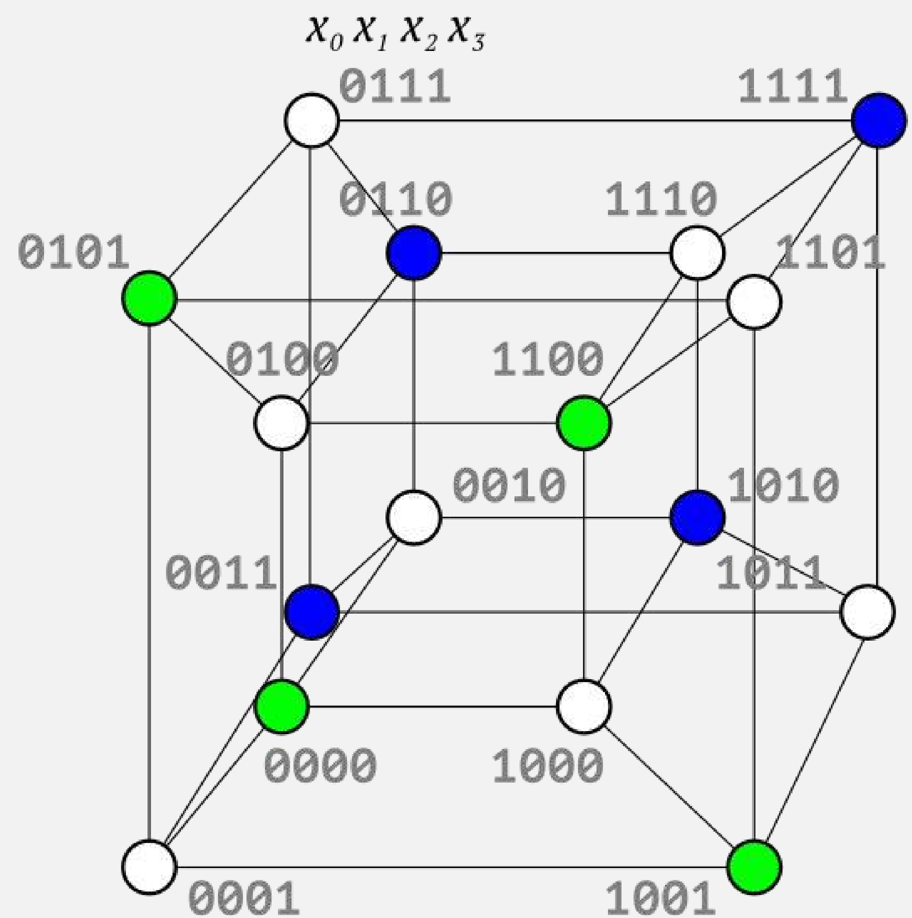


- We need a total of 4 runs
  - but it could be x_0 and x_1 acting together

More Factors

- Gains are more significant with more factors

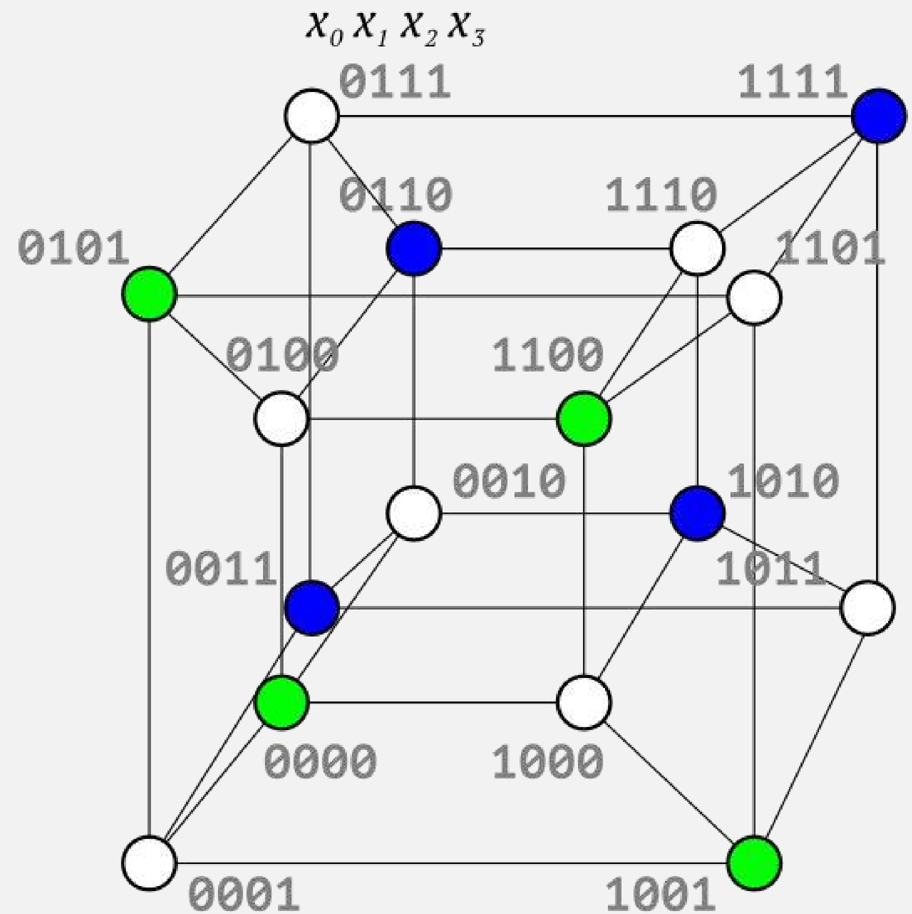
$$x_2 = \frac{\Sigma \textcolor{blue}{\bullet}}{4} - \frac{\Sigma \textcolor{green}{\bullet}}{4}$$



More Factors

- Gains are more significant with more factors
  - deal with multiple factor interaction

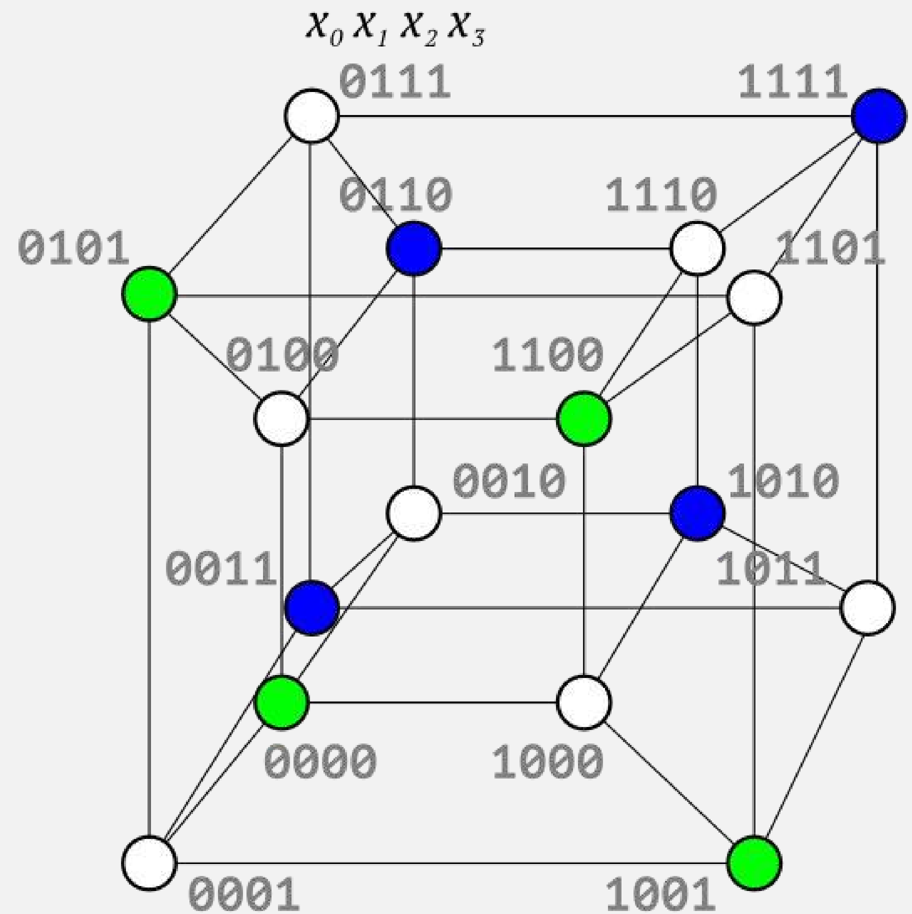
$$x_2 = \frac{\Sigma \bullet}{4} - \frac{\Sigma \bullet}{4}$$



More Factors

- Gains are more significant with more factors
  - deal with multiple factor interaction
  - challenge is tools

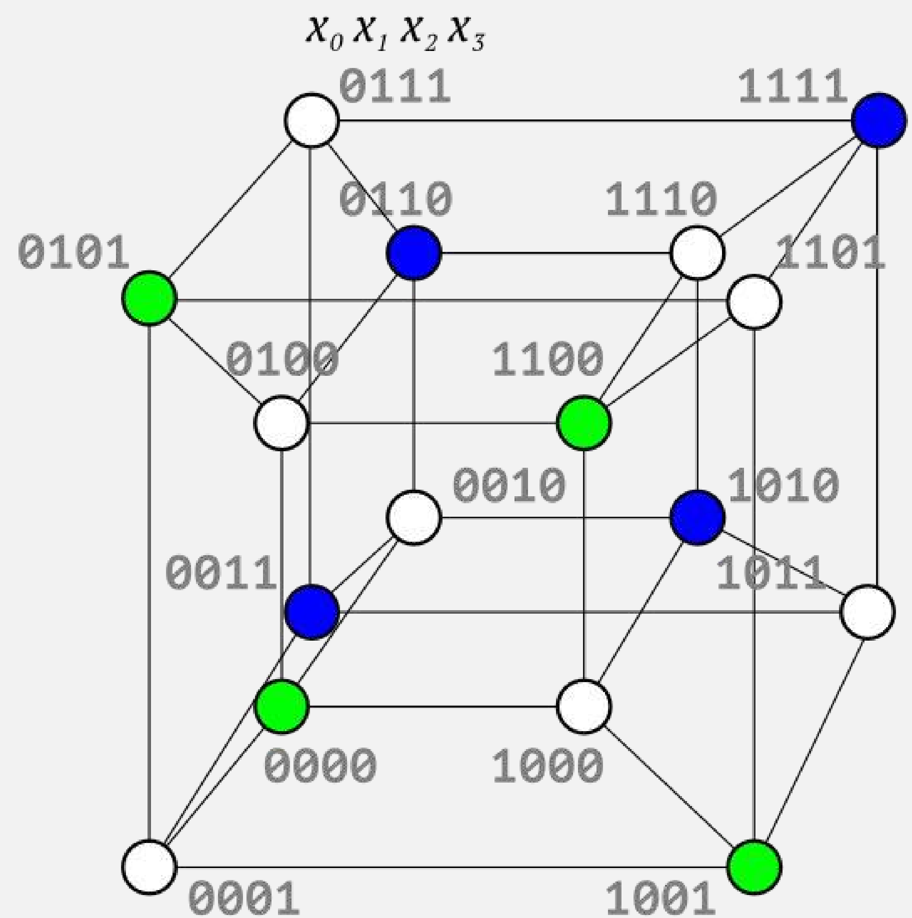
$$x_2 = \frac{\sum \textcolor{blue}{\bullet}}{4} - \frac{\sum \textcolor{green}{\bullet}}{4}$$



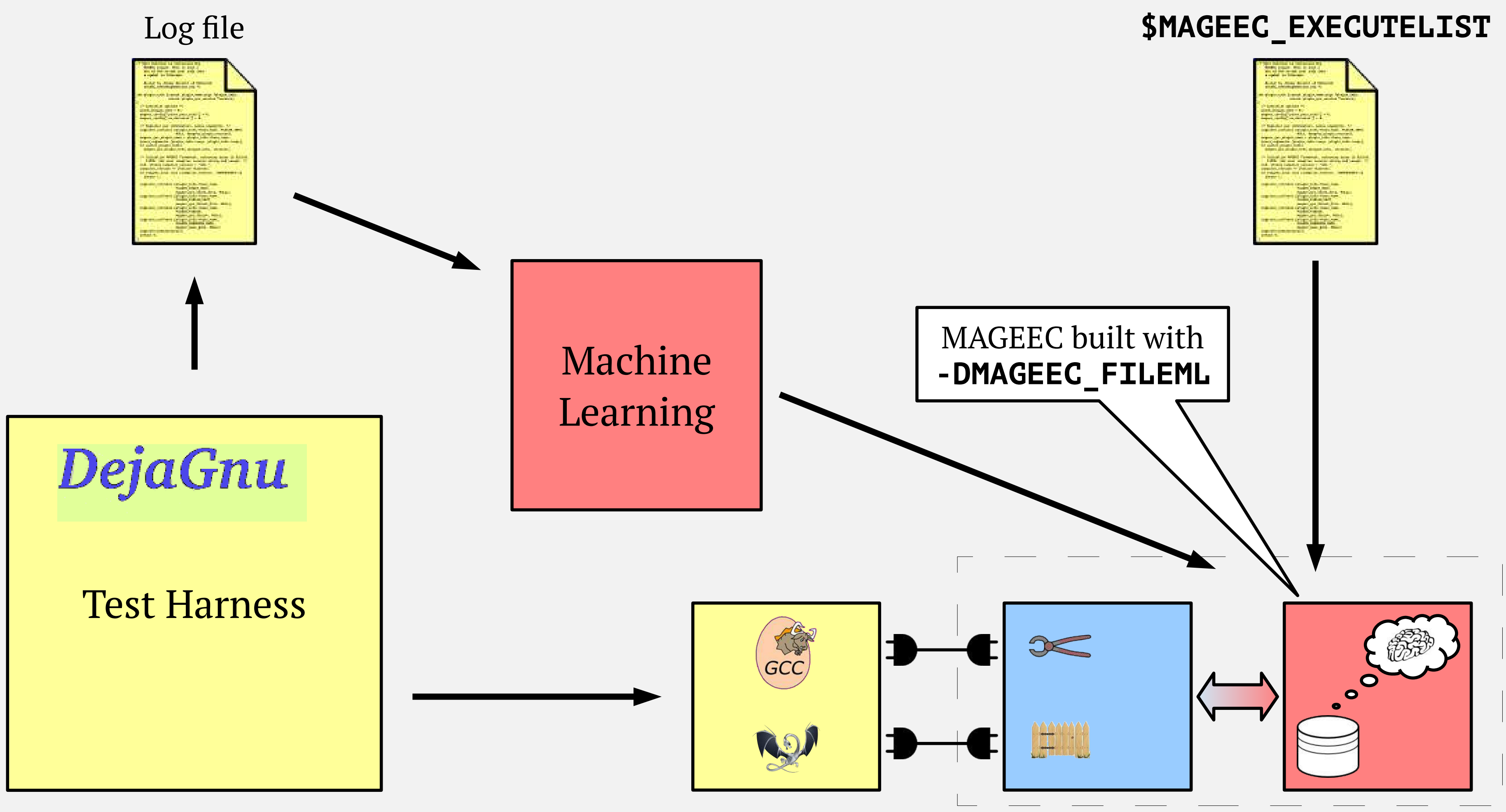
More Factors

- Gains are more significant with more factors
  - deal with multiple factor interaction
  - challenge is tools
  - current limit is 120 factors

