

GNU Superoptimizer 2.0

James Pallister & Jeremy Bennett
Embecosm

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?

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

```
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:
```

int sign (int n)	cmp.l d0, 0	add.l d0, d0
{	ble L1	subx.l d1, d1
if (n > 0)	move.l d1, 1	negx.l d0
return 1;	bra L3	addx.l d1, d1
else if (n < 0)	L1:	
return -1;	bge L2	
else	move.l d1, -1	
return 0;	bra L3	
}	L2:	
	move.l d1, 0	
	L3:	

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
---	---	----

0	6	-1
---	---	----

-1

x	d0	d1
---	----	----

0	0	
---	---	--

0	0	
---	---	--

0	0	0
---	---	---

0	0	0
---	---	---

0	0	0
---	---	---

0

x	d0	d1
---	----	----

0	2	
---	---	--

0	4	
---	---	--

0	4	0
---	---	---

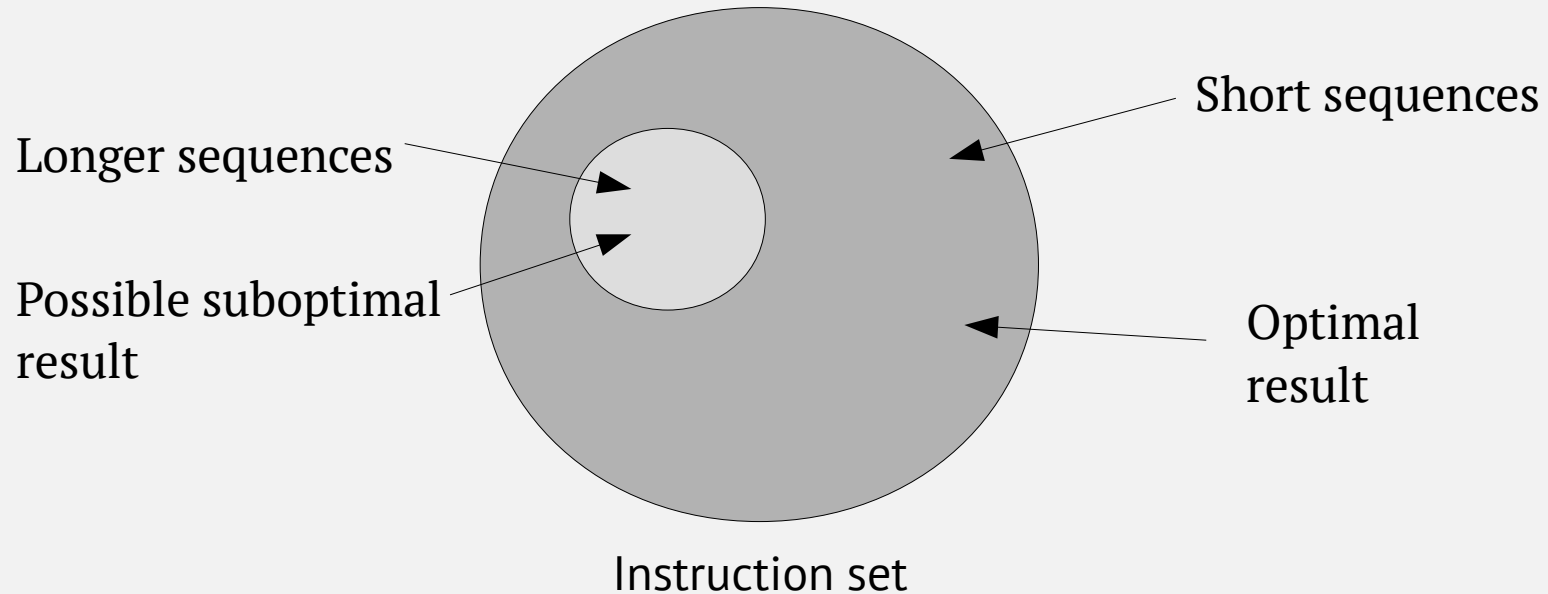
