Virtual Machine Construction for Dummies

Jim Huang <jserv@0xlab.org>

Rifur Ni <rifurdoma@gmail.com>

Xatier Lee <xatierlike@gmail.com>

Jan 29, 2013 ! TOSSUG / March 7, 2013 ! 新竹碼農

Rights to copy

contact@0xlab.org

creative commons
COMMONS DEED

Corrections, suggestions, contributions and translations are welcome!

Latest update: Feb 21, 2015

Goals of this Presentation

- Build a full-functioned virtual machine from scratch
  - The full source code is inside the slides.
- Basic concepts about interpreter, optimizations techniques, language specialization, and platform specific tweaks.
- Brainfuck is selected as the primary programming language because
  - it's a very simple turing-complete programming language.
  - it's easier to write its compiler than its interpreter.
  - it's easier to write its interpreter than its real programs.

Brainfuck Programming Language

- created in 1993 by Urban Müller
- Only 8 instructions
  - Müller's Amiga compiler was 240 bytes in size
  - x86/Linux by Brian Raiter had 171 Bytes!

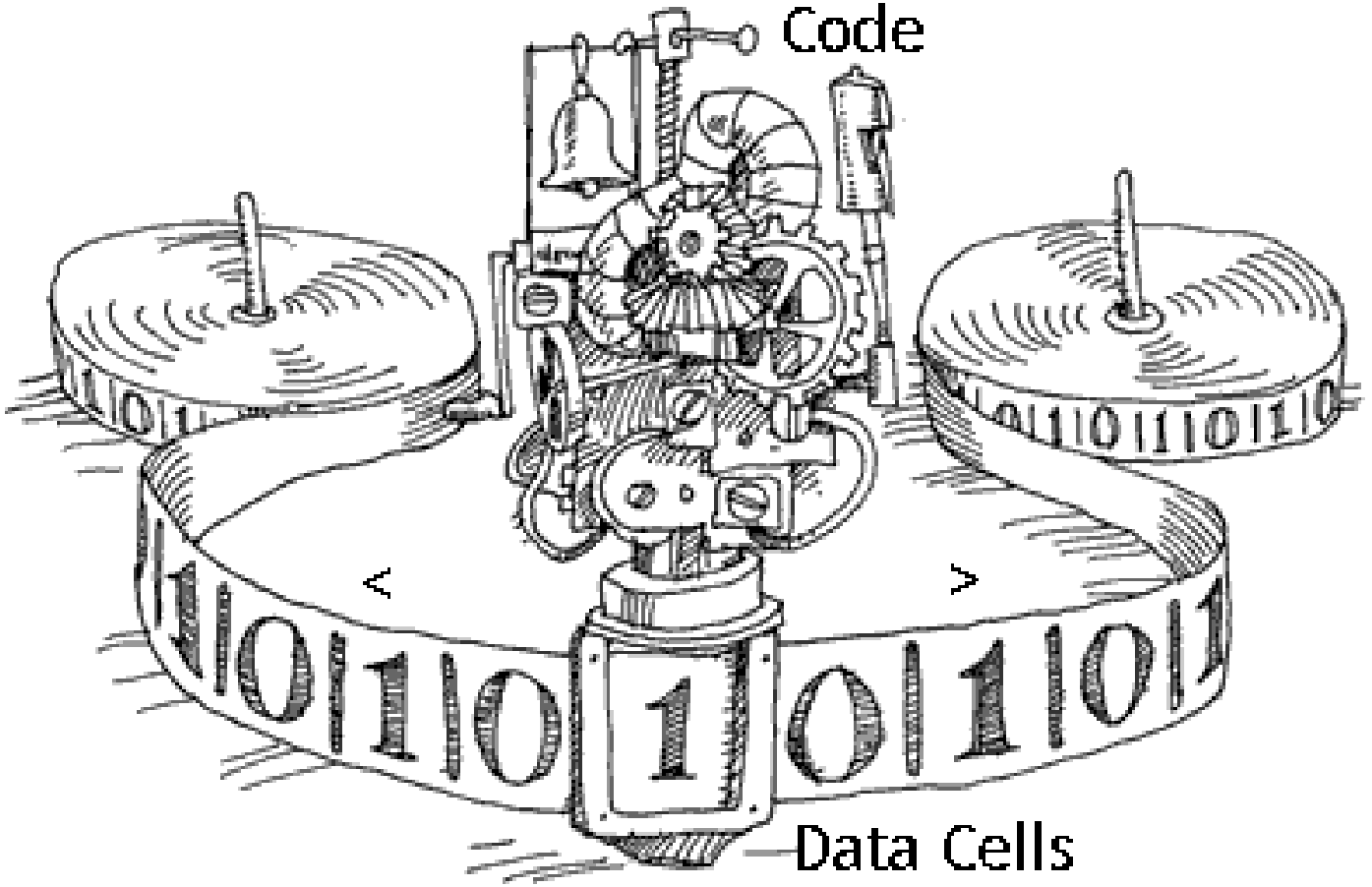
```
+++[>++++++[>++++++>++++++>++++>++
++>+<<<<<-]>++++>+++>+++>+<<<<<
-]>>.>.>+.>.>-----.
```

Learn such a stupid language! Why?

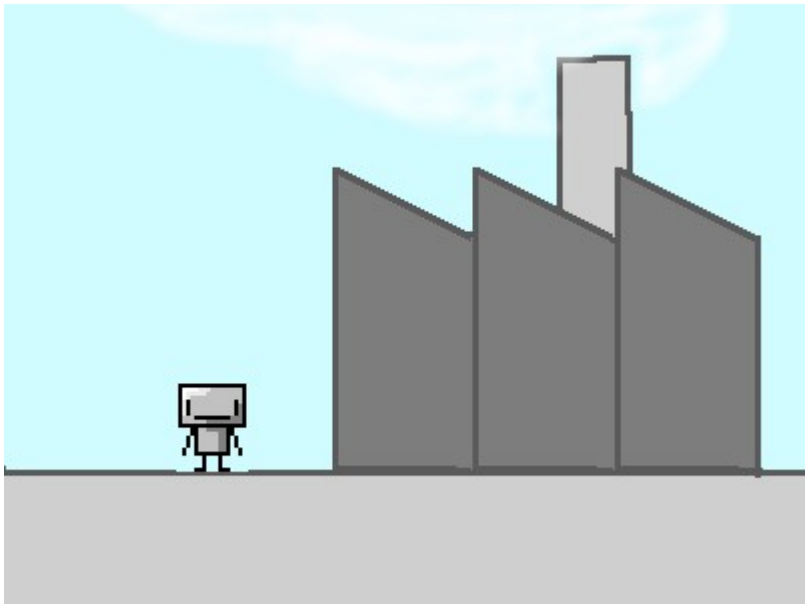
- Understand how basic a Turing-complete programming language can be.
  - A common argument when programmers compare languages is "*well they're all Turing-complete*", meaning that anything you can do in one language you can do in another.
- Once you've learnt brainfuck, you'll understand just how difficult it can be to use a Turing-complete language, and how that argument holds no water.

Source: http://skilldrick.co.uk/2011/02/why-you-should-learn-brainfuck-or-learn-you-a-brainfuck-for-great-good/

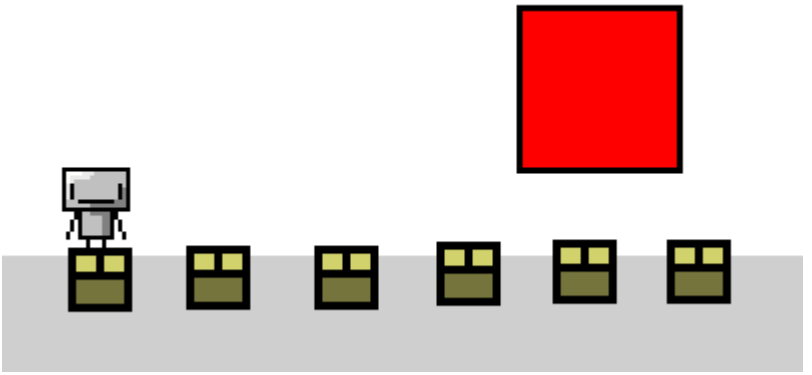
Brainfuck: Turing Complete

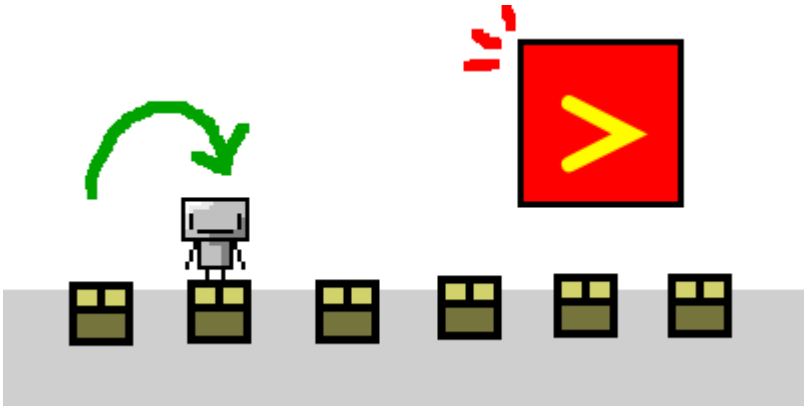


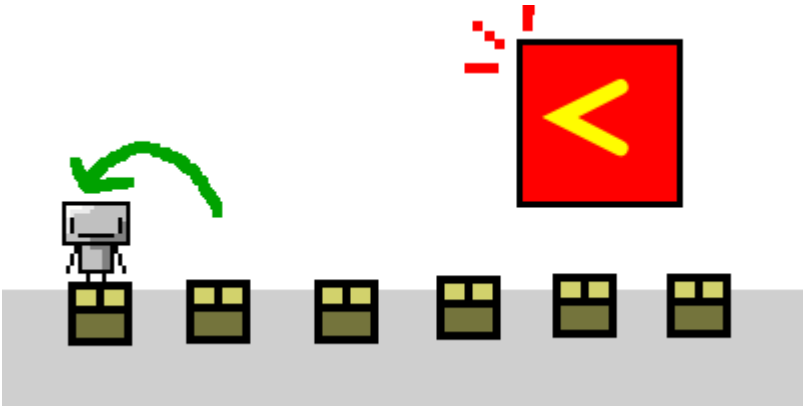
Source:
http://jonfwilkins.blogspot.tw/2011/06/happy-99th-birthday-alan-turing.html