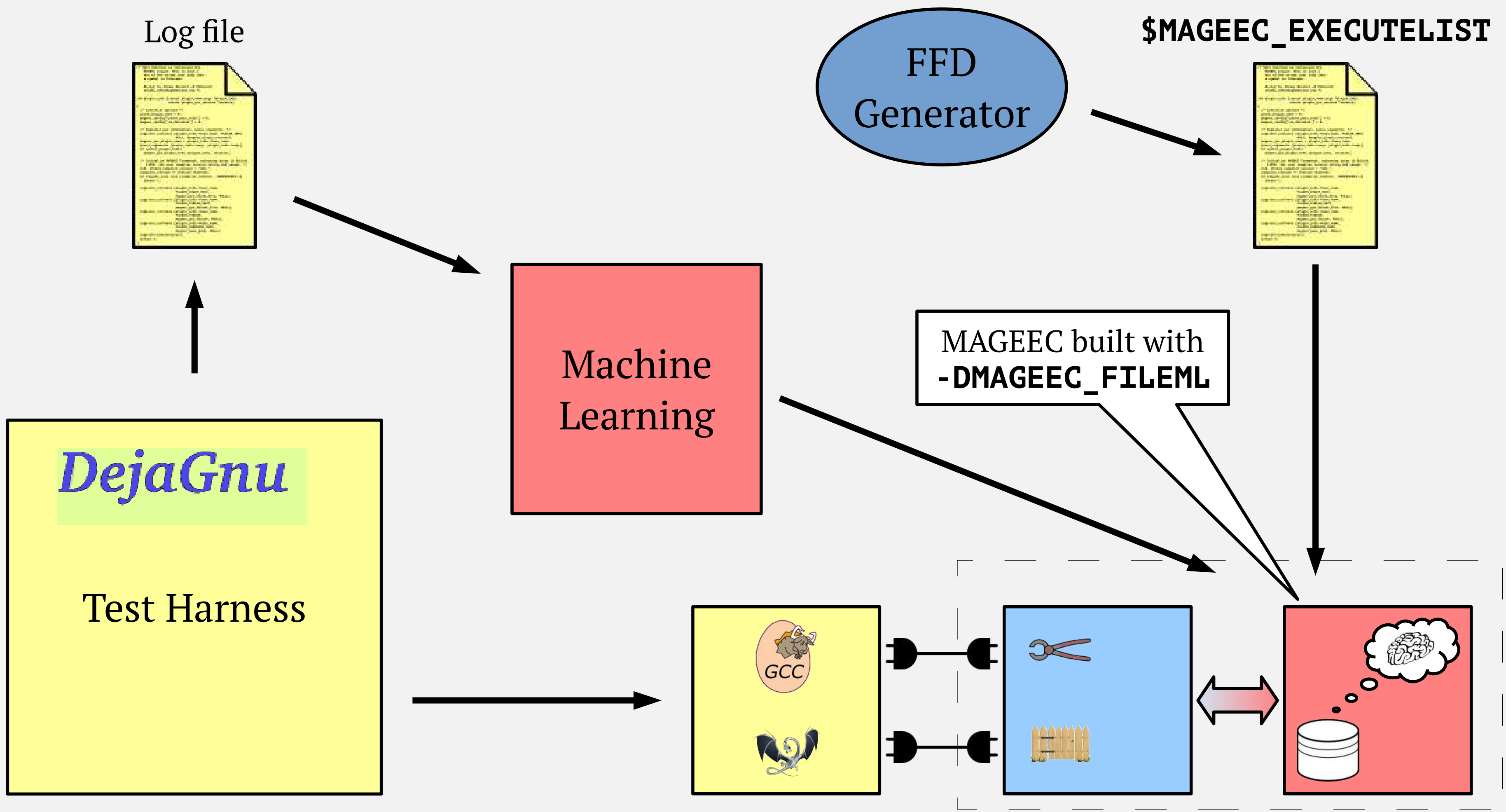
$$x_2 = \frac{\sum \bullet}{4} - \frac{\sum \bullet}{4}$$



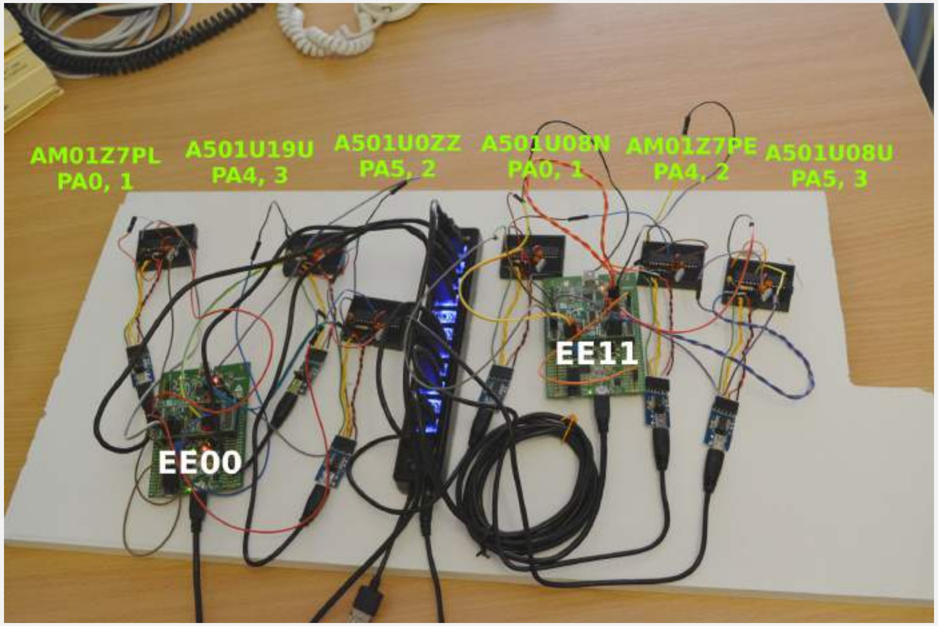
Building the Database



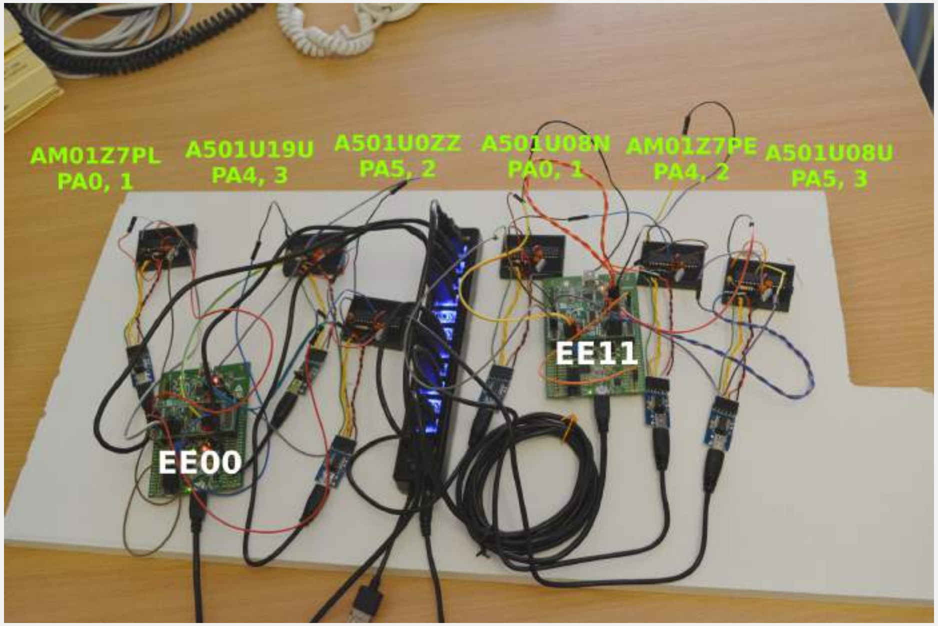
Building the Database



Scalability

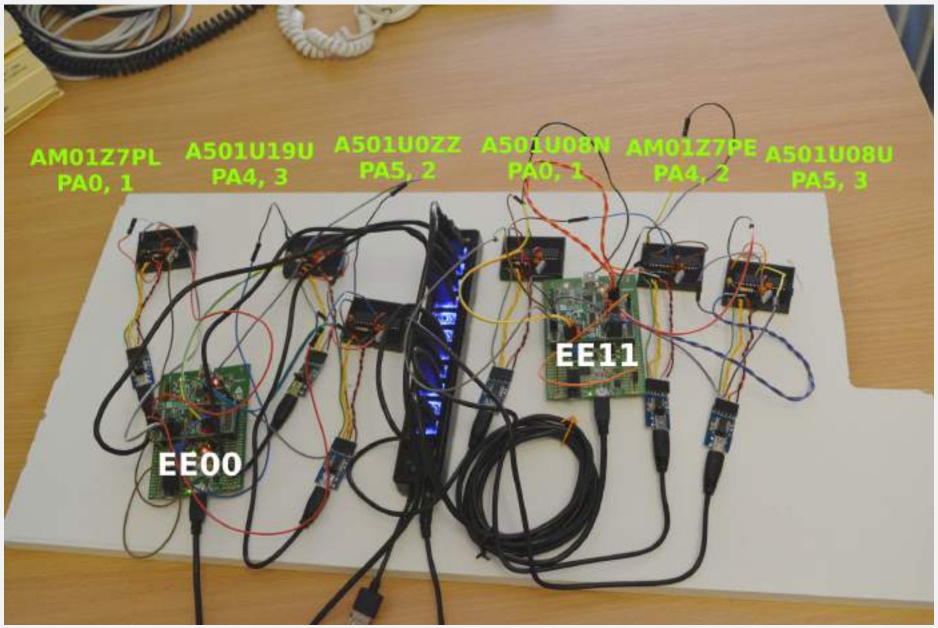


Scalability



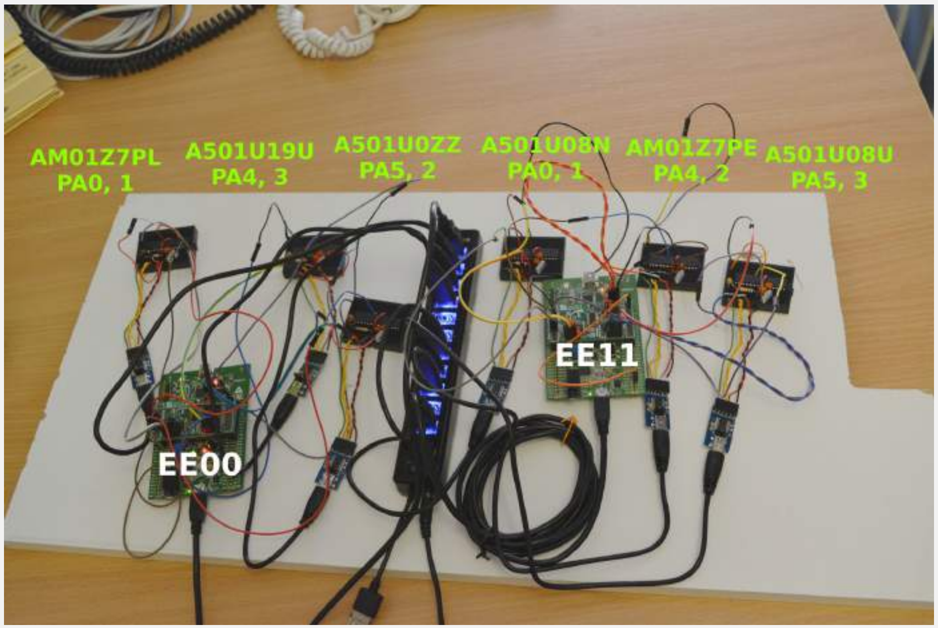
- Each AVR test takes 4s
  - 2s to flash device
  - 2s to run test

Scalability



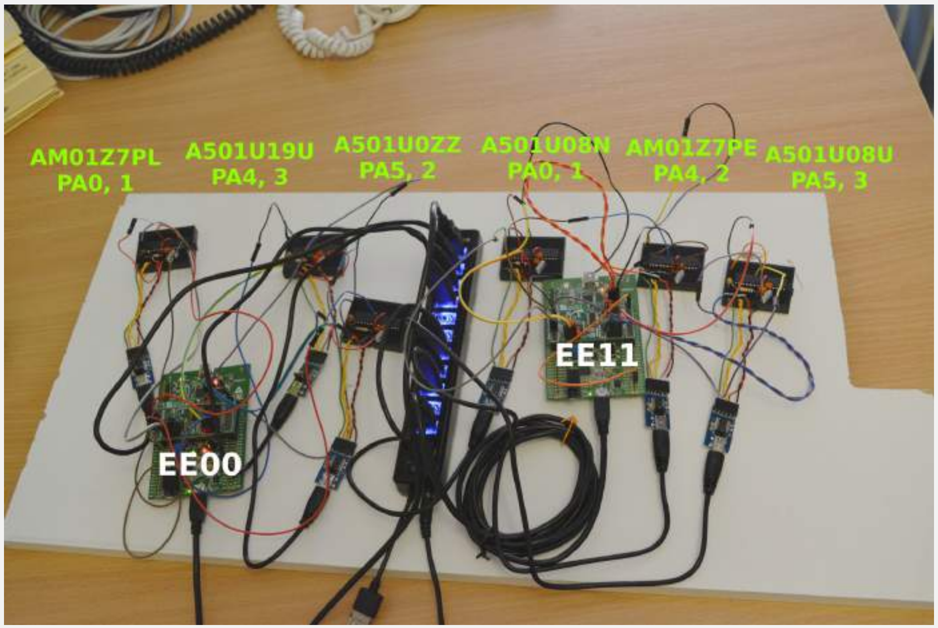
- Each AVR test takes 4s
  - 2s to flash device
  - 2s to run test
- Approx 100 BEEBS tests
  - run 6 boards at once
  - 67s per test run

Scalability



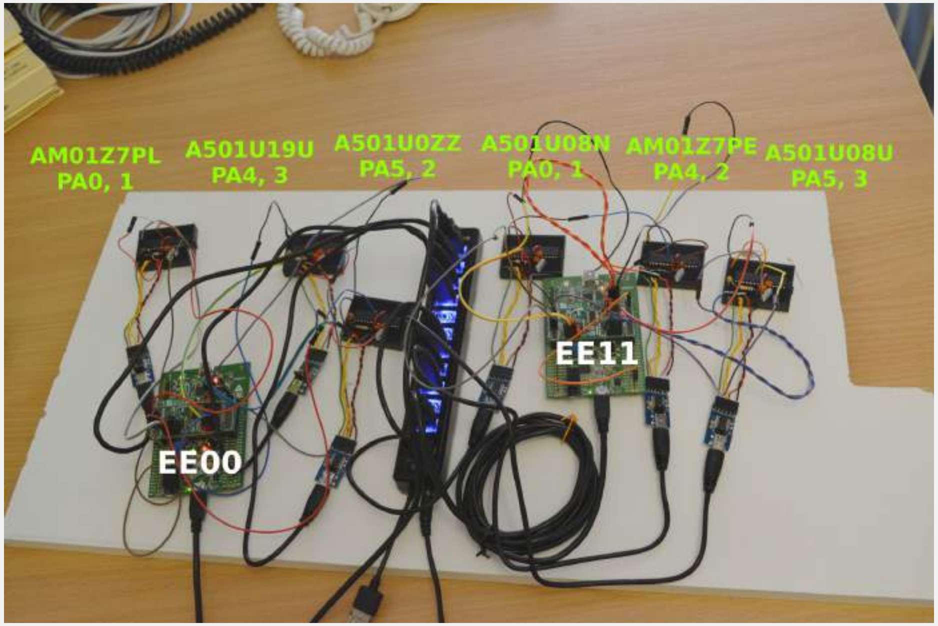
- Each AVR test takes 4s
  - 2s to flash device
  - 2s to run test
- Approx 100 BEEBS tests
  - run 6 boards at once
  - 67s per test run
- 200+ optimization passes
  - 2^200 possibilities