1	-4	0
---	----	---

0	-4	1
---	----	---

1

Generating the sequences of instructions

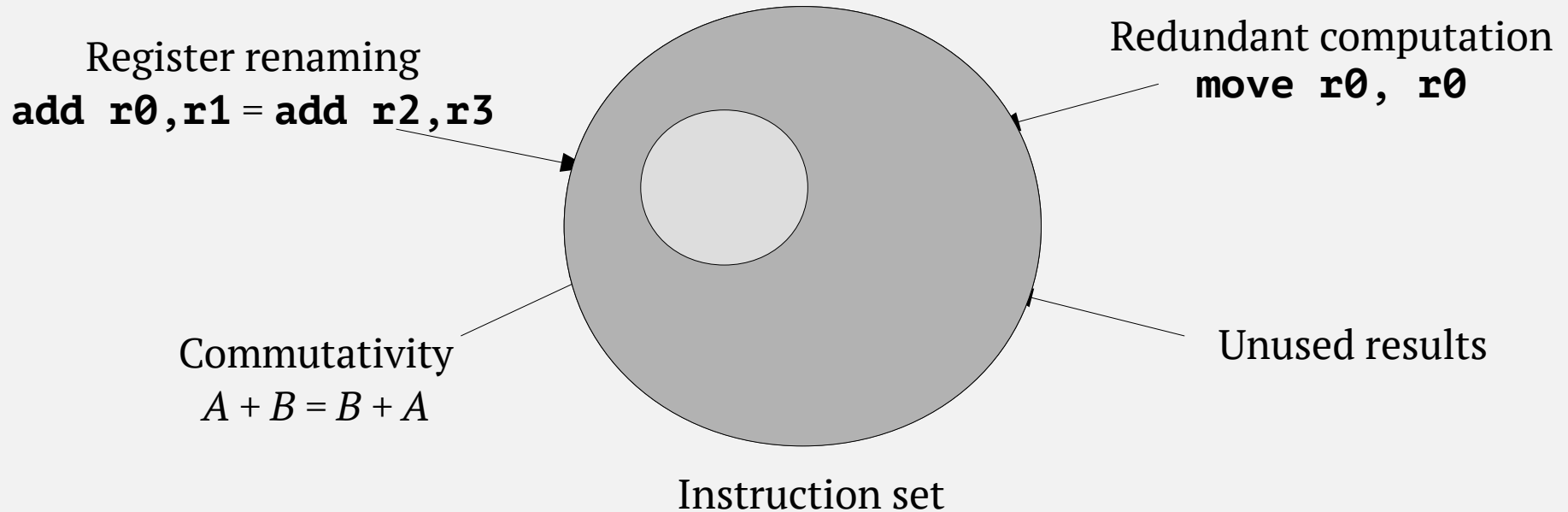
- But doing them all takes far too long



How to select the sequences of instructions?

Not all instruction sequences are valid.

How do we quickly ignore bad sequences?



Testing (simulation)



Mathematical proof (symbolic solving)

Formal verification
Proves the sequence correct
Slow

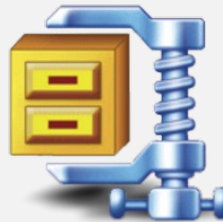
1. Choose some input
2. Run/simulate
3. Check output

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

- 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

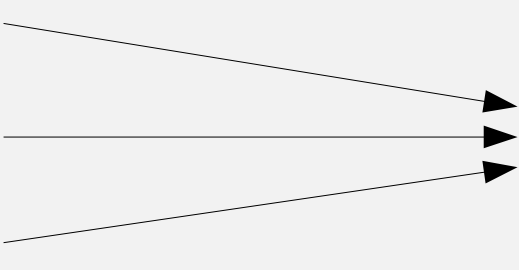
- **mov** r0, r0
- **add** r0, r0, #0

Canonical form

mov r1, r0 has many equivalent versions

Rename each register so they appear in sequence:

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



mov r1, r0

With 16 registers this replaces 16×15 equivalent versions

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

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

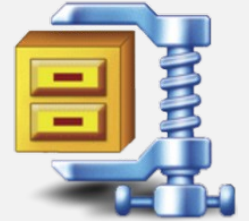
@200,000 tests/second

2.9 million years

16 days

<3 days

Sequence cost is simple if code size is to be minimised



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

- Pipeline modelling
- Cycle accurate simulation



Energy

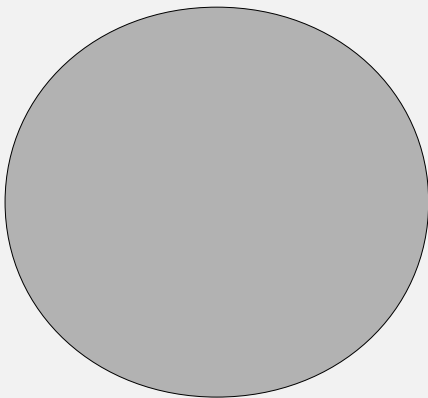
- Total Software Energy and Reporting (TSERO)



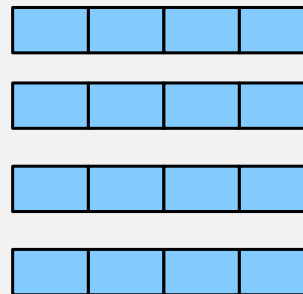
Characteristics of the instruction set affect how well a superoptimizer will perform.

Smaller instruction set → fewer optimal sequences (?)