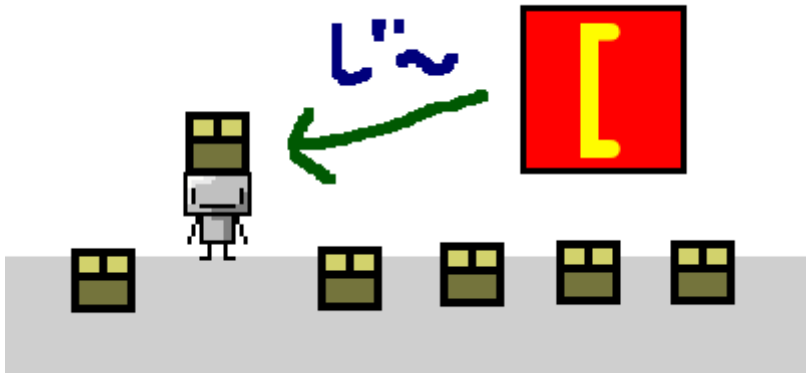


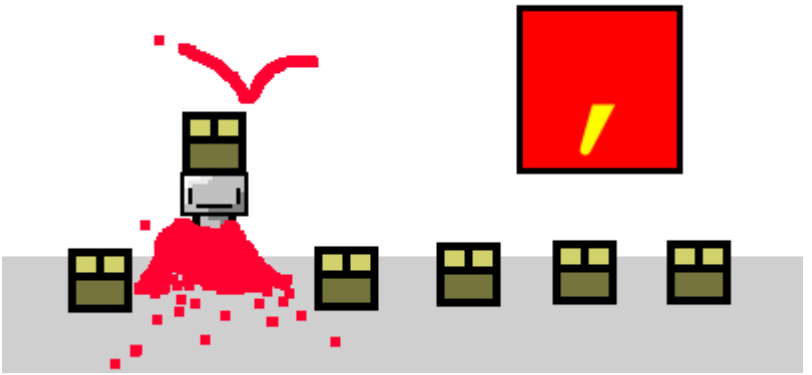
http://bugrammer.g.hatena.ne.jp/nisemono_san/20111114/1321218802











Brainfuck Instructions

(mapped to C language)

Brainfuck	C	
>	++p;	Increment the data pointer to point to the next cell.
<	--p;	Decrement the data pointer to point to the previous cell.
+	++*p;	Increment the byte value at the data pointer.
–	--*p;	Decrement the byte value at the data pointer.
.	putchar(*p);	Output the byte value at the data pointer.
,	*p = getchar();	Input one byte and store its value at the data pointer.
[	while (*p) {	If the byte value at the data pointer is zero, jump to the instruction following the matching] bracket. Otherwise, continue execution.
]	}	Unconditionally jump back to the matching [bracket.

Writing a Brainfuck compiler is Easy!

```
#!/usr/bin/awk -f
BEGIN {
  print "int main() {";
  print "  int c = 0;"; print "  static int b[30000];\n";
}
{
  gsub(/\]/, "  }\n");
  gsub(/\[/, "  while(b[c] != 0) {\n");
  gsub(/\+/, "  ++b[c];\n");
  gsub(/\-/, "  --b[c];\n");
  gsub(/>/, "  ++c;\n");
  gsub(/</, "  --c;\n");
  gsub(/\./, "  putchar(b[c]);\n");
  gsub(/\,/, "  b[c] = getchar();\n");
  print $0
}
END {
  print "\n  return 0;";
  print "}";
}
```

Brainfuck interpreter in portable C (1/3)

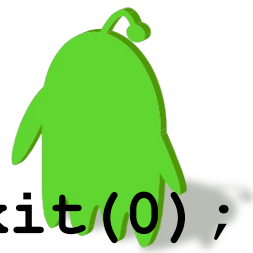
```
#include <stdio.h>
#include <stdlib.h>

int  p, r, q;
char a[5000], f[5000], b, o, *s = f;

void interpret(char *c)
{
    char *d;
    r++;
    while (*c) {
        switch (o = 1, *c++) {
        case '<': p--; break;
        case '>': p++; break;
        case '+': a[p]++; break;
        case '-': a[p]--; break;
```

Brainfuck interpreter in portable C (2/3)

```
case '.':
    putchar(a[p]);
    fflush(stdout); break;
case ',':
    a[p] = getchar();
    fflush(stdout); break;
case '[':
    for (b = 1, d = c; b && *c; c++)
        b += *c == '[', b -= *c == ']';
    if (!b) {
        c[-1] = 0;
        while (a[p]) interpret(d);
        c[-1] = ']'; break;
    }
case ']':
    puts("Unblanced brackets"),exit(0);
```



Brainfuck interpreter in portable C (3/3)

```
            default: o = 0;
            }
            if (p < 0 || p > 100)
                puts("Range error"), exit(0);
        }
        r--;
    }
    int main(int argc, char *argv[])
    {
        FILE *z; q = argc;
        if ((z = fopen(argv[1], "r"))) {
            while ((b = getc(z)) > 0) *s++ = b;
            *s = 0; interpret(f);
        }
        return 0;
    }
```

Self-Interpreter can be short!

Writen by Oleg Mazonka & Daniel B. Cristofani

21 November 2003

```
>>>+[[-]>>[-]++>+>+++++++[<++++>>++<-]++>>+>+>+++++[>
++>++++++<<-]+>>>,<++[[>[->>]<[>>]<<-]<[<]<+>>[>]>[<+
>-[[<+>-]>]<[[[-]<]++<-[<+++++++++>[<->-]>>]>>]]<<]<]
<
[[<]>[[>]>>[>>]+[<<]<[<]<+>>-]>[>]+[->>]<<<<[[<<]<[<]
+<<[+>+<<-[>-->+<<-[>+<[>>+<<-]]]>[<+>-]<]++>>-->[>]>
>[>>]]<<[>>+<[[<]<]>[[<<]<[<]+[-<+>>-[<<+>++>-[<->[<<
+>>-]]]<[>+<-]>]>[>]>]>[>>]>>]<<[>>+>>+>>]<<[->>>>>>>>
>]<<[>.>>>>>>>>]<<[>->>>>>]<<[>,>>>]<<[>+>]<<[+<<]<]
```

Turing Complete (again)

- In fact, Brianfuck has 6 opcode + Input/Output commands
- gray area for I/O (implementation dependent)
  - EOF
  - tape length
  - cell type
  - newlines
- That is enough to program!
- Extension: self-modifying Brainfuck
  https://soulsphere.org/hacks/smbf/

Statement: while

- Implementing a while statement is easy, because the Brainfuck `[ .. ]` statement is a while loop.
- Thus, `while (x) { <foobar> }` becomes

```
(move pointer to a)
[
(foobar)
(move pointer to a)
]
```

Statement: x=y

- Implementing assignment (copy) instructions is complex.
- Straightforward way of doing that resets y to zero:

```
(move pointer to y) [ -
(move pointer to x) +
(move pointer to y) ]
```

- A temporary variable t is needed:

```
(move pointer to y) [ -
(move pointer to t) +
(move pointer to y) ]
(move pointer to t) [ -
(move pointer to x) +
(move pointer to y) +
(move pointer to t) ]
```

Statement: if

- The if statement is like a while-loop, but it should run its block only once. Again, a temporary variable is needed to implement if (x) { <foobar> }:

```
(move pointer to x) [ -
(move pointer to t) +
(move pointer to x) ]
(move pointer to t) [
    [ -
    (move pointer to x) +
    (move pointer to t) ]
    (foobar)
(move pointer to t) ]
```

Example: clean

```
[-]
```

```
while(cell[0]) {
  --cell[0];
}
```

Brainfuck	C
>	++p;
<	--p;
+	++*p;
–	--*p;
.	putchar(*p);
,	*p = getchar();
[	while (*p) {
]	}

Example: cat

```
+ [ , . ]
```

```
cell[0] ← 1
while(cell[0]) {
  Read-in a character
  print it
}
```

Brainfuck	C
>	++p;
<	--p;
+	++*p;
–	--*p;
.	putchar(*p);
,	*p = getchar();
[	while (*p) {
]	}

Example: if-endif

```
$f +
$A + [
        $B +    /* $B = $B + 1 */
        $f [-]  /* end if */
]
```

Brainfuck	C
>	++p;
<	--p;
+	++*p;
–	--*p;
.	putchar(*p);
,	*p = getchar();
[	while (*p) {
]	}

```
$A = 1;
if($A) {
   $B = $B + 1;
}
```

Example: if-else-endif

```
$f +
$A + [
        $B +    /* $B = $B + 1 */
        $f [-] /* end if */
] $f [
        $B -    /* $B = $B - 1 */
         $f [-]  /* end if */
]
```

```
$A = 1;
if($A) { ++$B; } else { --$B; }
```

Brainfuck	C
>	++p;
<	--p;
+	++*p;
–	--*p;
.	putchar(*p);
,	*p = getchar();
[	while (*p) {
]	}