Scalability



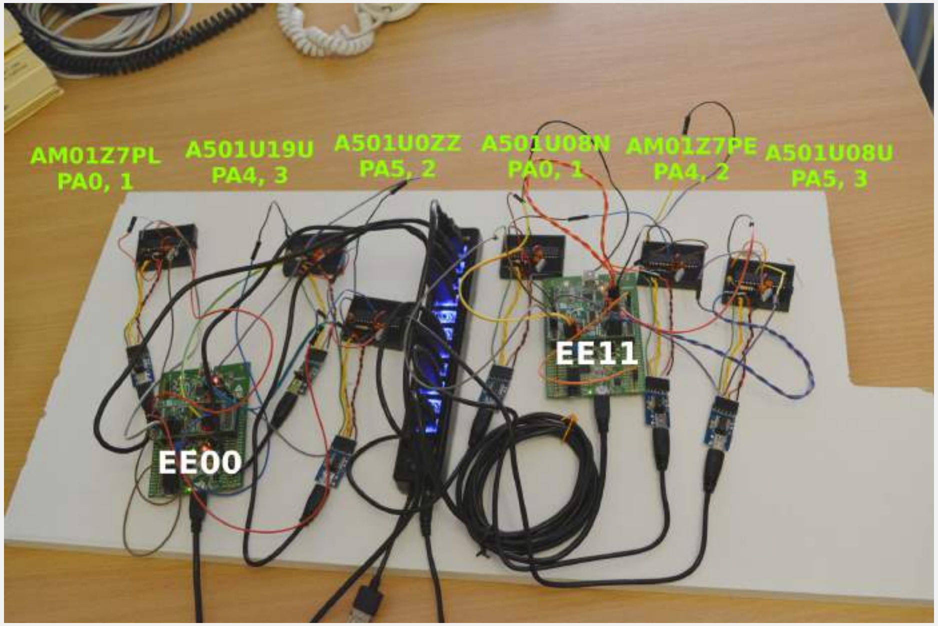
- Each AVR test takes 4s
  - 2s to flash device
  - 2s to run test
- Approx 100 BEEBS tests
  - run 6 boards at once
  - 67s per test run
- 200+ optimization passes
  - 2^200 possibilities
  - FFD reduces this

Scalability



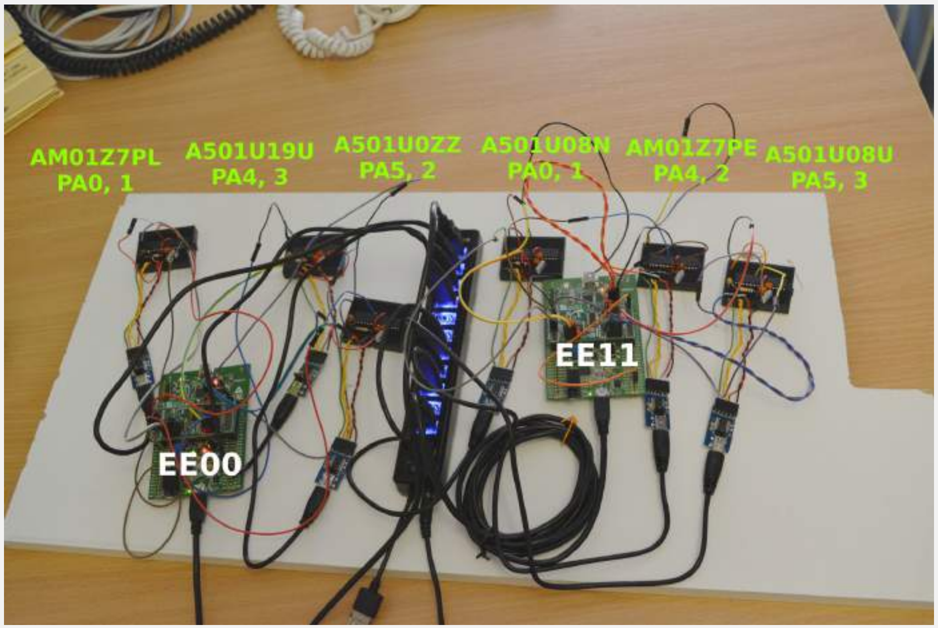
- Each AVR test takes 4s
  - 2s to flash device
  - 2s to run test
- Approx 100 BEEBS tests
  - run 6 boards at once
  - 67s per test run
- 200+ optimization passes
  - 2^200 possibilities
  - FFD reduces this
  - to 2^16 = 65,536 runs

Scalability



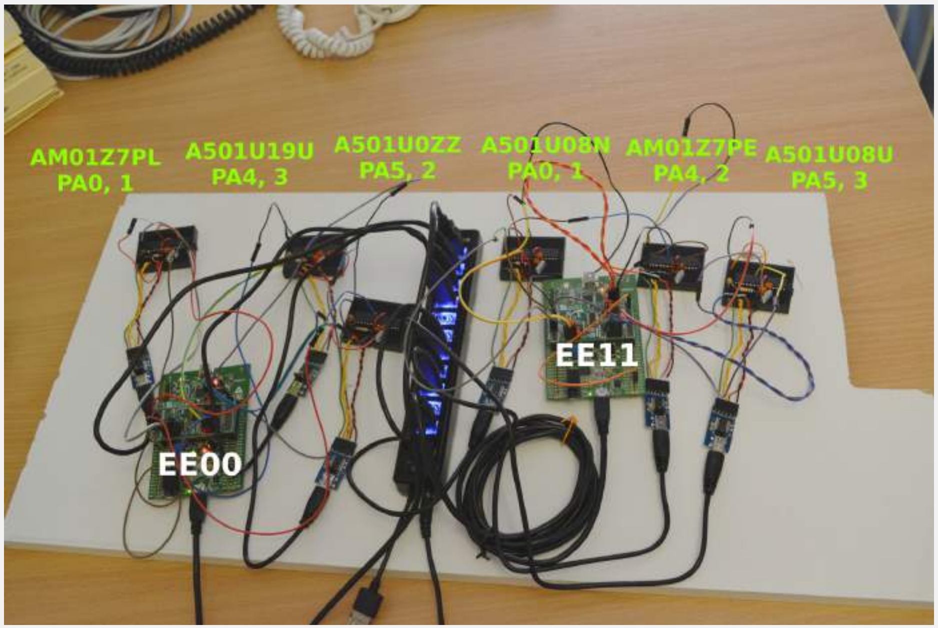
- Each AVR test takes 4s
  - 2s to flash device
  - 2s to run test
- Approx 100 BEEBS tests
  - run 6 boards at once
  - 67s per test run
- 200+ optimization passes
  - 2^200 possibilities
  - FFD reduces this
  - to 2^16 = 65,536 runs
  - = 4,369,067s

Scalability



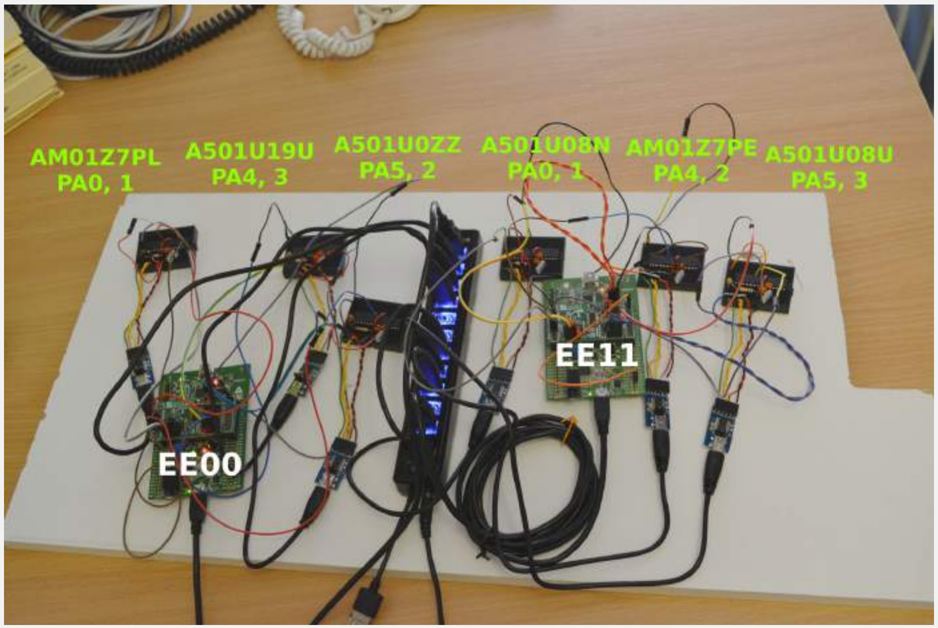
- Each AVR test takes 4s
  - 2s to flash device
  - 2s to run test
- Approx 100 BEEBS tests
  - run 6 boards at once
  - 67s per test run
- 200+ optimization passes
  - 2^200 possibilities
  - FFD reduces this
  - to 2^16 = 65,536 runs
  - = 4,369,067s
  - = 50d 13h 37m 47s

Scalability



- Each AVR test takes 4s
  - 2s to flash device
  - 2s to run test
- Approx 100 BEEBS tests
  - run 6 boards at once
  - 67s per test run
- 200+ optimization passes
  - 2^200 possibilities
  - FFD reduces this
  - to 2^16 = 65,536 runs
  - = 4,369,067s
  - = 50d 13h 37m 47s
- Oh dear

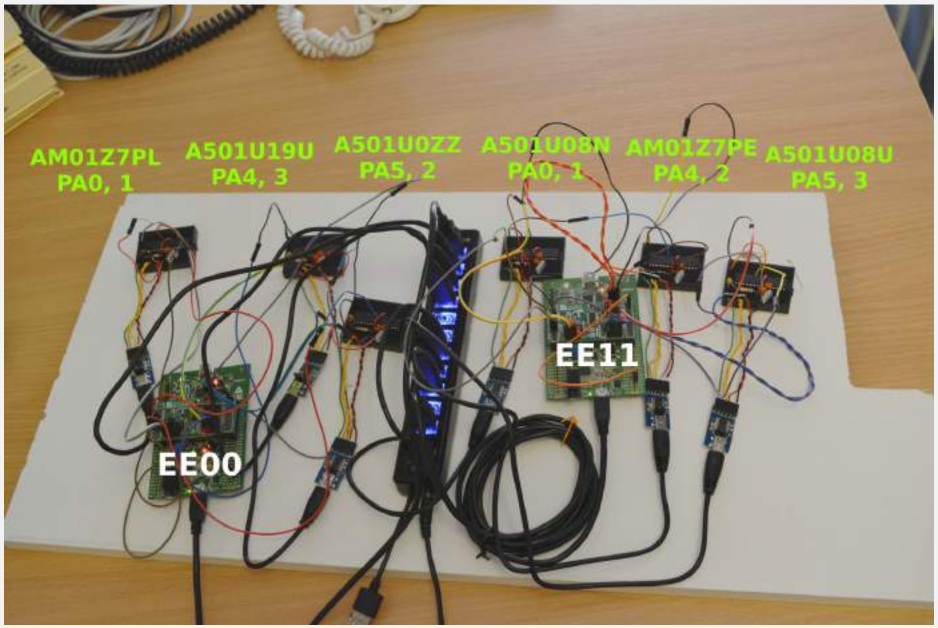
Scalability



One compiler on one CPU

- Each AVR test takes 4s
  - 2s to flash device
  - 2s to run test
- Approx 100 BEEBS tests
  - run 6 boards at once
  - 67s per test run
- 200+ optimization passes
  - 2^200 possibilities
  - FFD reduces this
  - to 2^16 = 65,536 runs
  - = 4,369,067s
  - = 50d 13h 37m 47s
- Oh dear ☹

Scalability



One compiler on one CPU

Atmel have 200+ AVR variants

- Each AVR test takes 4s
  - 2s to flash device
  - 2s to run test
- Approx 100 BEEBS tests
  - run 6 boards at once
  - 67s per test run
- 200+ optimization passes
  - 2^200 possibilities
  - FFD reduces this
  - to 2^16 = 65,536 runs
  - = 4,369,067s
  - = 50d 13h 37m 47s
- Oh dear ☹

Placket Burman to the Rescue

- A special case of FFD

Placket Burman to the Rescue

- A special case of FFD
- One more run than the number of factors (passes).

Placket Burman to the Rescue

- A special case of FFD
- One more run than the number of factors (passes).
- Assumes independence of factors.

Placket Burman to the Rescue

- A special case of FFD
- One more run than the number of factors (passes).
- Assumes independence of factors.
- 210 optimization passes means 211 test runs
  - 23h 27m on one board.

Placket Burman to the Rescue

- A special case of FFD
- One more run than the number of factors (passes).
- Assumes independence of factors.
- 210 optimization passes means 211 test runs
  - 23h 27m on one board.
- Can then use FFD on most important passes

Placket Burman to the Rescue

- A special case of FFD
- One more run than the number of factors (passes).
- Assumes independence of factors.
- 210 optimization passes means 211 test runs
  - 23h 27m on one board.
- Can then use FFD on most important passes

Run	X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9	X_{10}	X_{11}
1	+	+	+	+	+	+	+	+	+	+	+
2	−	+	−	+	+	+	−	−	−	+	−
3	−	−	+	−	+	+	+	−	−	−	+
4	+	−	−	+	−	+	+	+	−	−	−
5	−	+	−	−	+	−	+	+	+	−	−
6	−	−	+	−	−	+	−	+	+	+	−
7	−	−	−	+	−	−	+	−	+	+	+
8	+	−	−	−	+	−	−	+	−	+	+
9	+	+	−	−	−	+	−	−	+	−	+
10	+	+	+	−	−	−	+	−	−	+	−
11	−	+	+	+	−	−	−	+	−	−	+
12	+	−	+	+	+	−	−	−	+	−	−

Superoptimization

Superoptimization is an old technique

- Henry Massalin. *Superoptimizer—A look at the Smallest Program*. ASPLOS-II, 1987.

Context

Superoptimization is an old technique

- Henry Massalin. *Superoptimizer—A look at the Smallest Program*. ASPLOS-II, 1987.

There are free and open source implementations

- A Hacker's Assistant (Aha)
- the GNU Superoptimizer (GSO)
- all have limitations

Context

Superoptimization is an old technique

- Henry Massalin. *Superoptimizer—A look at the Smallest Program*. ASPLOS-II, 1987.

There are free and open source implementations

- A Hacker's Assistant (Aha)
- the GNU Superoptimizer (GSO)
- all have limitations

Can we now build a commercially robust tool?

- computers are faster, algorithms have advanced
- what are the areas where this can be applied?