Large instruction set

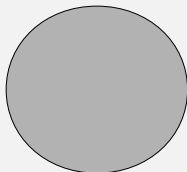


Many short sequences

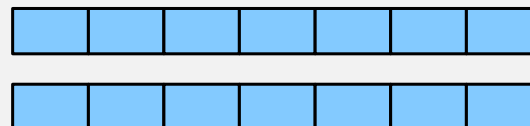


Hard for standard compilers

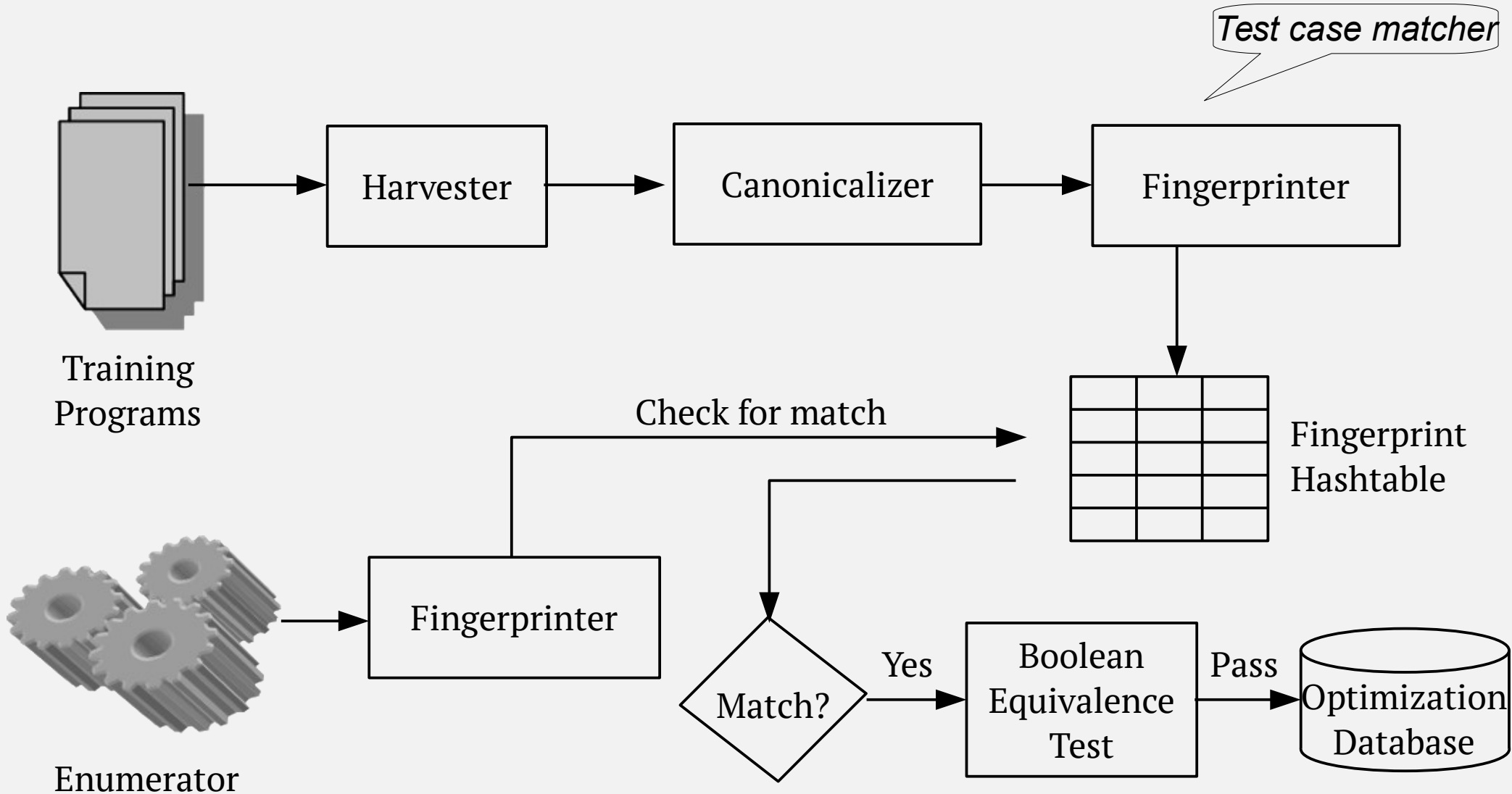
Small instruction set

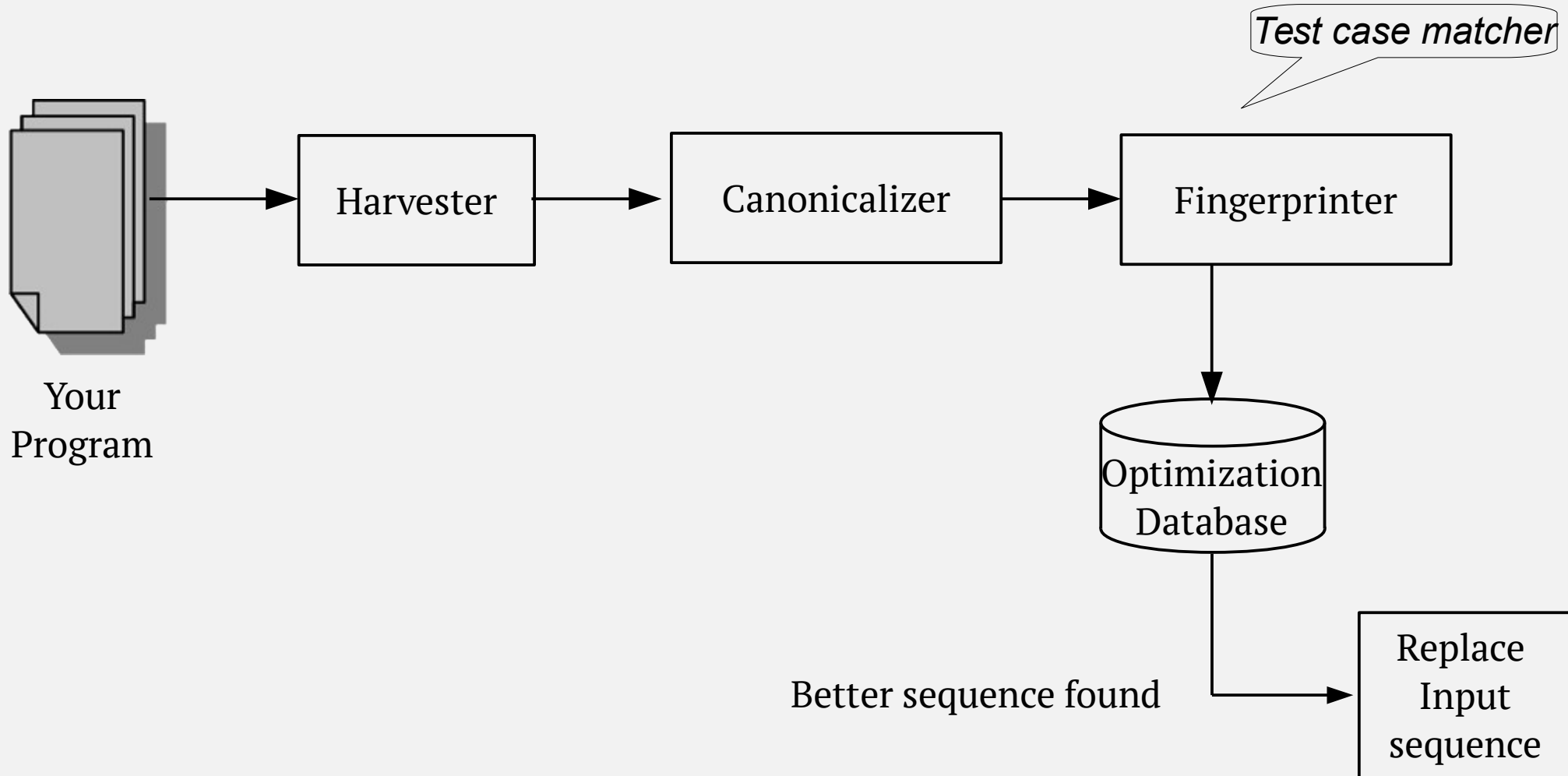


Few longer sequences

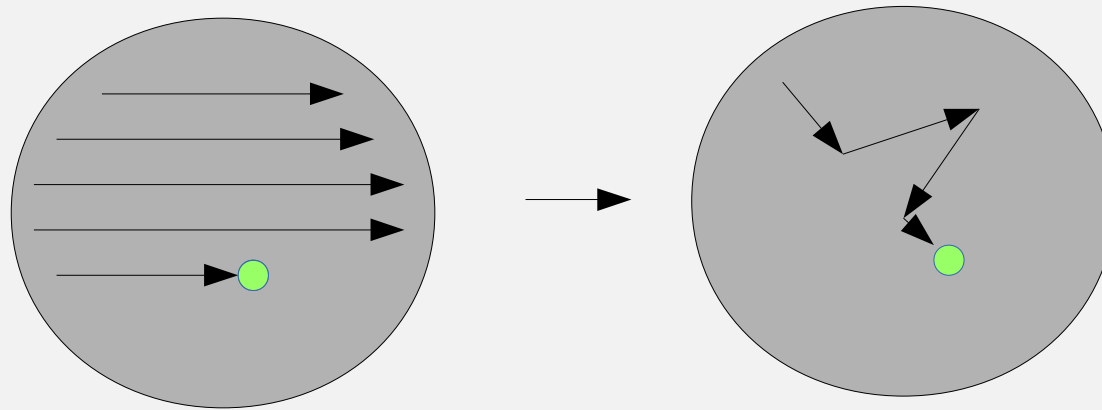


Easier for standard compilers