Example: Multiply (6x7)

```
+++ +++
    [ > +++ +++ +
    < - ]
```

```
cell[0] ← 6
while(cell[0]) {
  cell[1] += 7;
  --cell[0];
}
```

Brainfuck	C
>	++p;
<	--p;
+	++*p;
–	--*p;
.	putchar(*p);
,	*p = getchar();
[	while (*p) {
]	}

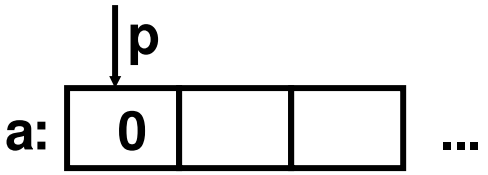
Example: Division (12/4)

```
++++++++++++    // Loop 12 times
[
>+              // Increment second cell
<----           // Subtract 4 from first cell
]
>.              // Display second cell's value
```

Example: Hello World!

```
+++++++[>+++++++++<-]>.     // 8 x 9 = 72 (H)
<+++++[>++++++<-]>-.        // 72 + (6 x 5) - 1 = 101 (e)
+++++++..                   // 101 + 7 = 108 (l)
+++.                        // 108 + 3 = 111 (o)
<++++++++[>>++++<<-]>>.     // 8 x 4 = 32 (SPACE)
<<++++[>------<-]>.         // 111 - 24 = 87 (W)
<++++[>++++++<-]>.          // 87 + 24 = 111 (o)
+++.                        // 111 + 3 = 114 (r)
------.                     // 114 - 6 = 108 (l)
--------.                   // 108 - 8 = 100 (d)
>+.                         // 32 + 1 = 33 (!)
```

Example: Hello World!



a[0]	a[1]	
0	9↑	
72	0↑	
72↑	0	H
101↑	0	e
108↑	0	ll
111↑	0	o
0↑	0	

32↑	0	␣
87↑	0	W
111↑	0	o
114↑	0	r
108↑	0	l
100↑	0	d
33↑	0	!
10↑	0	\n

```
>++++++++[<++++++++>-]<.>+++++++[<++++>-]<+.
+++++++..+++.[-]>++++++++[<++++>-]<.
>++++++++++[<+++++>-]<.>++++++++[<+++>-]<.
+++.------.--------.[-]>++++++++[<++++>-]<+.
[-]++++++++++.
```

Example: Bubble Sort

```
>>>>>,.[>>>,.]
<<<
[<<<
[>>>
      [-<<<-<+>[>]>>]
      <<<[<]>>
      [>>>+<<<-]<
      [>+>>>+<<<<-]
<<]
>>>[.[-]]
>>>[>>>]<<<]
```

0	51 3	0	0	49 1	0	0	50 2	0	0	0

This implementation of the Turing Machine uses a Brainfuck program to define its behaviour.
Write a program in the box below and click Run to execute it.

```
>>>>>,.[>>>,. ]
<<<
[<<<
[>>>
[-<<<-<+>[>]>>]
<<<[<]>>
[>>>+<<<-]<
[>+>>>+<<<<-]
<<]
>>>[.[-]]
>>>[>>>]<<<]
```

Output

312

Modify Input...

Run Program

Example: Bubble Sort

```
>>>>>,.[>>>,.]
<<<
[<<<
[>>>
      [-<<<-<+>[>]>>]
      <<<[<]>>
      [>>>+<<<-]<
      [>+>>>+<<<<-]
<<]
>>>[.[-]]
>>>[>>>]<<<]
```

0	51	0	49	0	0	0	1	0	0
	3		1						

This implementation of the Turing Machine uses a Brainfuck program to define its behaviour.
Write a program in the box below and click Run to execute it.

```
>>>>>,.[>>>,. ]
<<<
[<<<
[>>>
[-<<<-<+>[>]>>]
<<<[<]>>
[>>>+<<<-]<
[>+>>>+<<<<-]
<<]
>>>[.[-]]
>>>[>>>]<<<]
```

Output

312

Modify Input...

Idea: `if(b>a): swap(a, b)`
Operation: decrement `$a` and `$b`. Then, store the smaller one into `$t`

Example: Bubble Sort

```
>>>>>,.[>>>,.]
<<<
[<<<
[>>>
      [-<<<-<+>[>]>>]
      <<<[<]>>
      [>>>+<<<-]<
      [>+>>>+<<<<-]
<<]
>>>[.[-]]
>>>[>>>]<<<]
```

0	51	0	49	0	0	0	1	0	0	0
	3		1							

This implementation of the Turing Machine uses a Brainfuck program to define its behaviour.
Write a program in the box below and click Run to execute it.

```
>>>>>,.[>>>,. ]
<<<
[<<<
[>>>
[-<<<-<+>[>]>>]
<<<[<]>>
[>>>+<<<-]
```

Output

312

Modify Input...

Idea: `if(a>b): swap(a, b)`
Operation: when `b > a`, assign the value of `$b` to `$a`

Example: Bubble Sort

```
>>>>>,.[>>>,.]
<<<
[<<<
[>>>
      [-<<<-<+>[>]>>]
      <<<[<]>>
      [>>>+<<<-]<
      [>+>>>+<<<<-]
<<]
>>>[.[-]]
>>>[>>>]<<<]
```

0	51 3	0	0	49 1	0	0	50 2	0	0	0

This implementation of the Turing Machine uses a Brainfuck program to define its behaviour.
Write a program in the box below and click Run to execute it.

```
>>>>>,.[>>>,. ]
<<<
[<<<
[>>>
[-<<<-<+>[>]>>]
<<<[<]>>
[>>>+<<<-]<
[>+>>>+<<<<-]
<<]
>>>[.[-]]
>>>[>>>]<<<]
```

Output

312

Modify Input...

Idea: `if(b>a): swap(a, b)`

Example: Bubble Sort

```
>>>>>,.[>>>,.]
<<<
[<<<
[>>>
      [-<<<-<+>[>]>>]
      <<<[<]>>
      [>>>+<<<-]<
      [>+>>>+<<<<-]
<<]
>>>[.[-]]
>>>[>>>]<<<]
```

0	49 1	0	0	51 3	0	0	50 2	0	0	0

This implementation of the Turing Machine uses a Brainfuck program to define its behaviour.
Write a program in the box below and click Run to execute it.

```
>>>>>,.[>>>,. ]
<<<
[<<<
[>>>
[-<<<-<+>[>]>>]
<<<[<]>>
[>>>+<<<-]<
[>+>>>+<<<<-]
<<]
>>>[.[-]]
>>>[>>>]<<<]
```

Output

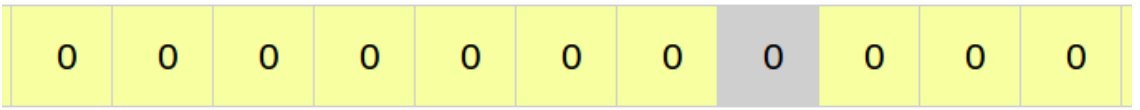
3121

Modify Input...

Run Program

Example: Bubble Sort

```
>>>>>,.[>>>,.]
<<<
[<<<
[>>>
      [-<<<-<+>[>]>>]
      <<<[<]>>
      [>>>+<<<-]<
      [>+>>>+<<<<-]
<<]
>>>[.[-]]
>>>[>>>]<<<]
```



This implementation of the Turing Machine uses a Brainfuck program to define its behaviour.
Write a program in the box below and click Run to execute it.

```
>>>>>,.[>>>,. ]
<<<
[<<<
[>>>
[-<<<-<+>[>]>>]
<<<[<]>>
[>>>+<<<-]<
[>+>>>+<<<<-]
<<]
>>>[.[-]]
>>>[>>>]<<<]
```

Output

312123

Modify Input...

Run Program

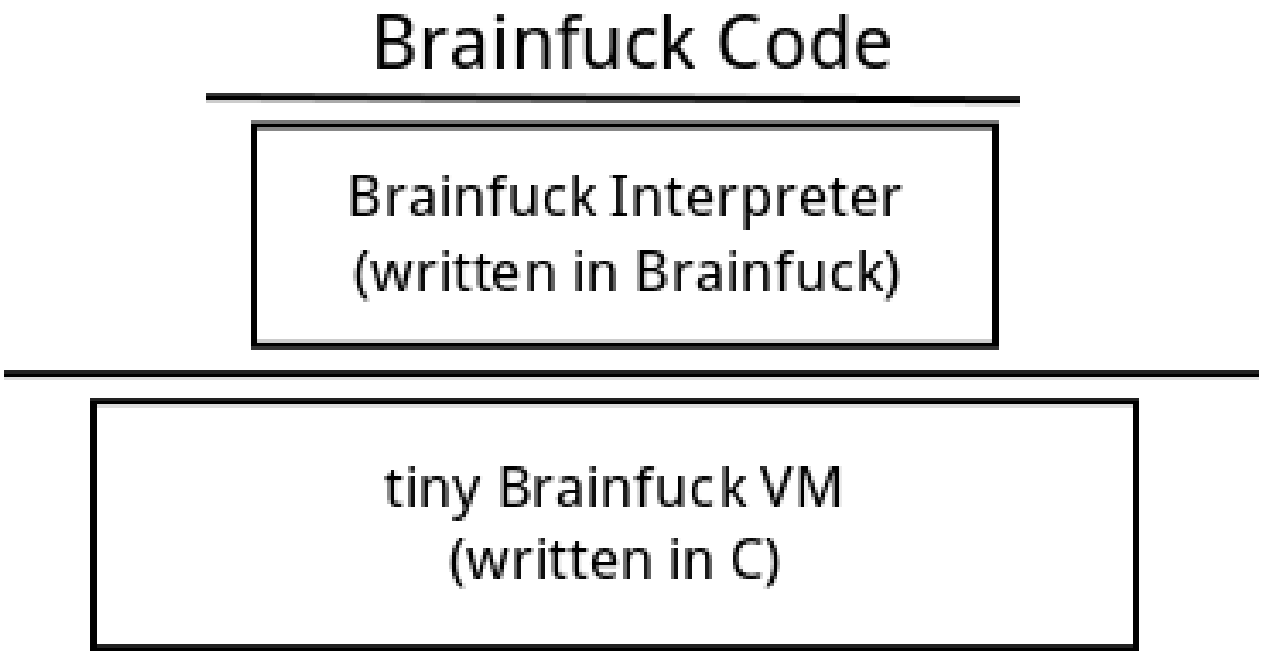
Online interpreter: http://work.damow.net/random/bf-turing/