Superoptimization in Action

```
int sign (int n)
{
   if (n > 0)
      return 1;
   else if (n < 0)
      return -1;
   else
      return 0;
}
```

Superoptimization in Action

```c
int sign (int n)
{
    if (n > 0)
        return 1;
    else if (n < 0)
        return -1;
    else
        return 0;
}
```

```
    cmp.l  d0, 0
    ble    L1
    move.l d1, 1
    bra    L3
L1:
    bge    L2
    move.l d1, -1
    bra    L3
L2:
    move.l d1, 0
L3:
```

Superoptimization in Action

```c
int sign (int n)
{
   if (n > 0)
      return 1;
   else if (n < 0)
      return -1;
   else
      return 0;
}
```

```
    cmp.l  d0, 0
    ble    L1
    move.l d1, 1
    bra    L3
L1:
    bge    L2
    move.l d1, -1
    bra    L3
L2:
    move.l d1, 0
L3:
```

```
add.l  d0, d0
subx.l d1, d1
negx.l d0
addx.l d1, d1
```

How Does it Work?

```
d0  ← n

add.l  d0, d0
subx.l d1, d1
negx.l d0
addx.l d1, d1

d1  → sign(n)
```

How Does it Work?

```
d0  ← n

add.l  d0, d0
subx.l d1, d1
negx.l d0
addx.l d1, d1

d1  → sign(n)
```

x	d0	d1
0	-3	

x	d0	d1
0	0	

x	d0	d1
0	2	

How Does it Work?

```
d0  ← n

add.l  d0, d0
subx.l d1, d1
negx.l d0
addx.l d1, d1

d1  → sign(n)
```

x	d0	d1
0	-3	
1	-6	

x	d0	d1
0	0	
0	0	

x	d0	d1
0	2	
0	4	

How Does it Work?

```
d0  ← n

add.l  d0, d0
subx.l d1, d1
negx.l d0
addx.l d1, d1

d1  → sign(n)
```

x	d0	d1
0	-3	
1	-6	
0	-6	-1

x	d0	d1
0	0	
0	0	
0	0	0

x	d0	d1
0	2	
0	4	
0	4	0

How Does it Work?

```
d0  ← n

add.l  d0, d0
subx.l d1, d1
negx.l d0
addx.l d1, d1

d1  → sign(n)
```

x	d0	d1
0	-3	
1	-6	
0	-6	-1
1	6	-1

x	d0	d1
0	0	
0	0	
0	0	0
0	0	0

x	d0	d1
0	2	
0	4	
0	4	0
1	-4	0

How Does it Work?

	x	d0	d1	x	d0	d1	x	d0	d1
d0 ← n	0	-3		0	0		0	2	
add.l d0, d0	1	-6		0	0		0	4	
subx.l d1, d1	0	-6	-1	0	0	0	0	4	0
negx.l d0	1	6	-1	0	0	0	1	-4	0
addx.l d1, d1	0	6	-1	0	0	0	0	-4	1

d1 → sign(n)

How Does it Work?

	x	d0	d1	x	d0	d1	x	d0	d1
`d0 ← n`	0	-3		0	0		0	2	
`add.l d0, d0`	1	-6		0	0		0	4	
`subx.l d1, d1`	0	-6	-1	0	0	0	0	4	0
`negx.l d0`	1	6	-1	0	0	0	1	-4	0
`addx.l d1, d1`	0	6	-1	0	0	0	0	-4	1
`d1 → sign(n)`			-1			0			1

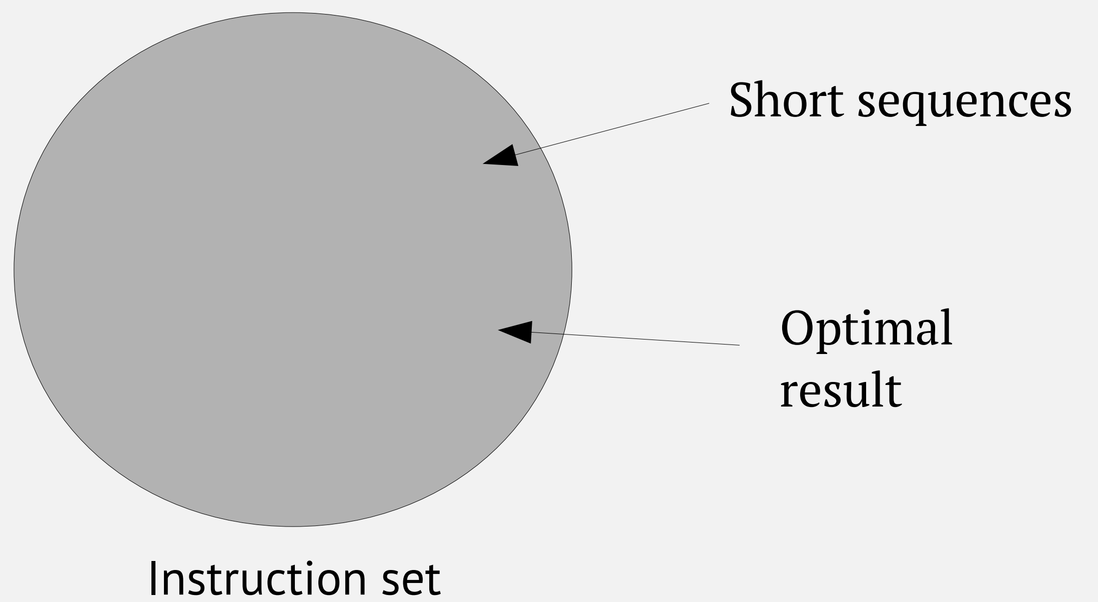
Generating the sequences of instructions

Generating the sequences of instructions

- But doing them *all* takes far too long

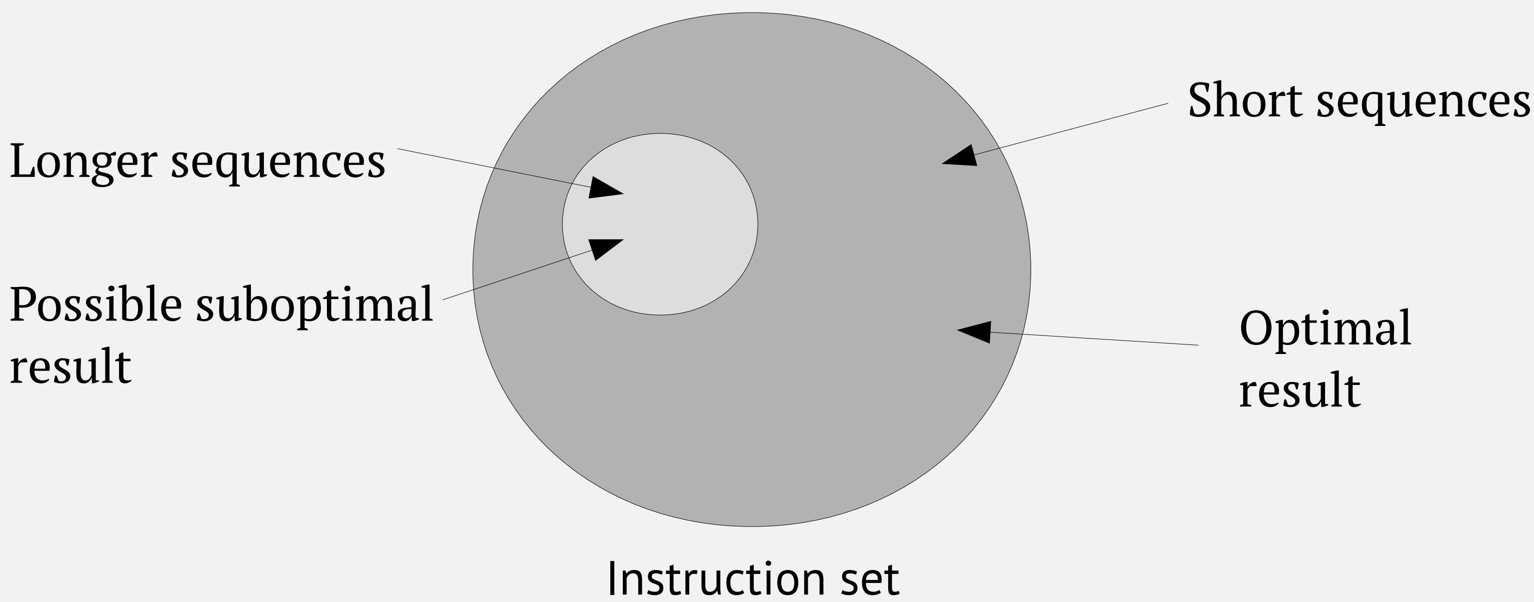
Generating the sequences of instructions

– But doing them *all* takes far too long



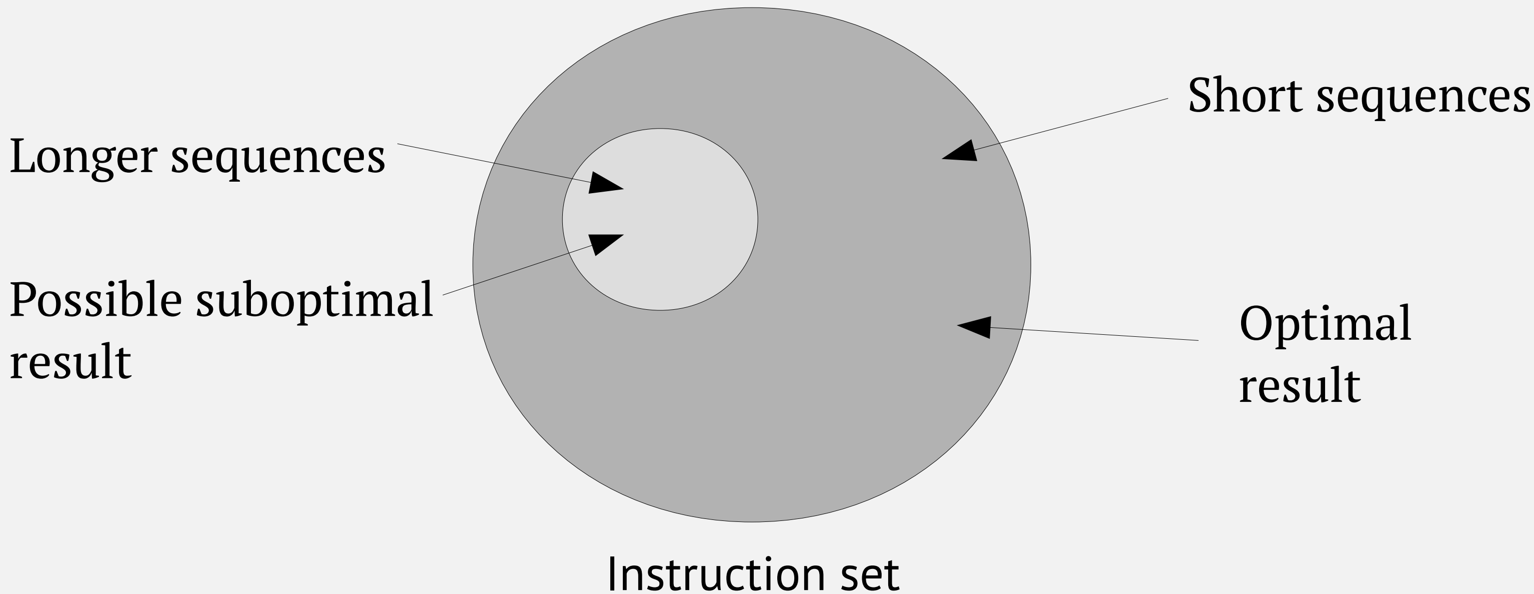
Generating the sequences of instructions

- But doing them *all* takes far too long

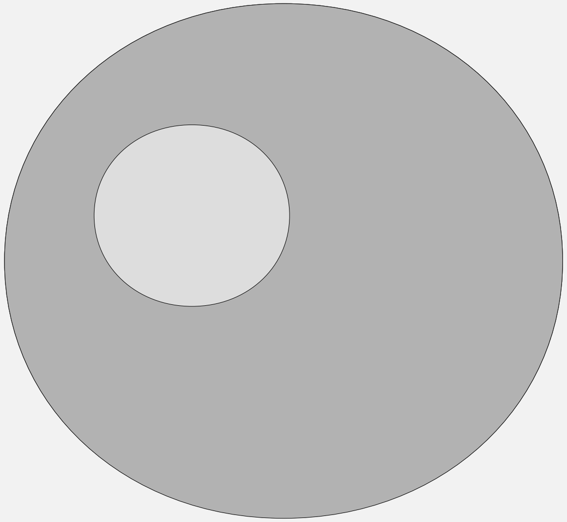


Generating the sequences of instructions

– But doing them *all* takes far too long

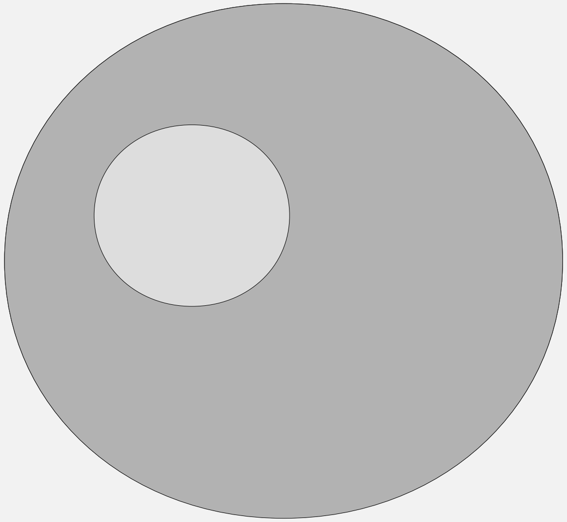


How to select the sequences of instructions?



Instruction set

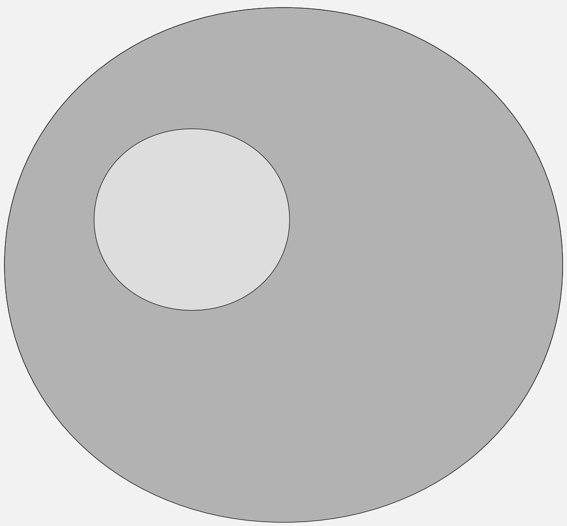
Not all instruction sequences are valid.



Instruction set

Not all instruction sequences are valid.

How do we quickly ignore bad sequences?

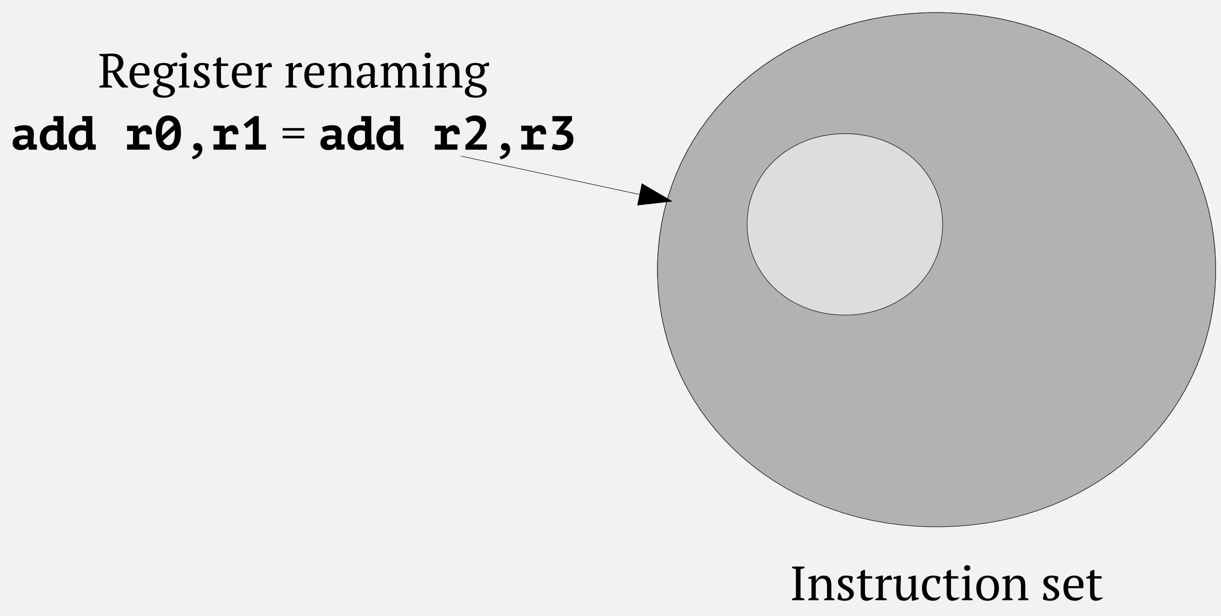


Instruction set

Superoptimization Fundamentals
Pruning

Not all instruction sequences are valid.