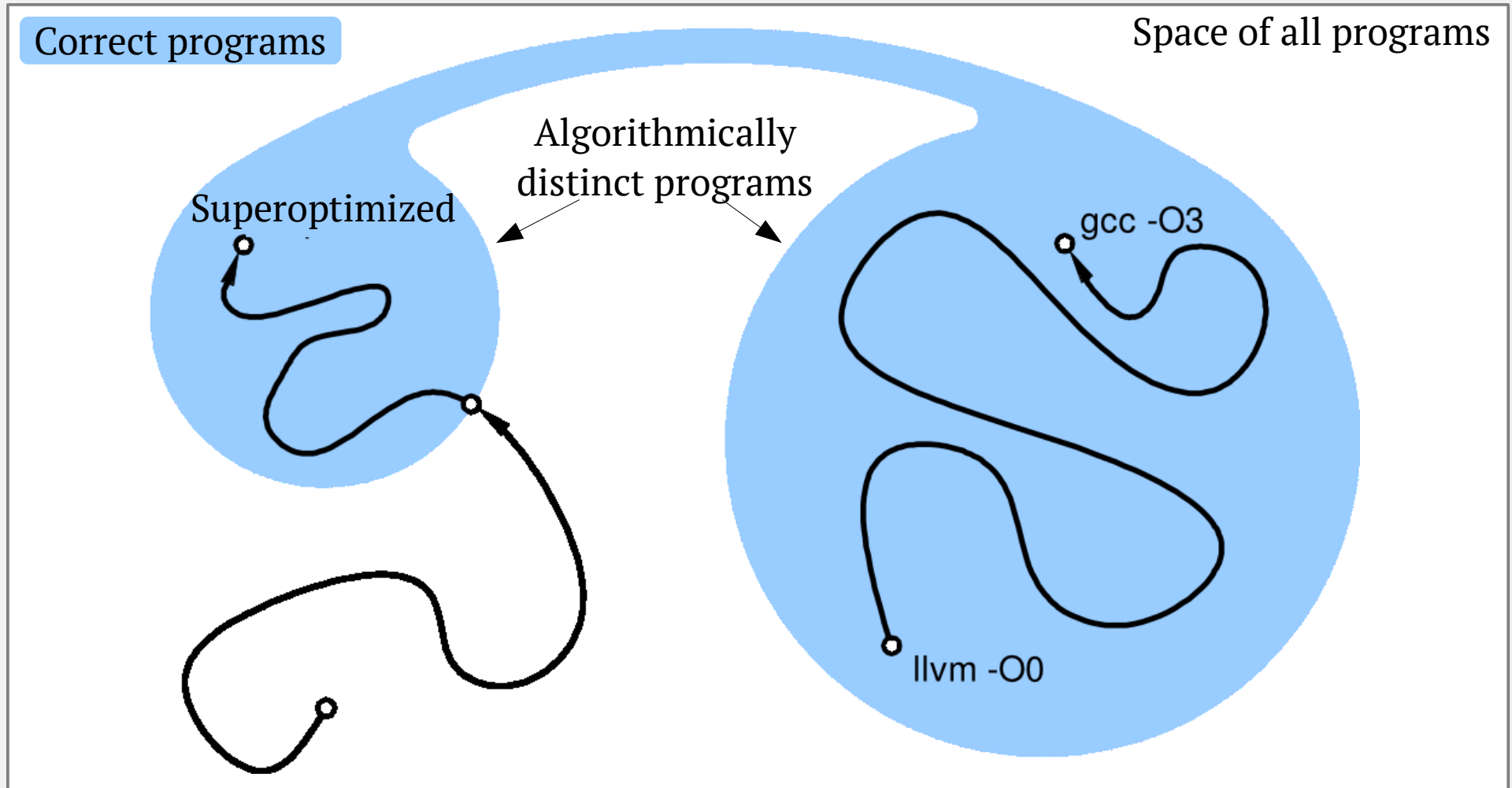
A different approach to instruction sequence enumeration



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.



Stochastic superoptimization's longer sequences make this more likely

GSO 1.0 has some limitations

- arithmetic only
- single result only
- generates impossible sequences
- only supports carry flag
- very simple cost model

But GSO 1.0 is very fast

GSO 2.0 approach is a toolkit

- multiple approaches
- can be slower
- use in conjunction with GSO 1.0

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

Machine state

Instructions

Slots

Describe what the instructions can access.

- registers
- flags
- memory

Machine state

Instructions

Slots

What operations can the processor perform.