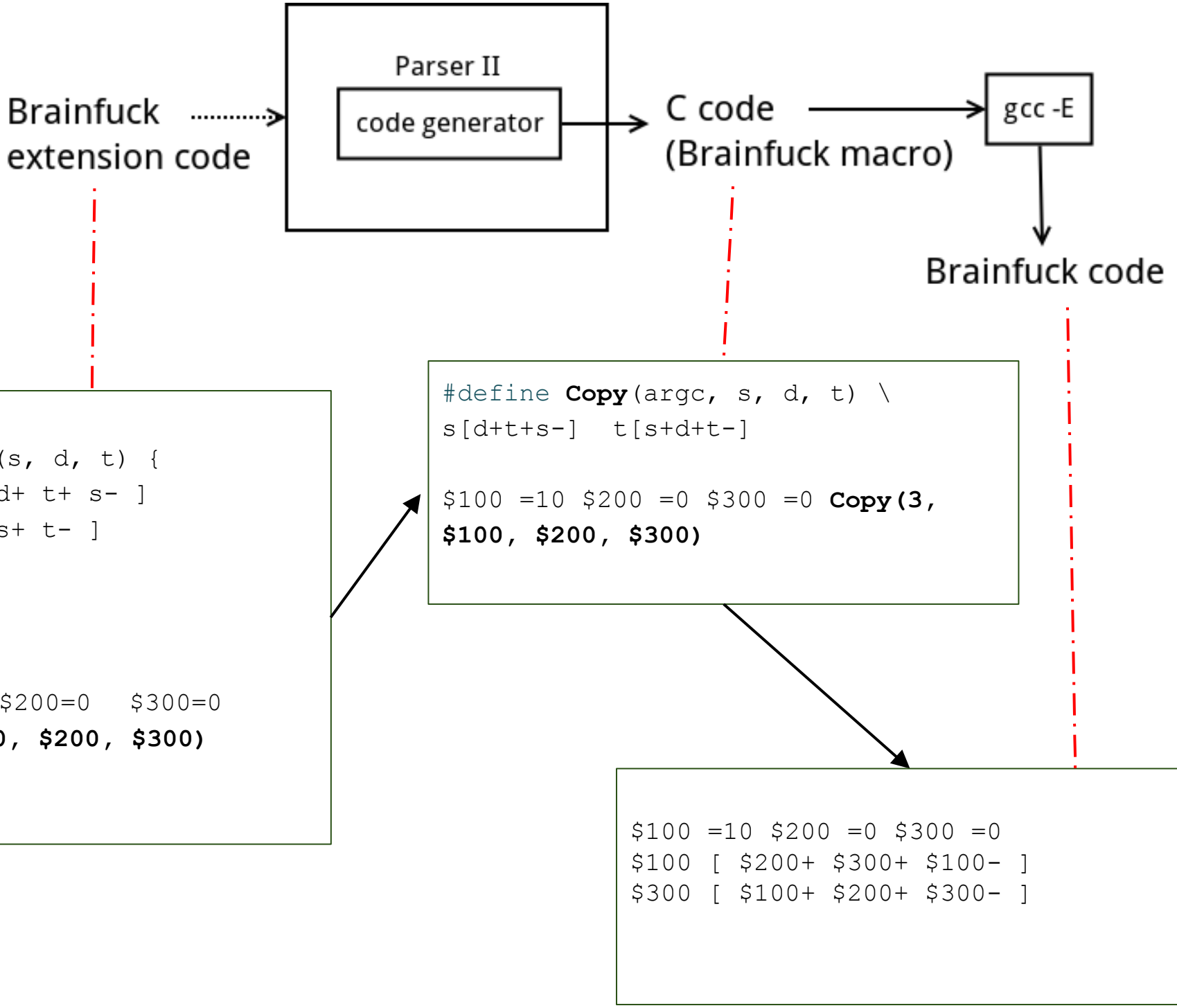
Brainfuck Toolchain

:: interpreter, translator, virtual machine, nested runtime ::

Nested Interpreting

- Translation:
  - BF extensions → Brainfuck
  - Other languages → Brainfuck
- Interpreter written in Brainfuck runs on BF VM




Brainfuck
extension code
Parser II
code generator
C code
(Brainfuck macro)
gcc -E
Brainfuck code
DEF
def Copy(s, d, t) {
s [d+ t+ s-]
t [s+ t-]
}
END
MAIN
$100=10 $200=0 $300=0
Copy($100, $200, $300)
#define Copy(argc, s, d, t) \
s[d+t+s-] t[s+d+t-]
$100 =10 $200 =0 $300 =0 Copy(3,
$100, $200, $300)
$100 =10 $200 =0 $300 =0
$100 [$200+ $300+ $100-]
$300 [$100+ $200+ $300-]

Brainfuck translator

(use BF as the backend)

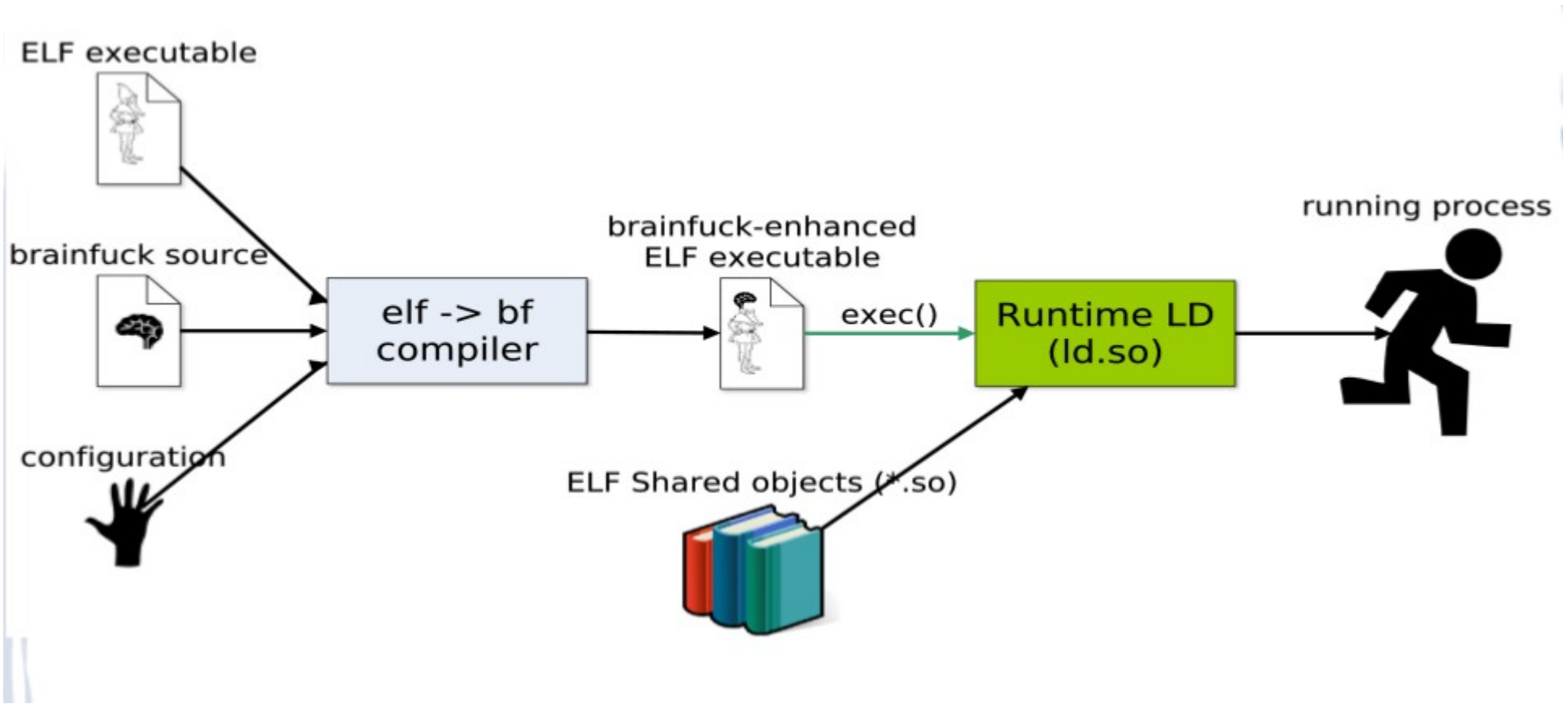
- Translate C-like language to Brainfuck
  - bfc [C] → bfa [Assembly] → bf [Machine/CPU]

    http://www.clifford.at/bfcpu/bfcomp.html
- Another C to Brainfuck

  http://esolangs.org/wiki/C2BF
- BASIC to Brainfuck

  http://esolangs.org/wiki/BFBASIC

Compile Brainfuck into ELF



Using Artificial Intelligence to Write Self-Modifying/Improving Programs

- AI program works, as follows:
  - A genome consists of an array of doubles.
  - Each gene corresponds to an instruction in the brainf-ck programming language.
  - Start with a population of random genomes.
  - Decode each genome into a resulting program by converting each double into its corresponding instruction and execute the program.
  - Get each program's fitness score, based upon the output it writes to the console (if any), and rank them.
  - Mate the best genomes together using roulette selection, crossover, and mutation to produce a new generation.
  - Repeat the process with the new generation until the target fitness score is achieved.