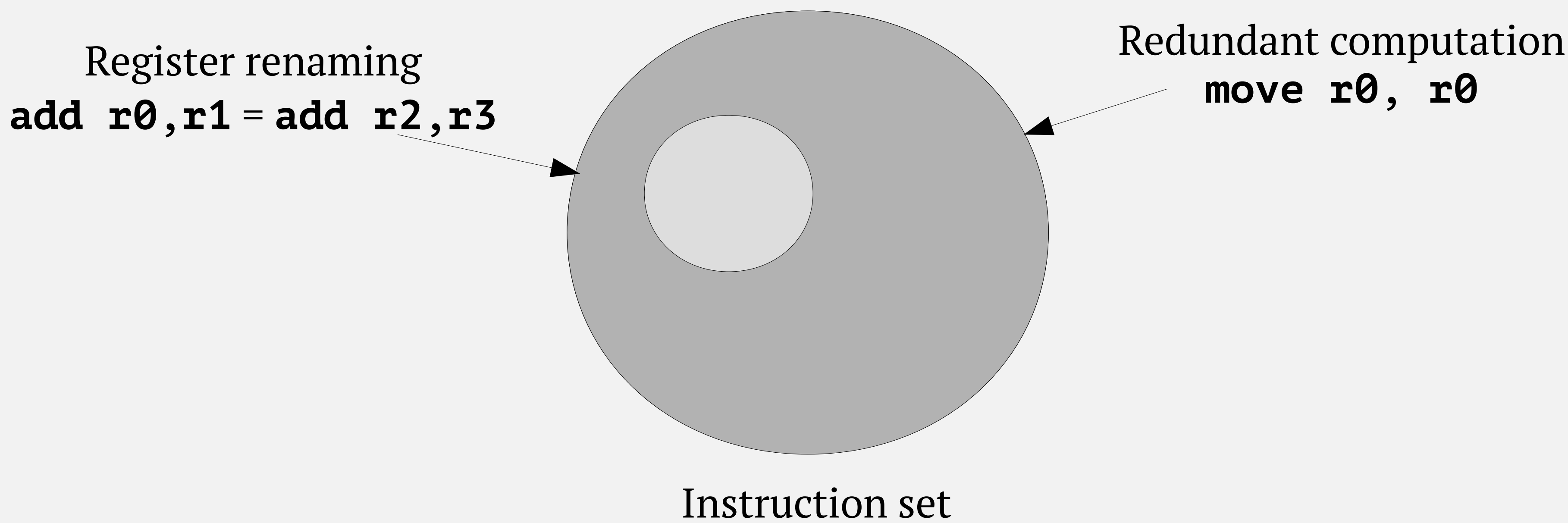
How do we quickly ignore bad sequences?



Superoptimization Fundamentals
Pruning

Not all instruction sequences are valid.

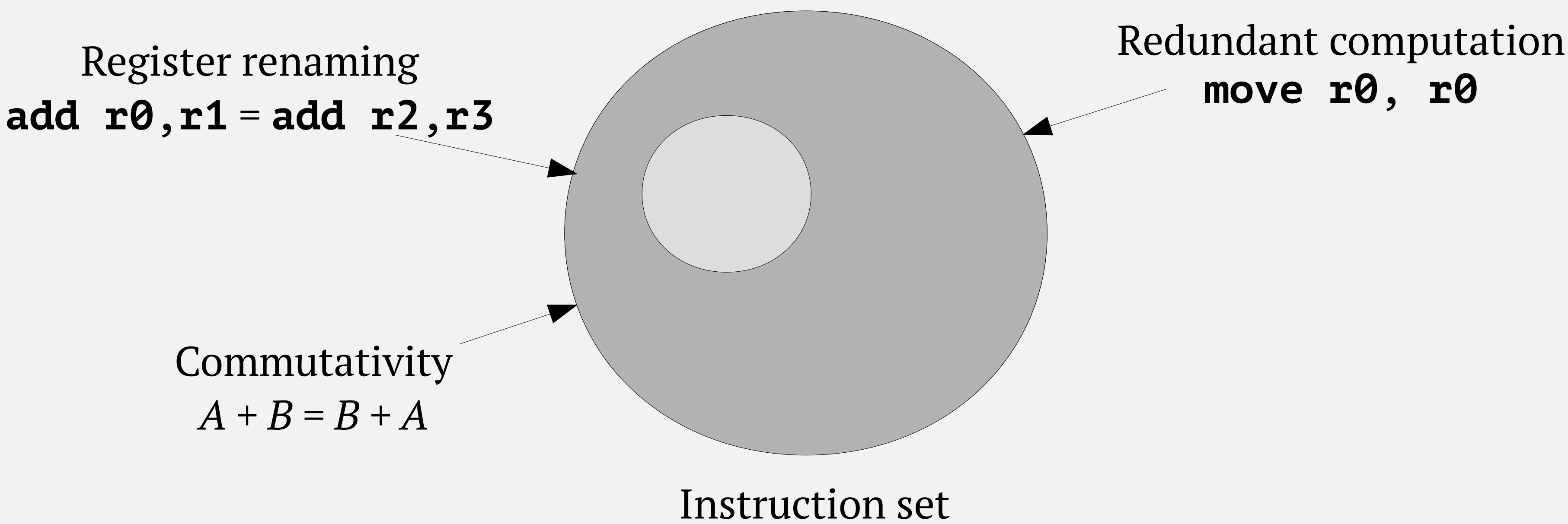
How do we quickly ignore bad sequences?



Superoptimization Fundamentals
Pruning

Not all instruction sequences are valid.

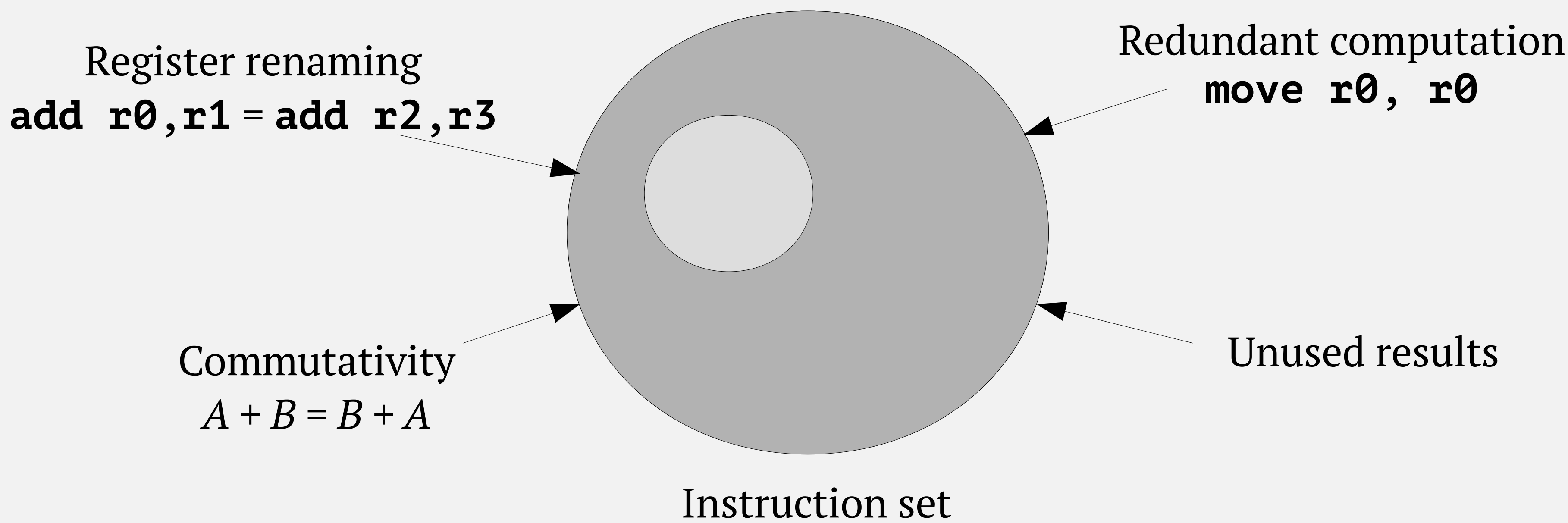
How do we quickly ignore bad sequences?



Superoptimization Fundamentals
Pruning

Not all instruction sequences are valid.

How do we quickly ignore bad sequences?



Is the sequence correct?

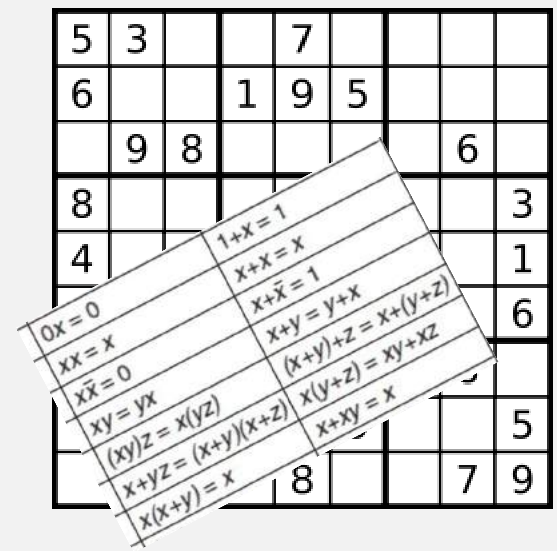
Is the sequence correct?

Testing (simulation)

Is the sequence correct?

Testing
(simulation)

Mathematical proof
(symbolic solving)



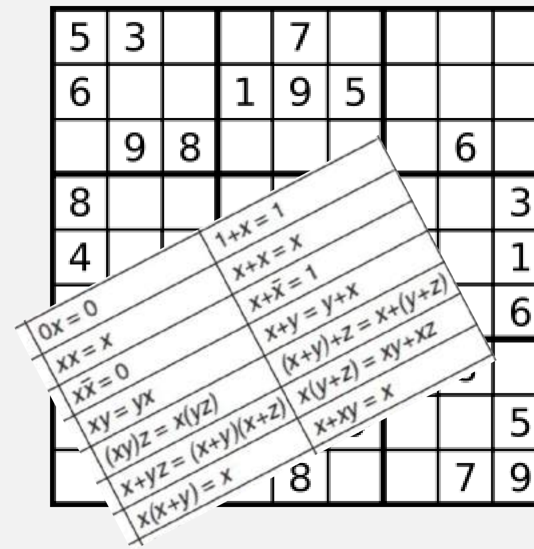
Superoptimization Fundamentals Testing

Is the sequence correct?

Testing (simulation)

1. Choose some input
2. Run/simulate
3. Check output

Mathematical proof (symbolic solving)



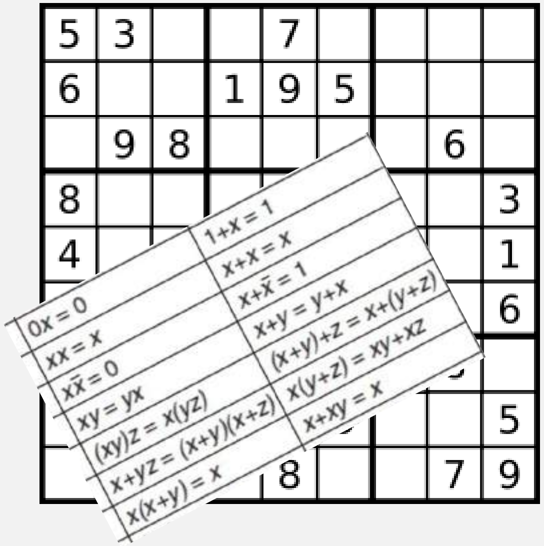
Superoptimization Fundamentals
Testing

Is the sequence correct?

Testing (simulation)

1. Choose some input
2. Run/simulate
3. Check output

Mathematical proof (symbolic solving)



Formal verification

Proves the sequence correct

Slow

Superoptimization Fundamentals
Testing

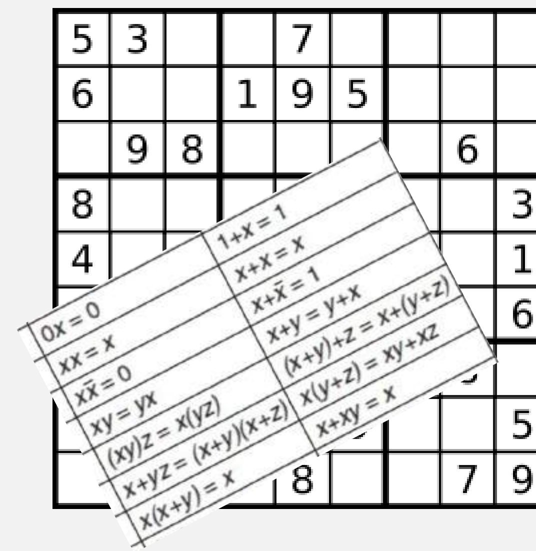
Is the sequence correct?

Testing (simulation)

1. Choose some input
2. Run/simulate
3. Check output

Use Both

Mathematical proof (symbolic solving)



Formal verification

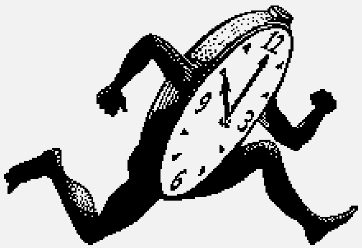
Proves the sequence correct

Slow

Which sequence is the best?

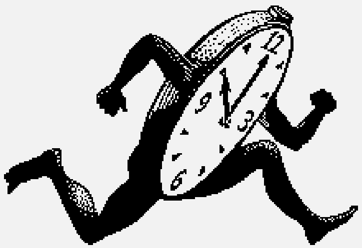
Which sequence is the best?

Execution time



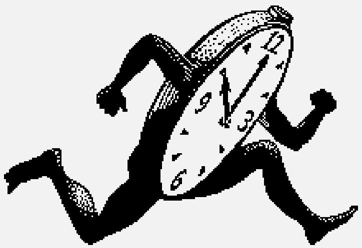
Which sequence is the best?

Execution time



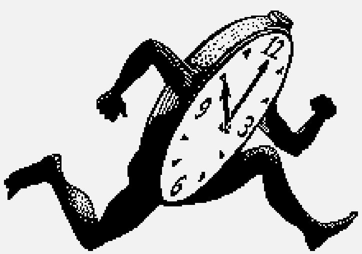
Code size

Which sequence is the best?

Execution time	Code size	Energy consumption
		

Which sequence is the best?

Execution time



Code size

Energy consumption

If you can enumerate the instructions in cost order, the first correct sequence is the optimal sequence.