- the “simulation” aspect of the superoptimizer
- helpers for basic input and output of instructions
- can be automatically generated

Machine state

Instructions

Slots

What the superoptimizer can change in an instruction sequence.

- a slot containing registers

e.g. **add r0, r1**

- a slot containing constants

e.g. **add r0, #128**

- slots and machine state are passed to the instructions.

Machine state

Instructions

Slots

```
x86_Machine mach;
```

```
...
```

```
// add ax, bx
```

```
Instruction *add = new x86_add();
```

```
vector<Slot*> slots = add->getSlots();
```

```
slots[0]->setValue(0);
```

```
slots[1]->setValue(1);
```

```
add->execute(&mach, slots);
```

Machine state

Bruteforce
iterator

Instruction
sequence
testing

Parallelisation

Instructions

Canonical form
iterator

Instruction
equivalence
checking

Slots

Constants
iterator

Peephole
superoptimizer
testing

Stochastic
iterator

Bruteforce
iterator

Iterate over a sequence of items,
visiting every combination.

Canonical form
iterator

```
vector<unsigned> values = {0,1,2,3};  
vector<vector<unsigned>::iterator>  
indices;
```

Constants
iterator

```
indices.push_back(values.begin());  
indices.push_back(values.begin());
```

Stochastic
iterator

```
do {  
    ...
```

```
} while(bruteforceIterate(values,  
indices));
```

Bruteforce
iterator

Canonical form
iterator

Constants
iterator

Stochastic
iterator

Iterate over register slots, ensuring each register renaming gets tested only once.

```
vector<Slot*> slots;  
...  
  
canonicalIterator c_iter(slots);  
  
do {  
    ...  
} while(c_iter.next());
```

Bruteforce
iterator

Canonical form
iterator

Constants
iterator

Stochastic
iterator

Iterate over constant slots, reducing the search space by only testing the most frequently used constants.

```
vector<Slot*> slots;  
...  
  
constantIterator cn_iter(slots);  
  
do {  
  
    ...  
  
} while(cn_iter.next());
```

Bruteforce
iterator

Canonical form
iterator

Constants
iterator

Stochastic
iterator

Allow for types of superoptimizer other than bruteforce.

- directed exploration of the search space

Machine state

Bruteforce
iterator

Instruction
sequence
testing

Parallelisation

Instructions

Canonical form
iterator

Instruction
equivalence
checking

Slots

Constants
iterator

Peephole
superoptimizer
testing

Stochastic
iterator

Instruction
sequence
testing

Instruction
equivalence
checking

Peephole
superoptimizer
testing

Test a sequence of instructions.
Several forms of testing:

- test whether a sequence of instructions matches another
- test whether a sequence of instructions matches an arbitrary function
- just execute the sequence

Support for executing multiple tests
for probabilistic correctness

Instruction
sequence
testing

Instruction
equivalence
checking

Peephole
superoptimizer
testing

Example:

```
vector<Instruction*> insns,  
goal_insns;  
vector<Slots*> slots, goal_slots;
```

... set up the above variables ...

```
bool success = testEquivalence(  
    insns, slots,  
    goal_insns, slots);
```

Instruction
sequence
testing

Instruction
equivalence
checking

Peephole
superoptimizer
testing

Verify that the instructions are equivalent for all possible inputs.

- translating to SMT form
- call an external solver
- guarantees correctness

This is expensive – the previous sequence tests should be used for quick exclusion.

Instruction
sequence
testing

Instruction
equivalence
checking

Peephole
superoptimizer
testing

Support for peephole superoptimizers: test a large number of sequences simultaneously.

- find many equivalent sequences
- use these sequences to form new peephole optimisations
- see “*Peephole Superoptimization*”, Bansal et al. 2006 for more details.

Machine state

Bruteforce
iterator

Instruction
sequence
testing

Parallelisation

Instructions

Canonical form
iterator

Instruction
equivalence
checking

Slots

Constants
iterator

Peephole
superoptimizer
testing

Stochastic
iterator

Parallelisation

Parallelise the search for valid instruction sequences.