Source: http://www.primaryobjects.com/CMS/Article149

Runtime Optimizations

Mandelbrot

Incremental optimizing interpreter

https://github.com/xatier/brainfuck-tools

https://github.com/xatier/brainfuck-bench

Interpreter vs. Static Compiler

Implementation	(user-space) Execution Time (in second)
simple bf	91.50
slight optimizations	8.03
bff	5.04
vff	3.10
vm + optimizations	3.02
BF-JIT	17.78
BF-JIT + optimizations	1.37
simple xbyak JIT	3.25
xbyak JIT + optimizations	0.93
custom JIT + aggressive optimizations	0.77
Simple lightning JIT	1.27

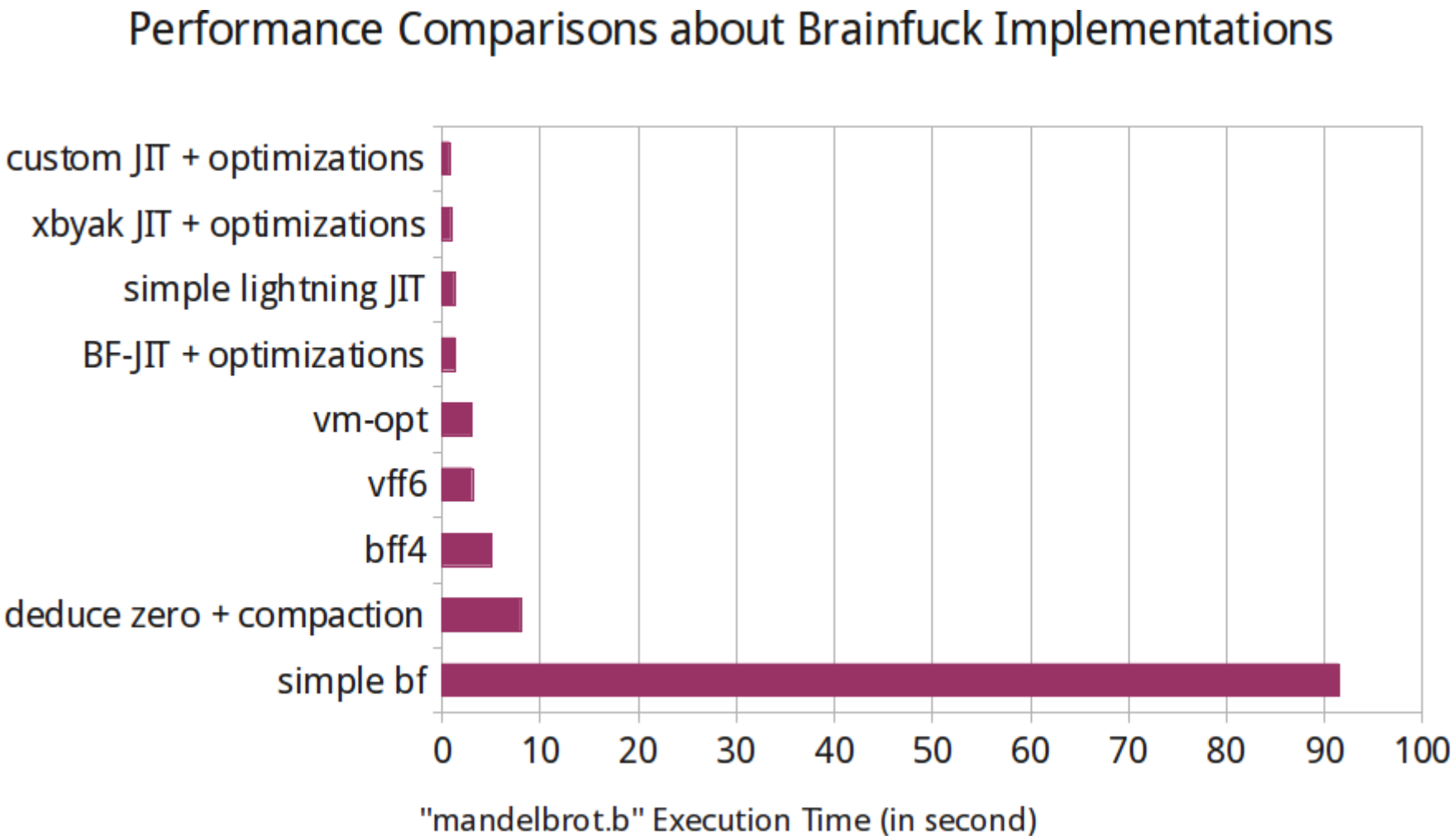
11x speedup!

Implementation	(user-space) Execution Time (in second)
simple BF to C	1.27
awib to C	1.05
esotope-bfc	0.72
bftran to C	0.66
bftran to ELF32c	3.58

The executable generated by static compiler (2 pass: BF → C → x86_64) is likely slower than optimized interpreters.

Plain JIT compilation without effective optimizations is slower than portable interpreters!

The fastest interpreter record appears on Lenovo X230 [Intel(R) Core(TM) i5-3320M CPU @ 2.60GHz].


Performance Comparisons about Brainfuck Implementations
custom JIT + optimizations
xbyak JIT + optimizations
simple lightning JIT
BF-JIT + optimizations
vm-opt
vff6
bff4
deduce zero + compaction
simple bf
0
10
20
30
40
50
60
70
80
90
100
"mandelbrot.b" Execution Time (in second)

Walk through typical Design Patterns

- Classify the executions of Brainfuck programs
- Eliminate the redundant
  - CSE: common sub-expression elimination
- Quick instructions
- Hotspot
  - Replace with faster implementation
- Enable Just-In-Time compilation

Pattern: +++

++++++++++

*ptr += 10

Pattern: >>>

>>>>>>>>>>

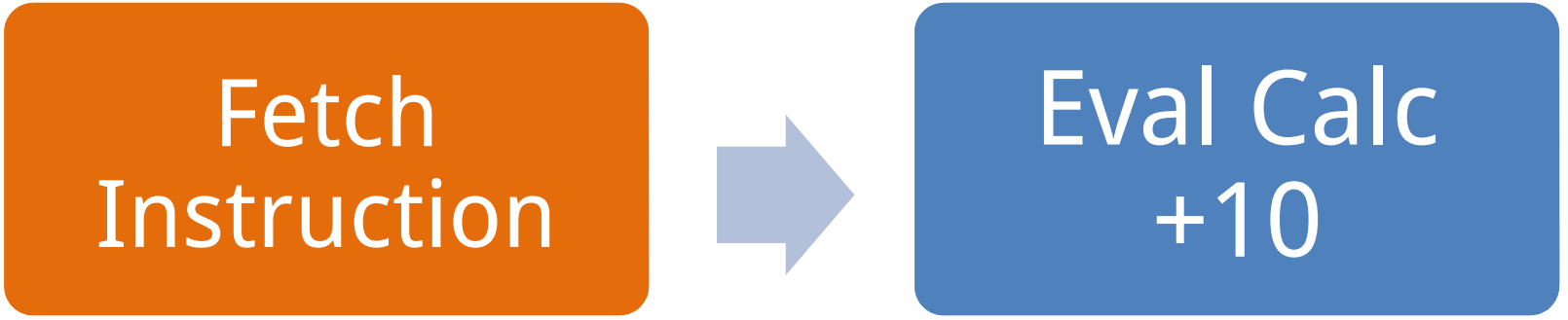
ptr += 10

Pattern: +++

++++++++++

*ptr += 10

10 Instructions



1 instruction

Pattern: [-]

[-]

*ptr = 0

Pattern: [>+<-]

[>+<-]

*(ptr+1) += *ptr

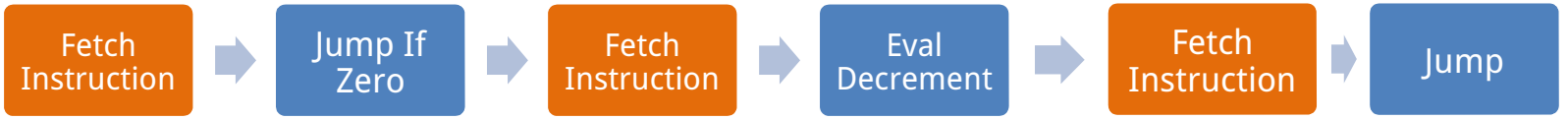
*ptr = 0

Pattern: [-]

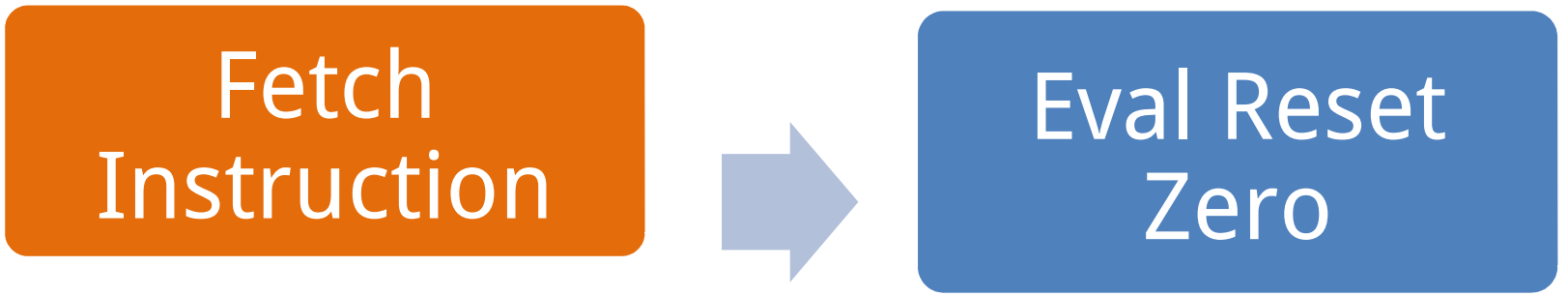
[-]