Restrict parameters

Restrict parameters

- Registers
  - 50% of instruction sequences of length 8 use less than 4 registers
- Immediate constants
  - Frequently used constants: -16 to +16, 2^n, 2^n-1

State of the Art
Search Space Pruning

Restrict parameters

- Registers
  - 50% of instruction sequences of length 8 use less than 4 registers
- Immediate constants
  - Frequently used constants: -16 to +16, 2^n, 2^n-1

Remove meaningless constructs

- `mov r0, r0`
- `add r0, r0, #0`

- Restrict parameters
  - Registers
    - 50% of instruction sequences of length 8 use less than 4 registers
  - Immediate constants
    - Frequently used constants: -16 to +16, 2^n, 2^n-1
- Remove meaningless constructs
  - `mov r0, r0`
  - `add r0, r0, #0`
- Canonical form

mov r1, r0 has many equivalent versions

State of the Art
Canonical Form

mov r1, r0 has many equivalent versions

Rename each register so they appear in sequence:

```
mov r1, r0
mov r4, r2
mov r2, r8
```

```
mov r1, r0
```

State of the Art
Canonical Form

mov r1, r0 has many equivalent versions

Rename each register so they appear in sequence:

```
mov r1, r0
mov r4, r2        mov r1, r0
mov r2, r8
```

With 16 registers this replaces 16*15 equivalent versions

State of the Art
Canonical Form

```
add r4, r8, r1          add r2, r1, r0
orr r8, r4, #1   --->   orr r1, r2, #1
sub r1, r2, #8          sub r0, r3, #8
```

State of the Art
Canonical Form

```
add r4, r8, r1          add r2, r1, r0
orr r8, r4, #1   →      orr r1, r2, #1
sub r1, r2, #8          sub r0, r3, #8
```

Single three operand instruction:

```
add rX, rX, rX   →   add r0, r0, r0
                     add r0, r0, r1
                     add r0, r1, r0
                     add r0, r1, r1
                     add r0, r1, r2
```

5 unique forms

Data processing instructions

- 16 ops, each using 3 of 16 possible registers.
- E.g.

```
add r0, r1, r2
sub r3, r4, r5
```

Data processing instructions

- 16 ops, each using 3 of 16 possible registers.
- E.g.

```
add r0, r1, r2
sub r3, r4, r5
```

Instructions	Normal	Canonical	Canonical (4 registers)
1	65,536	80	80
2	4,294,967,296	51,968	47,872
3	281,474,976,710,656	4,157,669,376	45,264,896
4	18,446,744,073,709,551,616	276,142,292,992	45,880,115,200

Data processing instructions

- 16 ops, each using 3 of 16 possible registers.
- E.g.

```
add r0, r1, r2
sub r3, r4, r5
```

Instructions	Normal	Canonical	Canonical (4 registers)
1	65,536	80	80
2	4,294,967,296	51,968	47,872
3	281,474,976,710,656	4,157,669,376	45,264,896
4	18,446,744,073,709,551,616	276,142,292,992	45,880,115,200

@200,000 tests/second

Data processing instructions

- 16 ops, each using 3 of 16 possible registers.
- E.g.

```
add r0, r1, r2
sub r3, r4, r5
```

Instructions	Normal	Canonical	Canonical (4 registers)
1	65,536	80	80
2	4,294,967,296	51,968	47,872
3	281,474,976,710,656	4,157,669,376	45,264,896
4	18,446,744,073,709,551,616	276,142,292,992	45,880,115,200

@200,000 tests/second 2.9 million years

Data processing instructions

- 16 ops, each using 3 of 16 possible registers.
- E.g.

```
add r0, r1, r2
sub r3, r4, r5
```

Instructions	Normal	Canonical	Canonical (4 registers)
1	65,536	80	80
2	4,294,967,296	51,968	47,872
3	281,474,976,710,656	4,157,669,376	45,264,896
4	18,446,744,073,709,551,616	276,142,292,992	45,880,115,200

@200,000 tests/second 2.9 million years 16 days

Data processing instructions

- 16 ops, each using 3 of 16 possible registers.
- E.g.

```
add r0, r1, r2
sub r3, r4, r5
```

Instructions	Normal	Canonical	Canonical (4 registers)
1	65,536	80	80
2	4,294,967,296	51,968	47,872
3	281,474,976,710,656	4,157,669,376	45,264,896
4	18,446,744,073,709,551,616	276,142,292,992	45,880,115,200

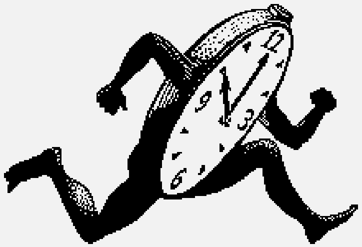
@200,000 tests/second 2.9 million years 16 days <3 days

Sequence cost is simple if code size is to be minimised

Sequence cost is simple if code size is to be minimised

Difficult to accurately measure the performance of short sequences of instructions.

- Pipeline modelling
- Cycle accurate simulation

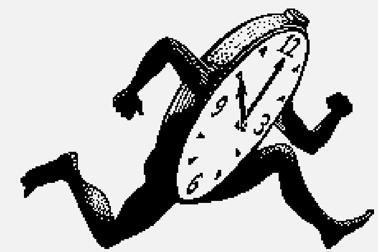


State of the Art Instruction Costing

Sequence cost is simple if code size is to be minimised

Difficult to accurately measure the performance of short sequences of instructions.

- Pipeline modelling
- Cycle accurate simulation



Energy

- Total Software Energy and Reporting (TSERO)

Characteristics of the instruction set affect how well a superoptimizer will perform.

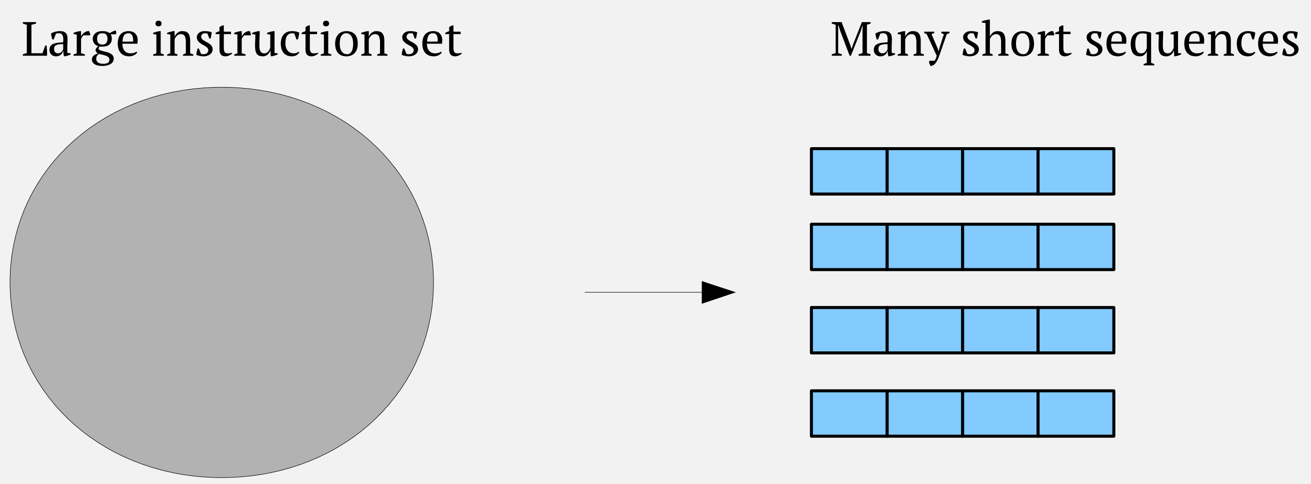
Characteristics of the instruction set affect how well a superoptimizer will perform.

Smaller instruction set → fewer optimal sequences (?)

State of the Art
Instruction Sets

Characteristics of the instruction set affect how well a superoptimizer will perform.

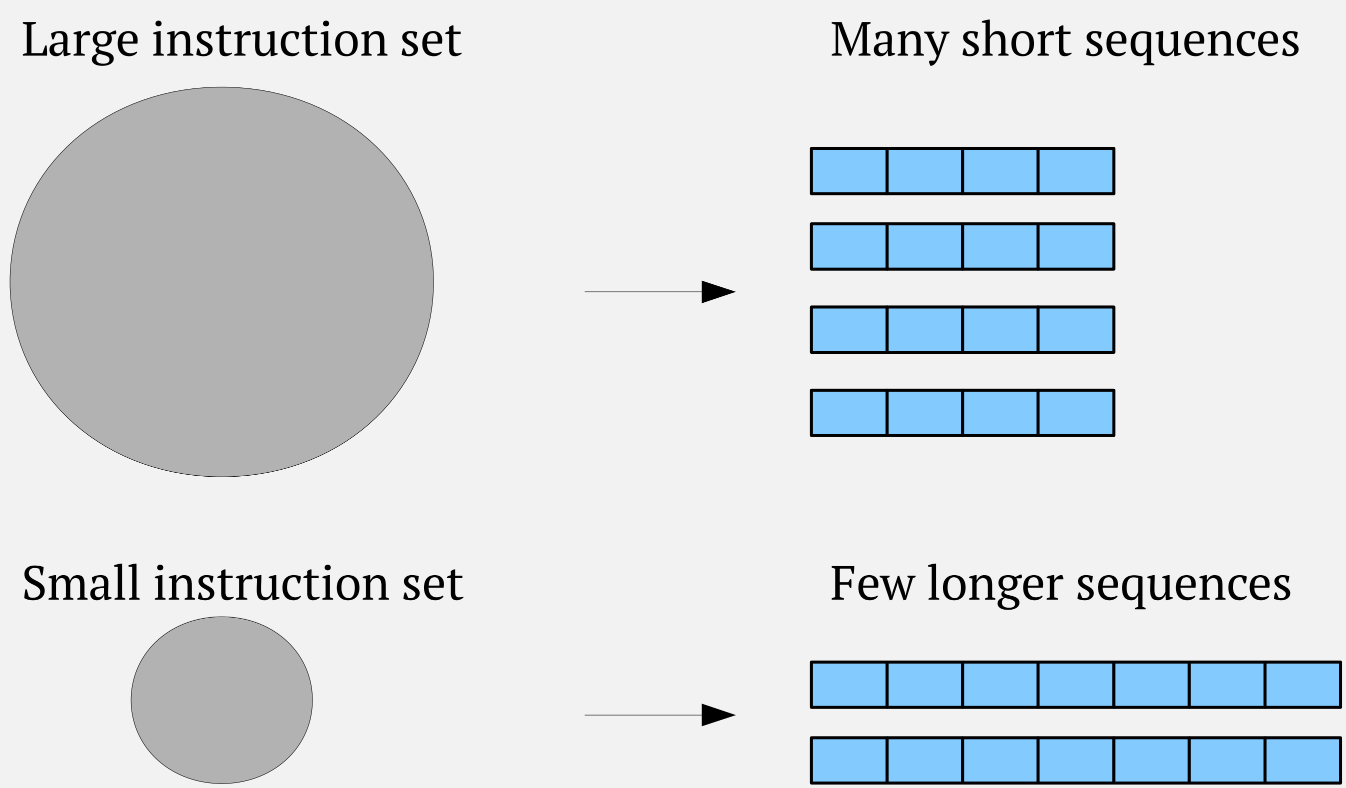
Smaller instruction set → fewer optimal sequences (?)



State of the Art Instruction Sets

Characteristics of the instruction set affect how well a superoptimizer will perform.

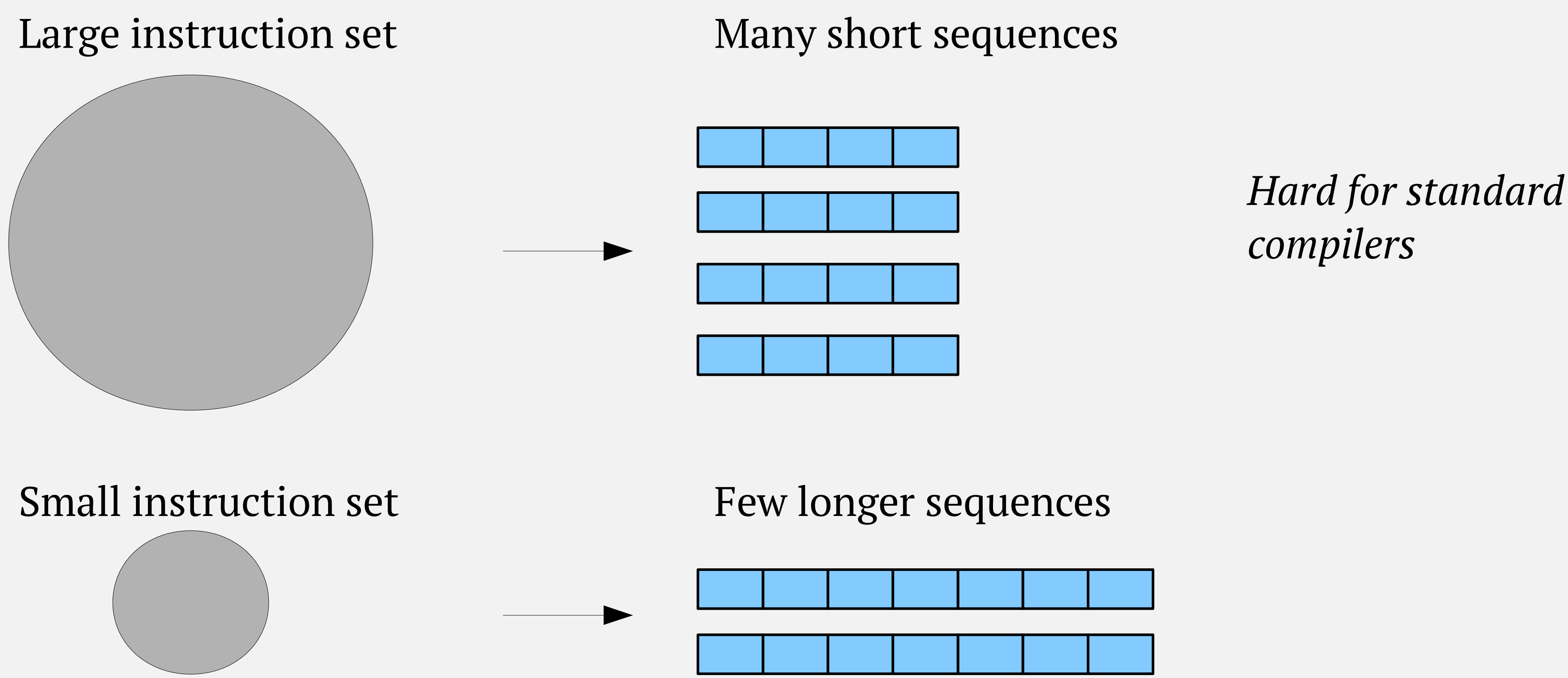
Smaller instruction set → fewer optimal sequences (?)



State of the Art
Instruction Sets

Characteristics of the instruction set affect how well a superoptimizer will perform.

Smaller instruction set → fewer optimal sequences (?)



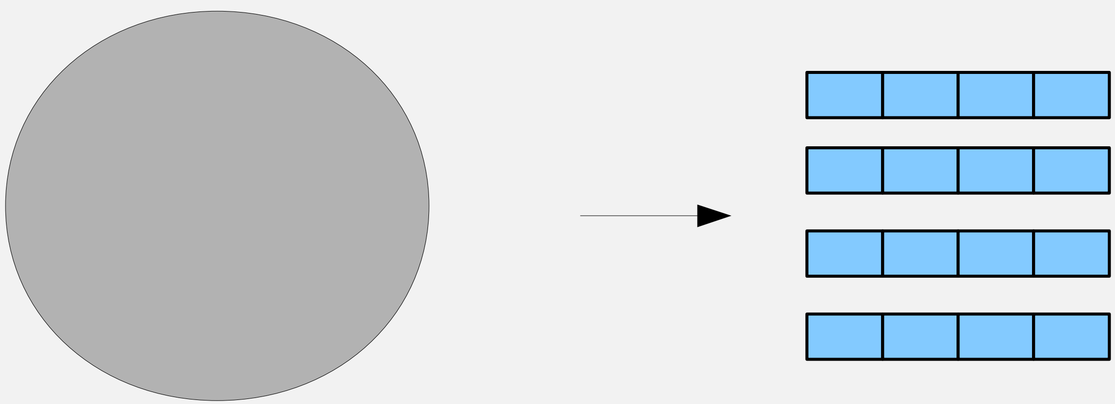
State of the Art
Instruction Sets

Characteristics of the instruction set affect how well a superoptimizer will perform.