- a generic master-slave MPI-based work queue
- based upon the bruteforceIterator
- the master distributes work items
- each slave requests a work item, sending a message back if it was successful.

Parallelisation

Example, master:

```
vector<vector<int>> instruction_ids =  
    {{0, 1, 2},  
        {0, 1, 2},  
        {0, 1, 2}};
```

...

```
distributeTasks(instruction_ids);
```

Parallelisation

Example, slave:

```
vector<int> insn_ids;

while(getWork(insn_ids)) {

    ...

    // computation with insn_ids

    ...
}

insn_ids = {0, 0, 0}
           {0, 0, 1}
           {0, 0, 2}
           {0, 2, 0}
           ...
```


Machine state ✓

Bruteforce
iterator ✓

Instruction
sequence
testing ✓

Parallelisation ✓

Instructions ✓

Canonical form
iterator ✓

Instruction
equivalence
checking

Slots ✓

Constants
iterator ✓

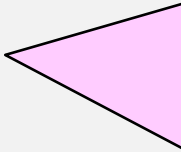
Peephole
superoptimizer
testing

Stochastic
iterator

Using the toolkit:

Bruteforce superoptimizers

- AVR (mostly complete)
- X86 (incomplete)
- ARM (planned)



add,	adc,	adiw,	sub,
subi,	sbc,	sbc_i,	sbiw,
and,	andi,	or,	ori,
eor,	com,	neg,	sbr,
cbr,	inc,	dec,	tst,
clr,	ser,	mul,	muls,
mulsu,	fmul,	fmuls,	
fmulsu,	cp,	cpc,	cpi,
mov,	movw,	ldi,	lsl,
lsr,	rol,	ror,	asr,
swap			

Peephole superoptimizers

Stochastic superoptimizers

```
std::string instruction_str =  
    "add r0, r0\n"  
    "add r0, r0";  
  
auto insn_factories = AvrMachine::getInstructionFactories();  
  
// Get a list of instructions and slots from the string  
vector<Instruction*> goal_insns;  
vector<Slot*> goal_slots;  
  
parseInstructionList(instruction_str, insn_factories,  
                    goal_insns, goal_slots);  
  
// current_factories will be used to generate the sequence  
vector<decltype(insn_factories)::iterator> current_factories;  
current_factories.push_back(insn_factories.begin());  
...
```

```
do {  
    // instruction sequence setup  
  
    // register iterator setup  
    // constant iterator setup  
  
    do {  
        do {  
  
            // perform tests on the instruction sequence  
  
        } while(reg_iter.next());  
    } while(cons_iter.next());  
  
} while(bruteforceIterate(insn_factories, current_factories));
```

```
// instruction sequence setup
vector<Instruction*> insns;

// Create the instructions, as per the current iterator
for(auto &factory: current_factories)
{
    Instruction *insn = (*factory)();

    insns.push_back(insn);

    auto s = insn->getSlots();
    slots.insert(slots.end(), s.begin(), s.end());
}

// register iterator setup
CanonicalIterator reg_iter(slots);

// constant iterator setup
ConstantIterator cons_iterator(slots);
```

// perform tests on the instruction sequence

```
if(testEquivalence(insns, slots, mach_initial, mach_expected))
{
    bool correct =
        testEquivalenceMultiple<AvrMachine>(insns, slots,
                                            goal_insns, goal_slots);

    if(correct)
    {
        cout << "Found:" << endl;
        cout << print(insns, slots) << endl;
    }
}
```

```
$ ./gsov2
```

Goal sequence:

```
add r0, r0  
add r0, r0
```

Found:

```
add r0, r0  
add r0, r0
```

Found:

```
ldi r16, 4  
mul r0, r16
```

What are commercially viable applications?

Optimization of emulation libraries

- libgcc and CompilerRt

Peephole discovery

- machine (and application) specific

Get the code

<https://github.com/embecosm/gso2>

Innovate UK

Technology Strategy Board

Thank You

www.embecoscsm.com