Interpret:

if(!*ptr) goto];--*ptr;goto [;



Contains Branch



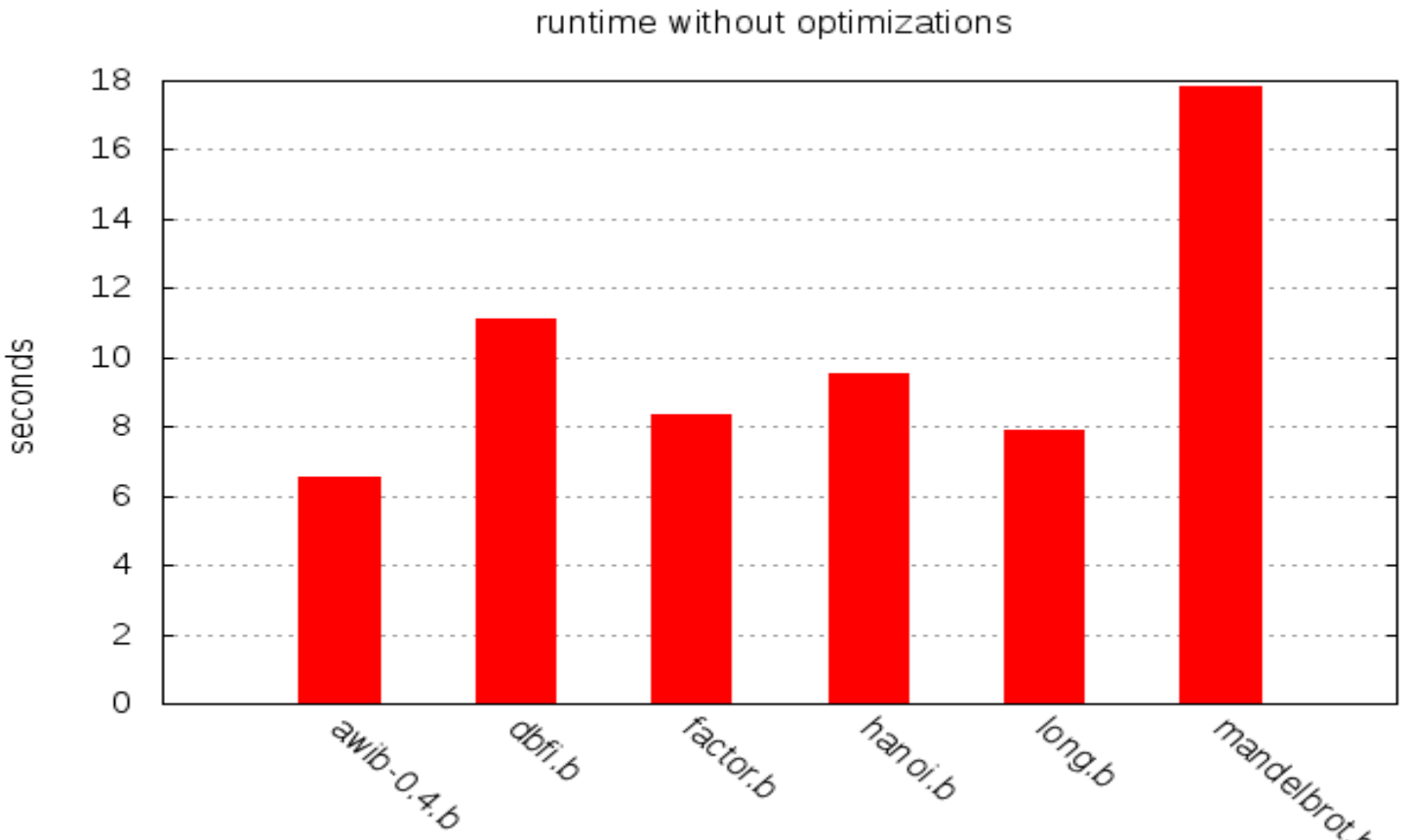
1 Instruction, No Branch

Optimization Techniques

- To evaluate the impact different optimization techniques can have on performance, we need a set of Brainfuck programs that are sufficiently non-trivial for optimization to make sense.
  - awib-0.4 (Brainfuck compiler)
  - factor.b
  - mandelbrot.b
  - hanoi.b
  - dbfi.b (self-interpreter)
  - long.b
- source: `https://github.com/matslina/bfoptimization`

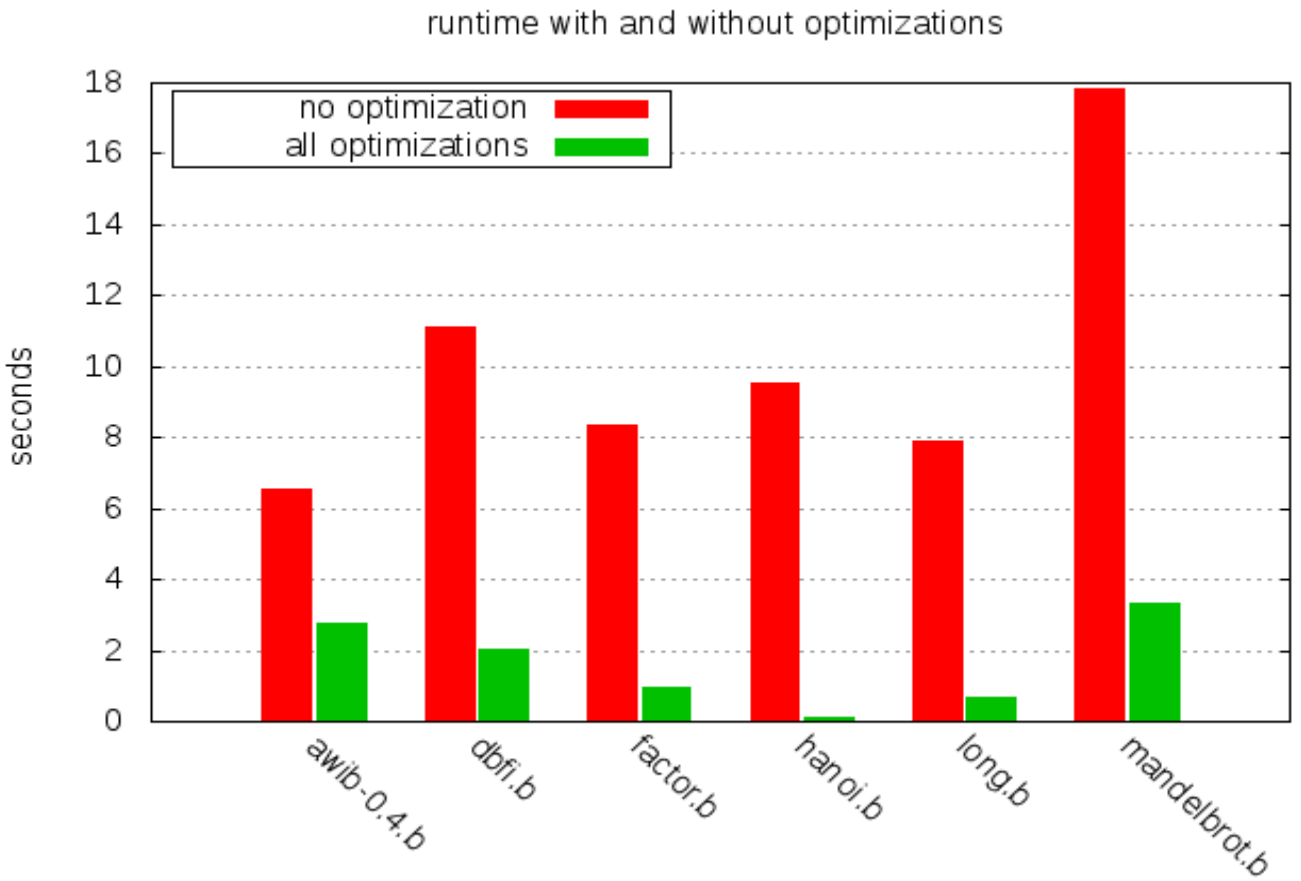
Benchmark Results

(without optimizations)



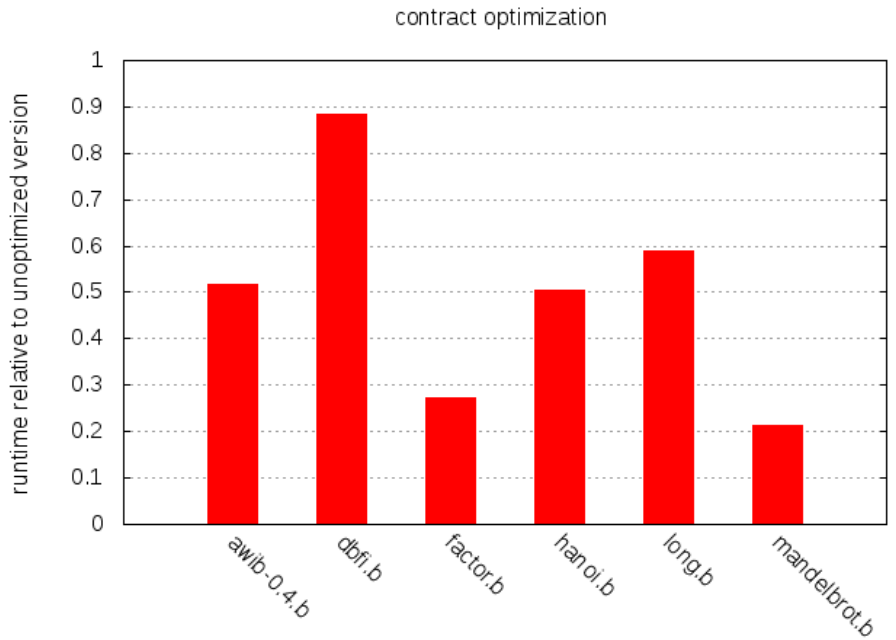
Benchmark Results

(w/ and w/o optimizations)