Smaller instruction set → fewer optimal sequences (?)

Large instruction set

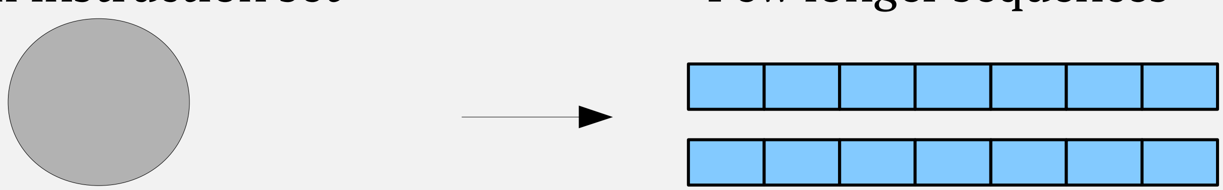
Many short sequences



Hard for standard compilers

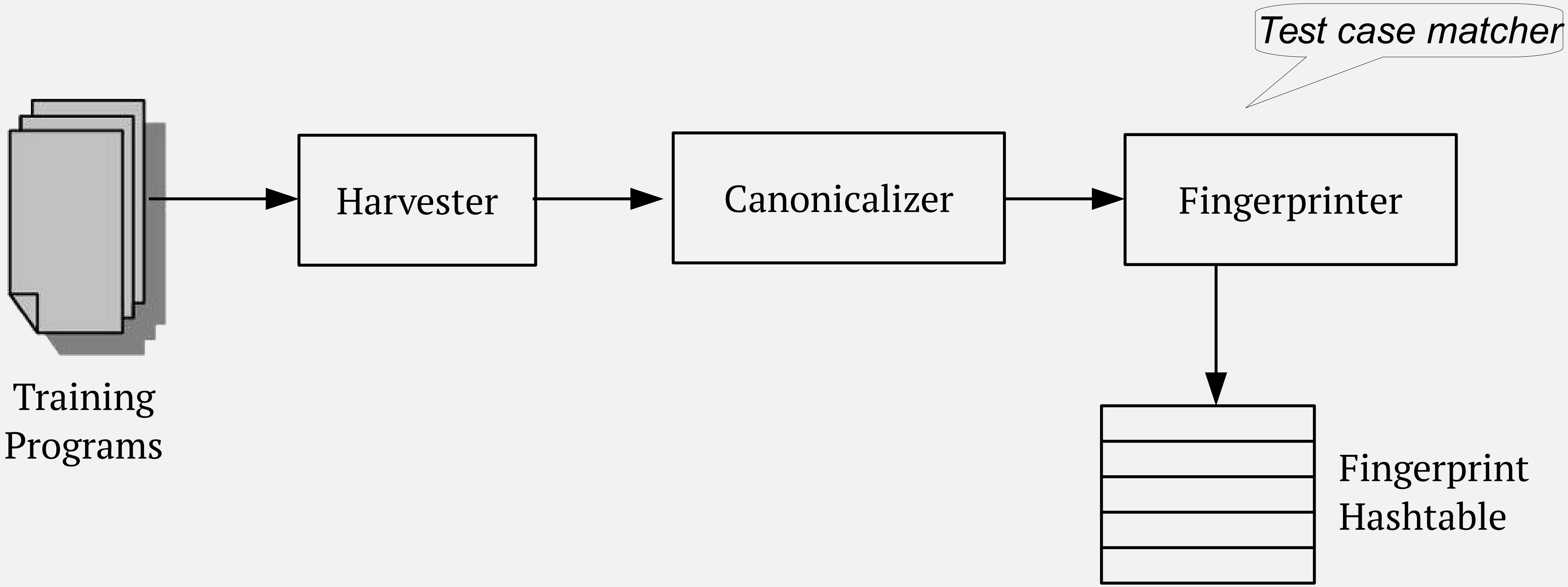
Small instruction set

Few longer sequences

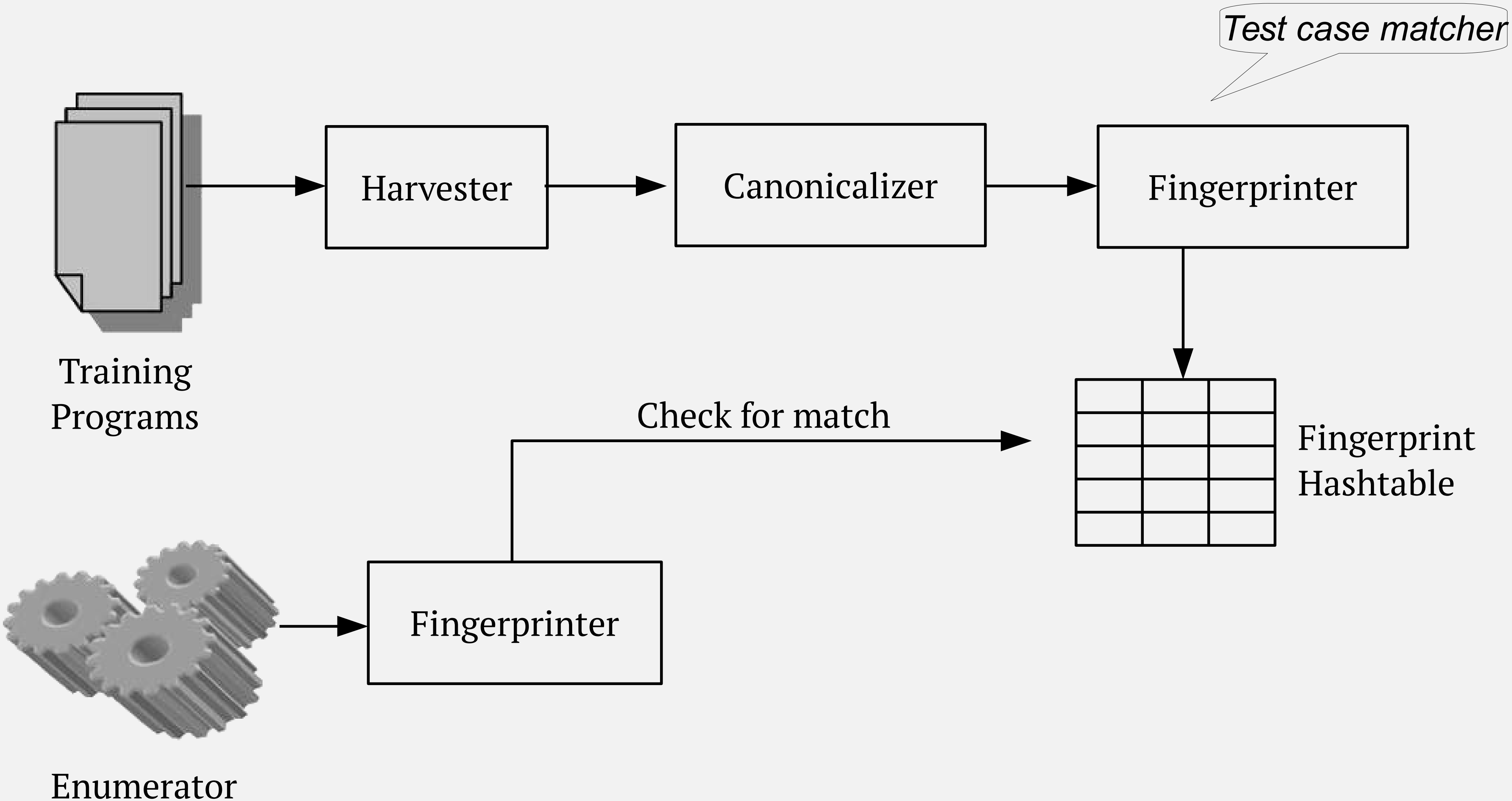


Easier for standard compilers

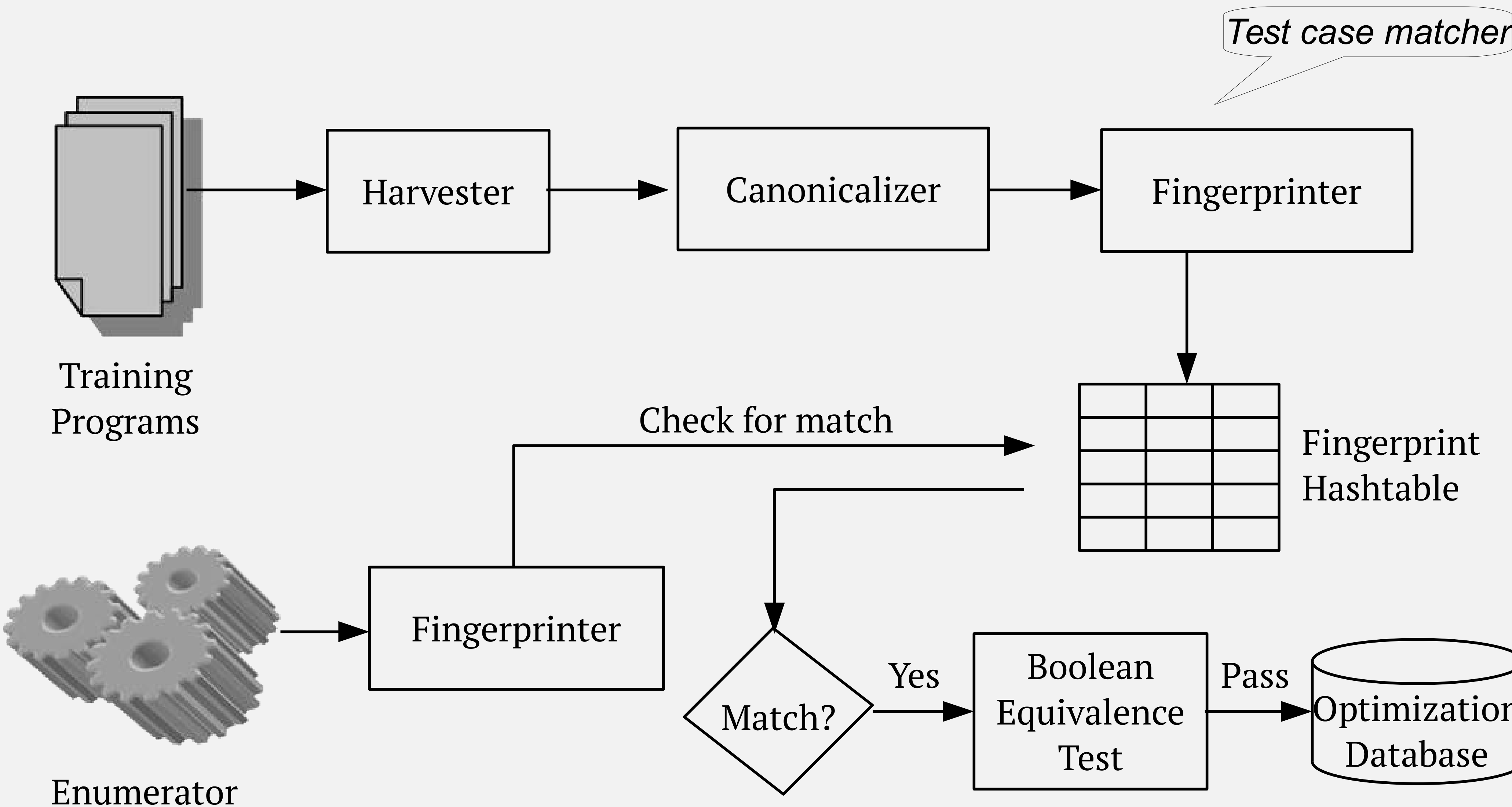
State of the Art
Peephole Superoptimizers



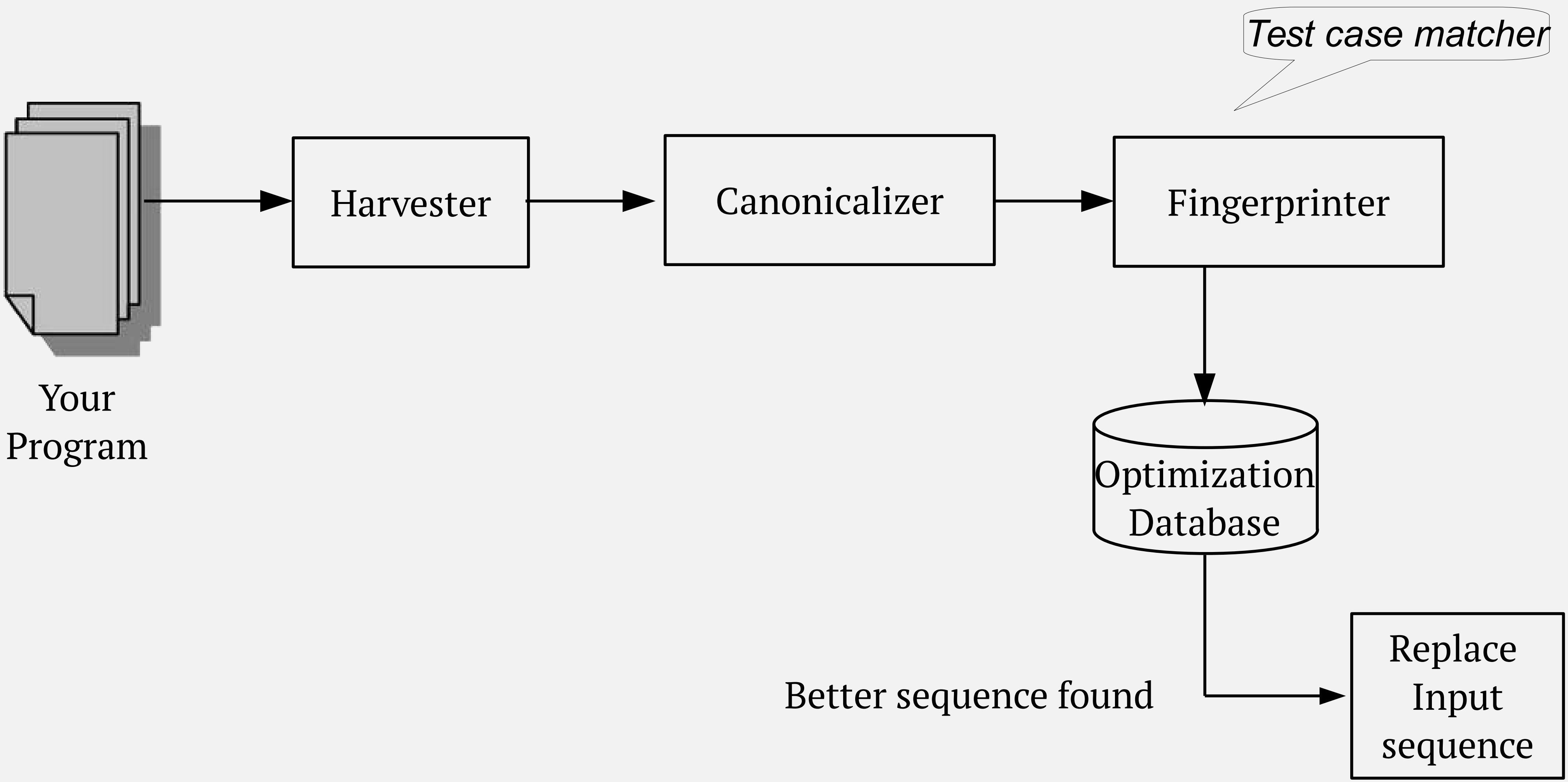
State of the Art
Peephole Superoptimizers