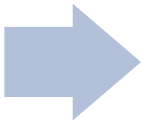
Benchmark Results

(Contraction)




```
+++++[->>>++<<<]>>>.
```

```
mem[p]++;
mem[p]++;
mem[p]++;
mem[p]++;
mem[p]++;
while (mem[p]) {
    mem[p]--;
    p++;
    p++;
    p++;
    mem[p]++;
    mem[p]++;
    p--;
    p--;
    p--;
}
p++;
p++;
p++;
putchar(mem[p]);
```

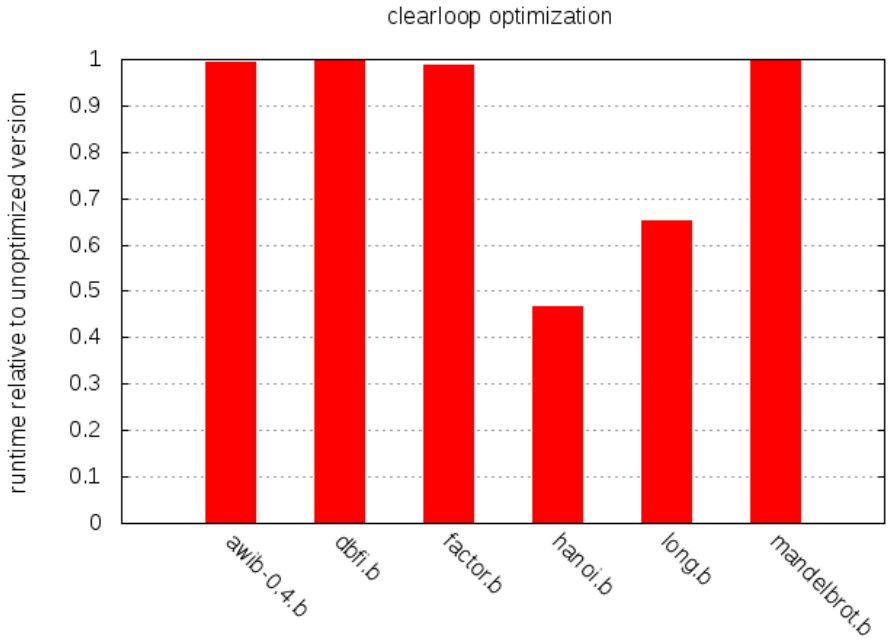


```
mem[p] += 5;
while (mem[p]) {
    mem[p] -= 1;
    p += 3;
    mem[p] += 2;
    p -= 3;
}
p += 3;
putchar(mem[p]);
```

IR	C
add(x)	mem[p] += x;
sub(x)	mem[p] -= x;
right(x)	p += x;
left(x)	p -= x;
output	putchar(mem[p]);
input	mem[p] = getchar();
open_loop	while(mem[p]) {
close_loop	}

Benchmark Results

(Clear loops)



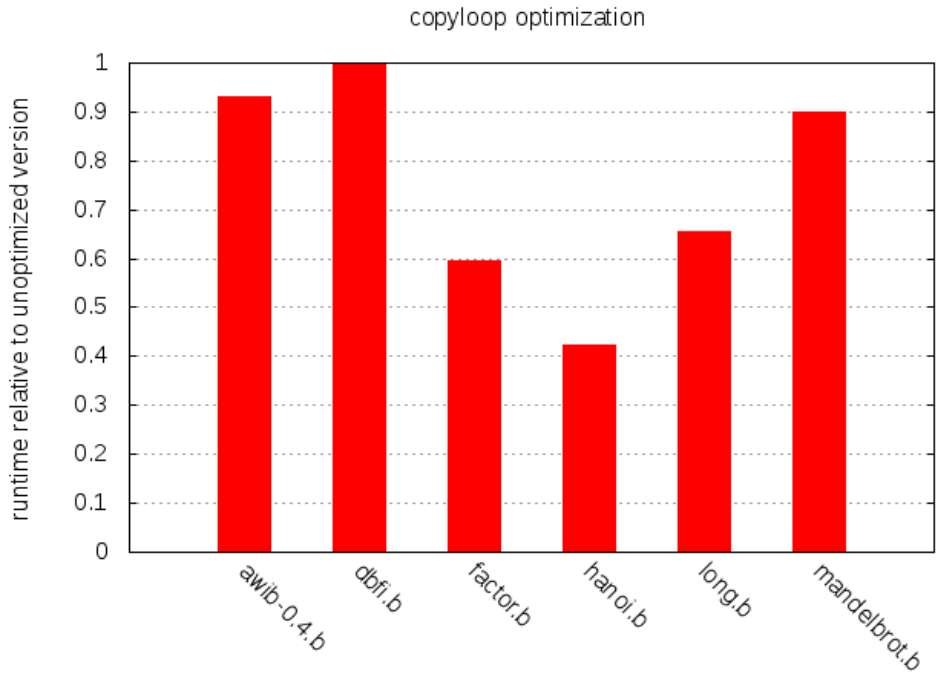

```
[ - ]
```

IR	C
add	mem[p]++;
sub	mem[p]--;
right	p++
left	p--
output	putchar(mem[p]);
input	mem[p] = getchar();
open_loop	while(mem[p]) {
close_loop	}
clear	mem[p] = 0;

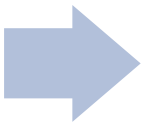
Eval Reset Zero

Benchmark Results

(Copy loops)




```
[->+>+<<])
```

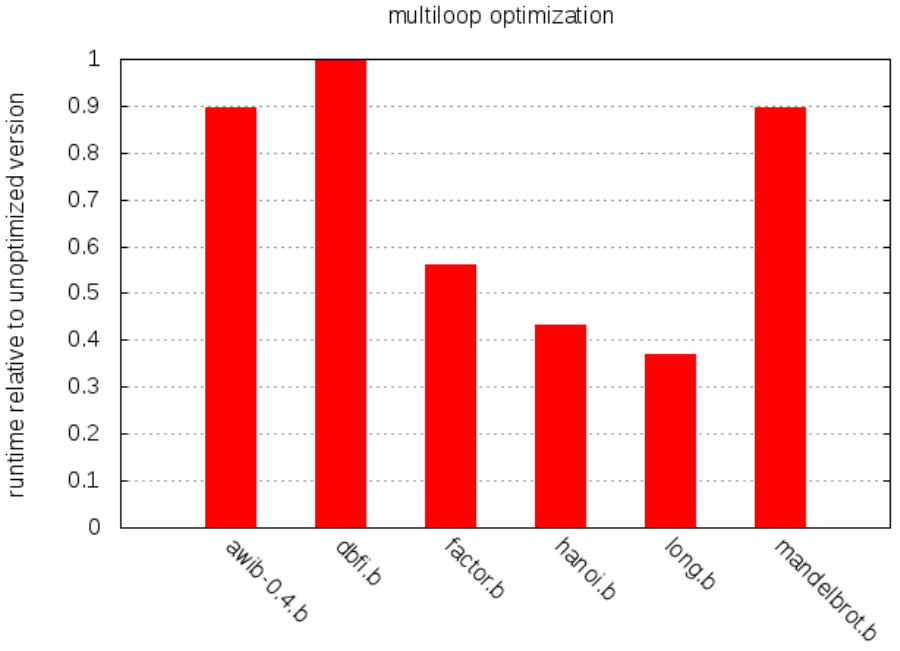


```
mem[p+1] += mem[p];
mem[p+2] += mem[p];
mem[p] = 0;
```

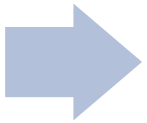
IR	C
add(x)	mem[p] += x;
sub(x)	mem[p] -= x;
right(x)	p += x;
left(x)	p -= x;
output	putchar(mem[p]);
input	mem[p] = getchar();
open_loop	while(mem[p]) {
close_loop	}
clear	mem[p] = 0;
copy(x)	mem[p+x] += mem[p];

Benchmark Results

(Multiplication loops)




```
[->+++>++++++++<<]
```

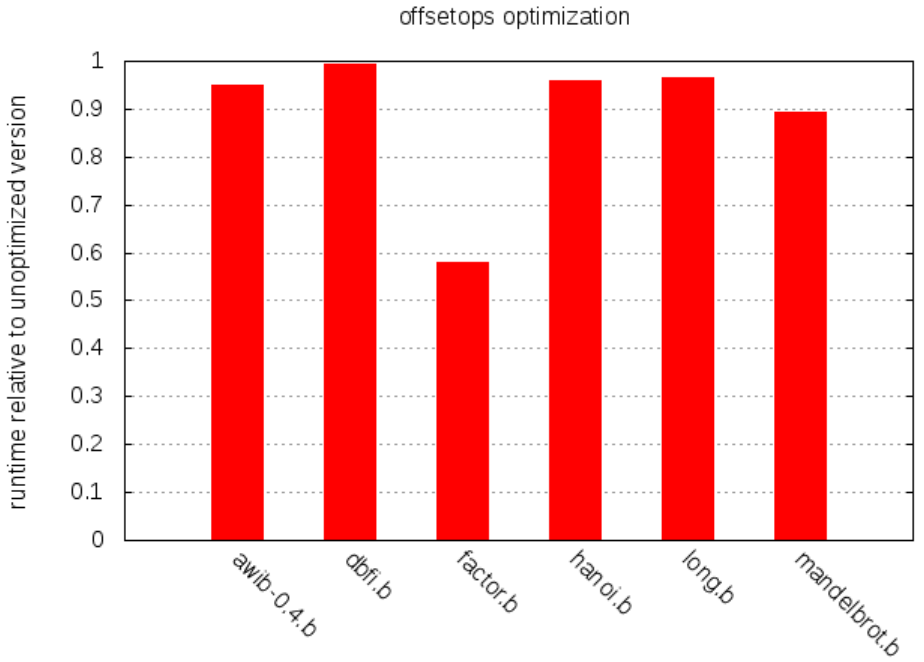


```
mem[p+1] += mem[p] * 3;
mem[p+2] += mem[p] * 7;
mem[p] = 0
```

IR	C
add(x)	mem[p] += x;
sub(x)	mem[p] -= x;
right(x)	p += x;
left(x)	p -= x;
output	putchar(mem[p]);
input	mem[p] = getchar();
open_loop	while(mem[p]) {
close_loop	}
clear	mem[p] = 0;
mul(x,y)	mem[p+x] += mem[p] * y;

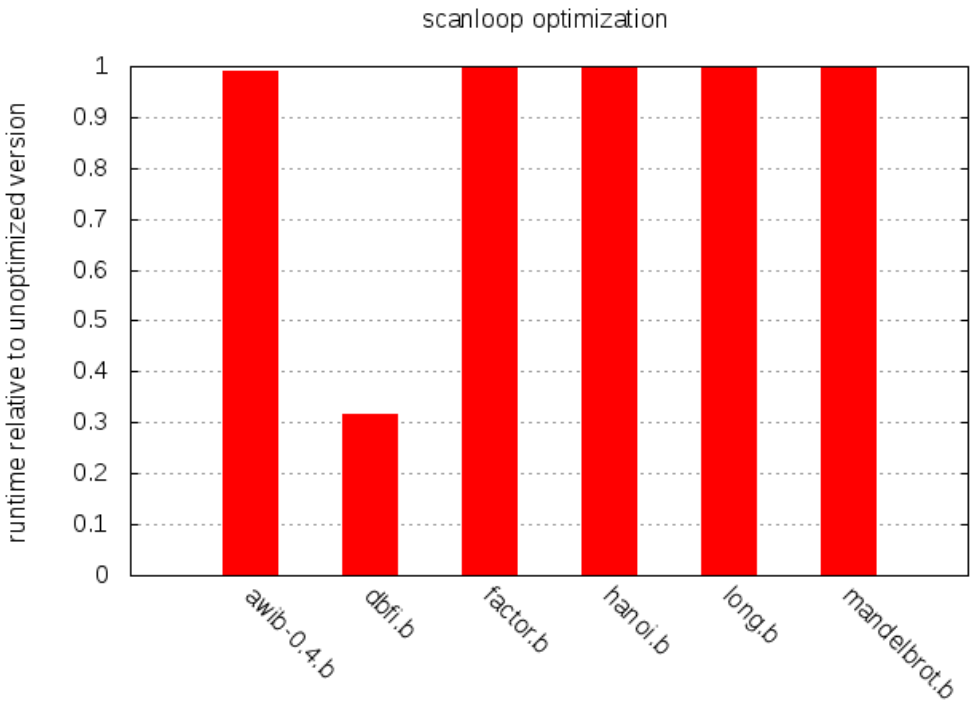
Benchmark Results

(Operation offsets)



Both the copy loop and multiplication loop optimizations share an interesting trait: they perform an arithmetic operation at an offset from the current cell. In brainfuck we often find long sequences of non-loop operations and these sequences typically contain a fair number of < and >. Why waste time moving the pointer around?

IR	C
add(x,off)	mem[p+off] += x;
sub(x,off)	mem[p+off] -= x;
right(x)	p++
left(x)	p--
output	putchar(mem[p+off]);
input	mem[p+off] = getchar();
open_loop	while(mem[p]) {
close_loop	}
clear	mem[p+off] = 0;
mul(x,y)	mem[p+x+off] += mem[p+off] * y;




```
+< [ > - ] > [ > ] <
```

The problem of efficiently searching a memory area for occurrences of a particular byte is mostly solved by the C standard library's memchr() function, which operates by loading full memory words (typically 32 or 64 bits) into a CPU register and checking the individual 8-bit components in parallel. This proves to be much more efficient than loading and inspecting bytes one at a time.

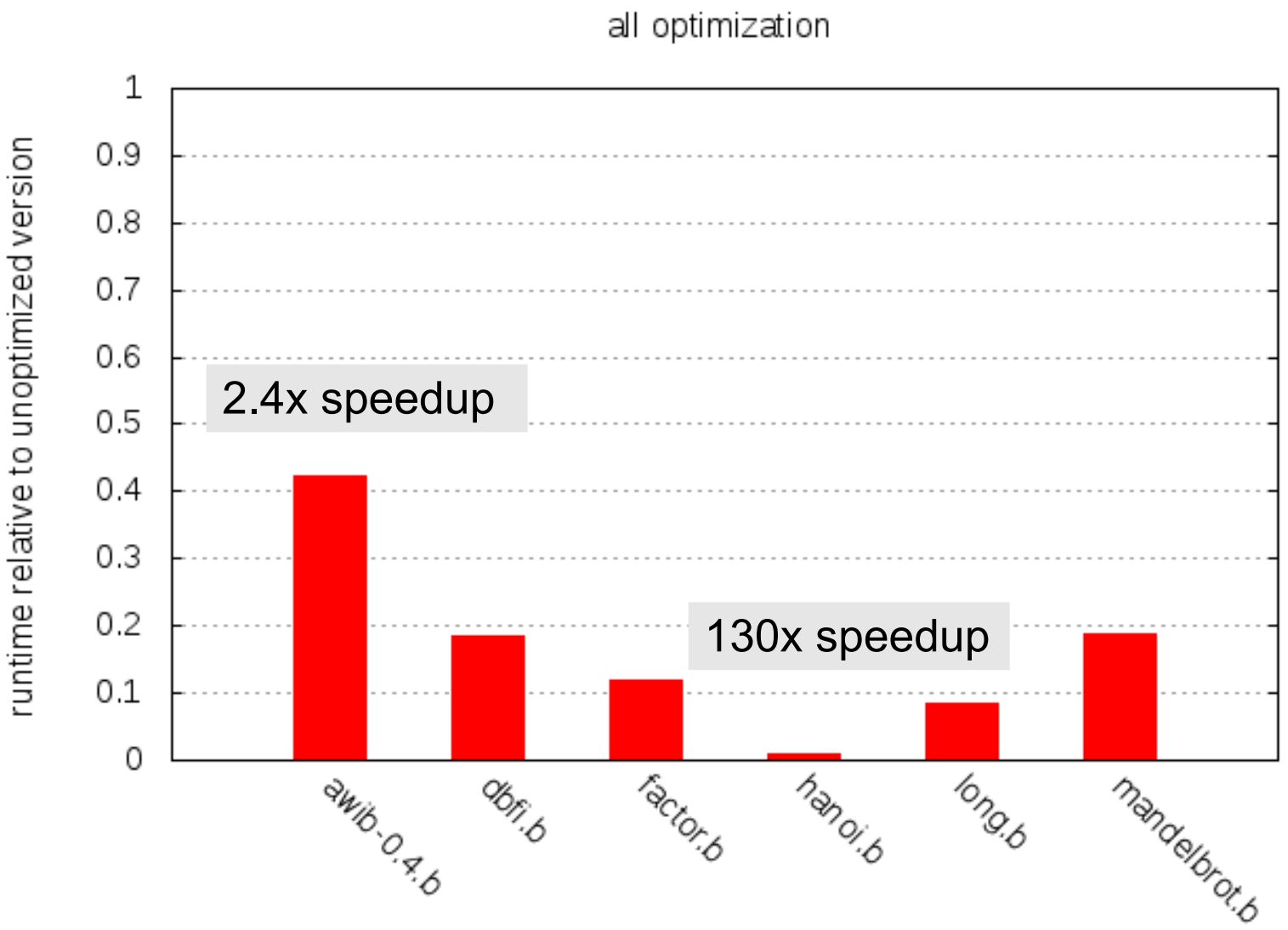
Benchmark Results

(Scan loops)

IR	C
add(x)	mem[p] += x;
sub(x)	mem[p] -= x;
right(x)	p += x;
left(x)	p -= x;
output	putchar(mem[p]);
input	mem[p] = getchar();
open_loop	while(mem[p]) {
close_loop	}
clear	mem[p] = 0;
mul(x,y)	mem[p+x] += mem[p] * y;
ScanLeft	p -= (long)((void *)(mem + p) - memrchr(mem, 0, p + 1));
ScanRight	p += (long)(memchr(mem + p, 0, sizeof(mem)) - (void *)(mem + p));

Benchmark Results

(apply all techniques)



Reference

1. Principles of Compiler Design: The Brainf*ck Compiler
   http://www.clifford.at/papers/2004/compiler/
2. brainfuck optimization strategies
   http://calmerthanyouare.org/2015/01/07/optimizing-brainfuck.html
3. Brainf*ck Compiler Project
   http://www.clifford.at/bfcpu/bfcomp.html
4. Brainfuck code generation
   http://esolangs.org/wiki/Brainfuck_code_generation