State of the Art Peephole Superoptimizers

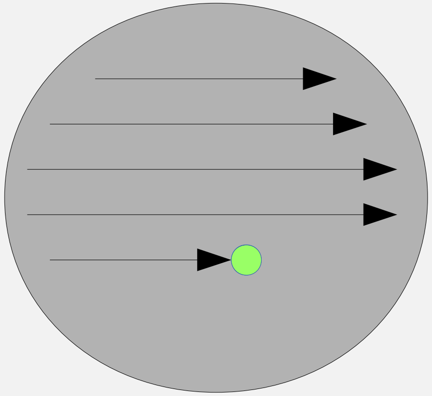


State of the Art
Peephole Superoptimizers

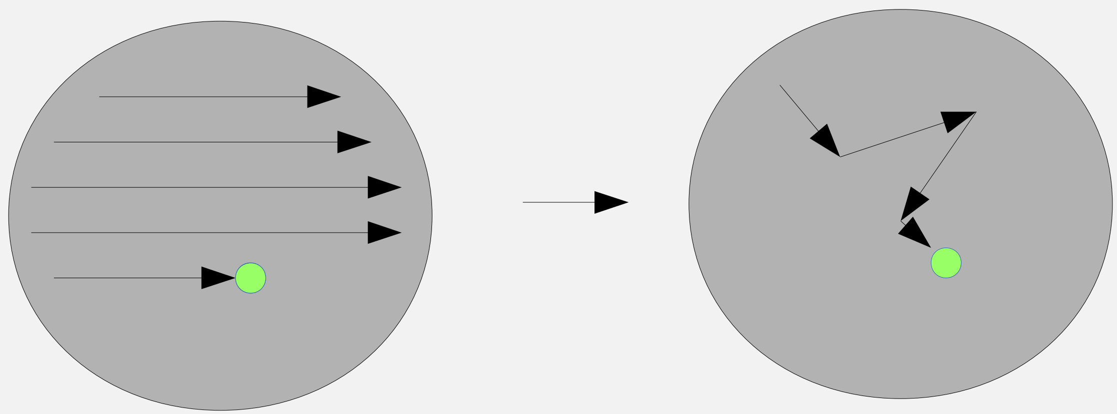


A different approach to instruction sequence enumeration

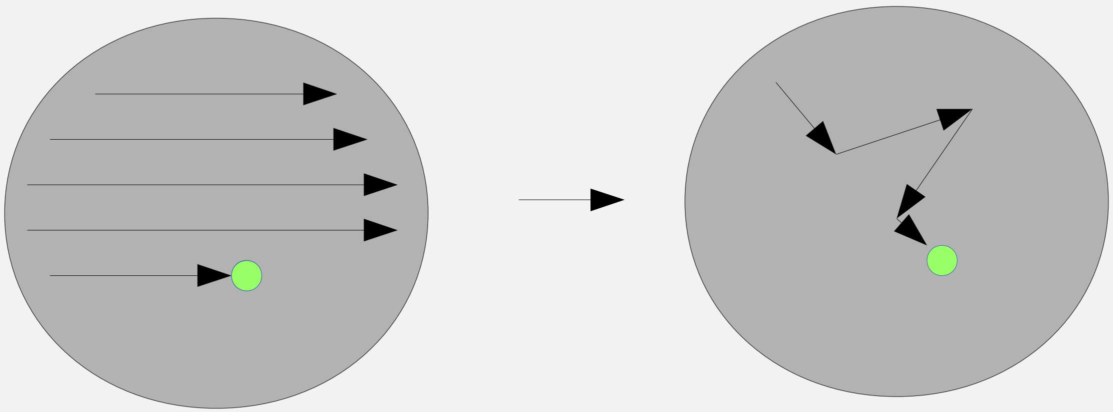
A different approach to instruction sequence enumeration



A different approach to instruction sequence enumeration



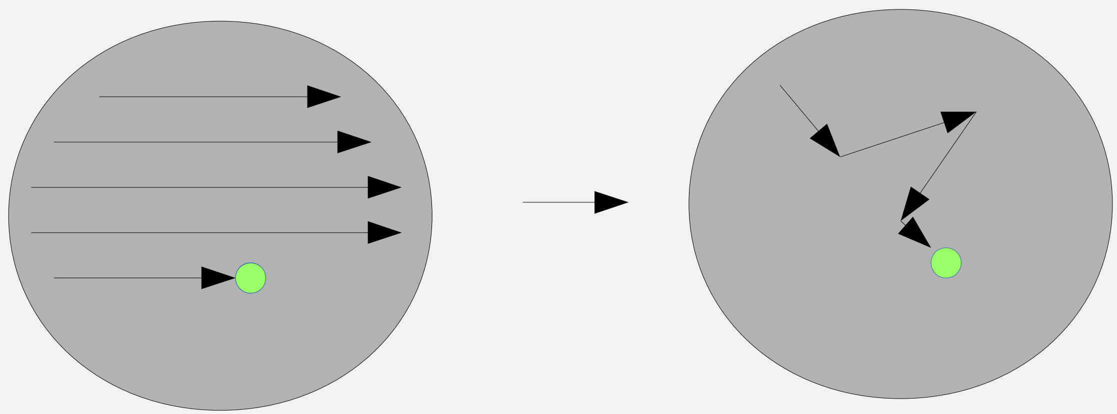
A different approach to instruction sequence enumeration



Longer sequences of instructions

- Sequences of >14 instructions were considered

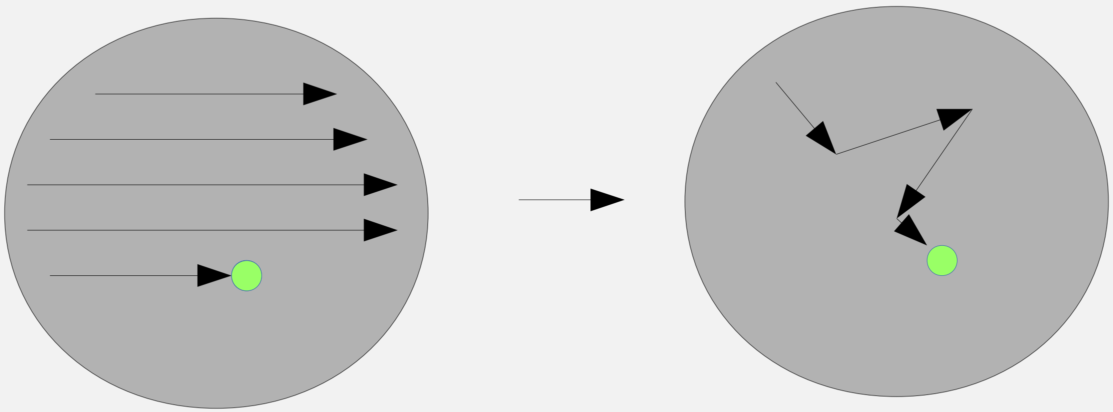
A different approach to instruction sequence enumeration



Longer sequences of instructions

- Sequences of >14 instructions were considered
- E.g. OpenSSL Montgomery multiplication 60% faster

A different approach to instruction sequence enumeration

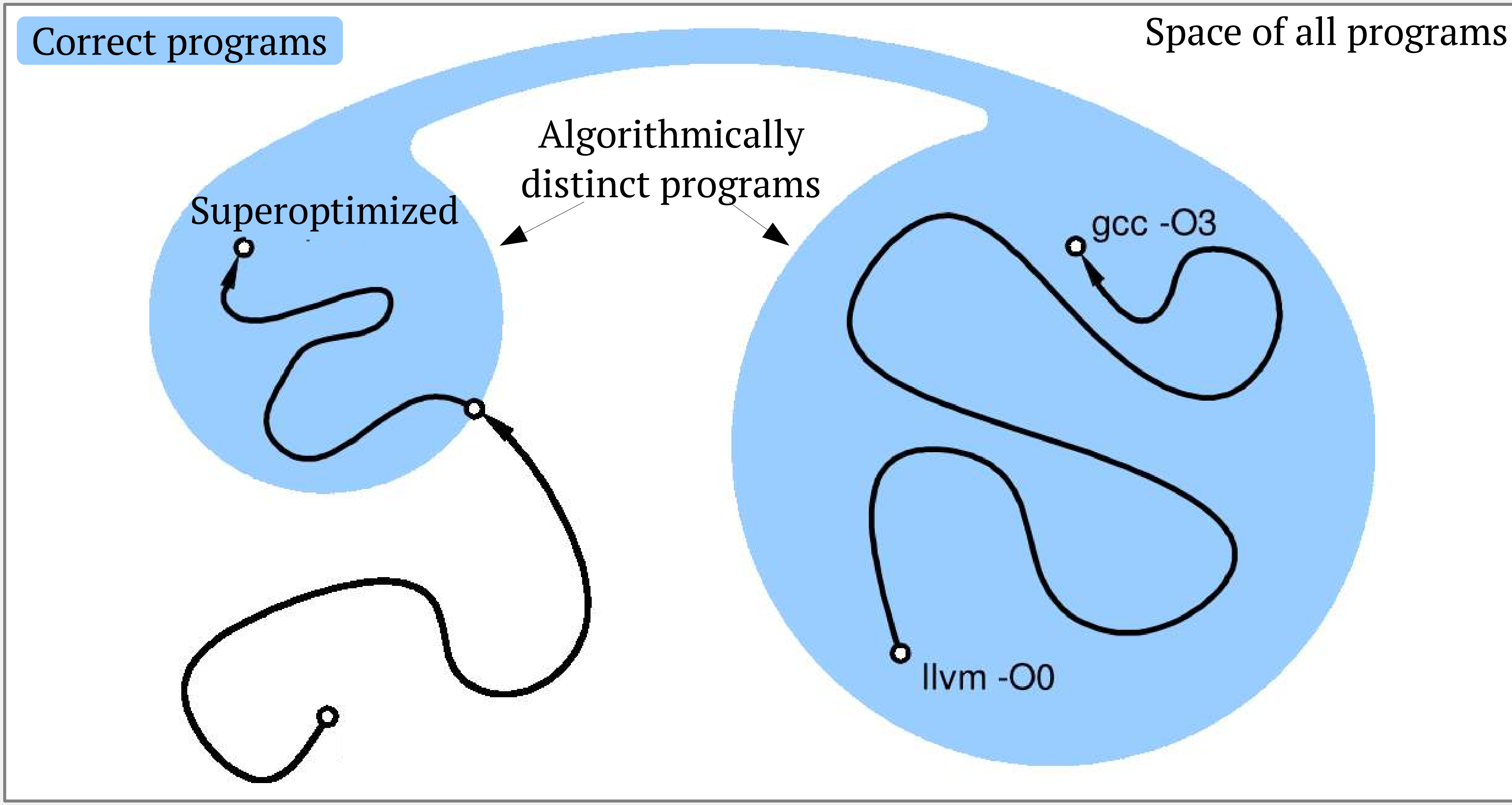


Longer sequences of instructions

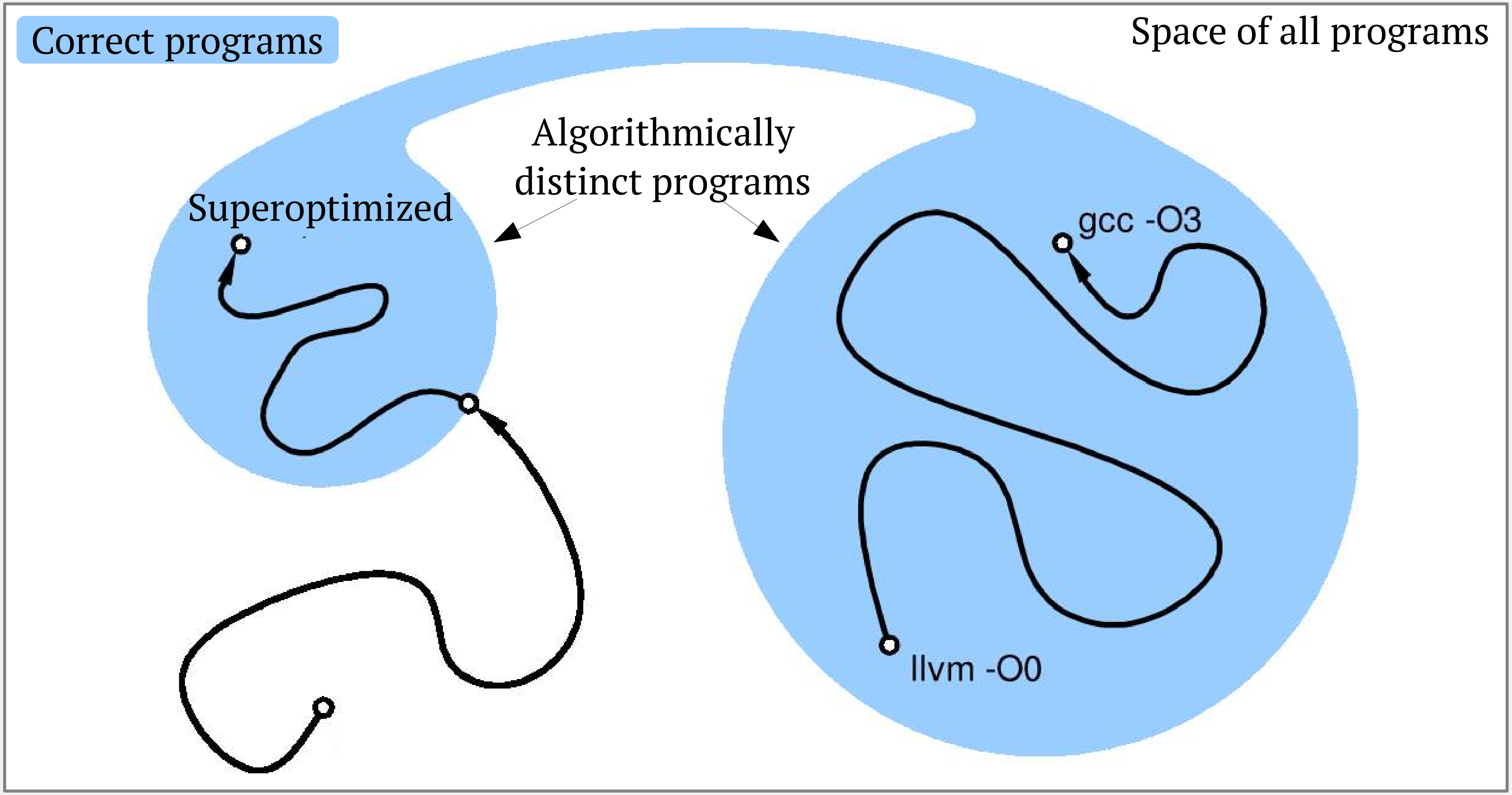
- Sequences of >14 instructions were considered
- E.g. OpenSSL Montgomery multiplication 60% faster

Schkufza, E., Sharma, R., & Aiken, A. (2013). Stochastic superoptimization. Architectural Support for Programming Languages and Operating Systems, 305.

State of the Art
Discovering New Algorithms



State of the Art
Discovering New Algorithms



Stochastic superoptimization's longer sequences make this more likely

GSO 2.0: A Superoptimizer Toolkit

Machine state	Bruteforce iterator	Instruction sequence testing	Parallelisation
Instructions	Canonical form iterator	Instruction equivalence checking	
Slots	Constants iterator	Peephole superoptimizer testing	
	Stochastic iterator		

Thank You

www.embecosm.com