HARVARD
John A. Paulson
School of Engineering
and Applied Sciences

Dynamic Analysis

CS252r Fall 2015

Reading

- *The Concept of Dynamic Analysis* by Thomas Ball, FSE 1999
- *Efficient Path Profiling* by Thomas Ball and James Larus, MICRO 1996
- *Adversarial Memory for Destructive Data Races* by Cormac Flanagan and Stephen Freund, PLDI 2010

Dynamic Analysis

- Analysis of the properties of a running program
- Static analysis typically finds properties that hold of **all** executions
- Dynamic analysis finds properties that hold of **one or more** executions
  - Can't prove a program satisfies a particular property
  - But can detect violations and provide useful information
- Usefulness derives from **precision of information** and **dependence on inputs**

Precision of Information

- Dynamic analysis typically instructs program to examine or record some of run-time state
- Instrumentation can be tuned to precisely data needed for a problem

Dependence on Program Inputs

- Easy to relate changes in program inputs to changes in program behavior and program output

Dynamic Analyses
are
input-centric

Static Analyses
are
program-centric

Complementary Techniques

- Completeness
  - Dynamic analyses can generate "dynamic program invariants", i.e., invariants of observed execution; static analyses can check them
  - Dynamic analyses consider only feasible paths (but may not consider all paths); static analyses consider all paths (but may include infeasble paths)
- Scope
  - Dynamic analyses examine one very long program path
    - Can discover semantic dependencies widely separated in path and in time
    - Static analyses typically and at discovering "dependence at a distance"
- Precision

Two (plus a bonus) Dynamic Analyses

- Frequency Spectrum Analysis
  - Efficient path profiling
- Dynamic race detection

Frequency Spectrum Analysis

- Understanding frequency of execution of program parts can help programmer:
  - partition program by levels of abstraction
  - find related computations
  - find computations related to specific attributes of input or output

Understanding an Obfuscated C Program

```
#include <stdio.h>
main(t,_,a)
char *a;
{
return!0<t?t<3?main(-79,-13,a+main(-87,1-_,main(-86,0,a+1)+a)):
1,t<_?main(t+1,_,a):3,main(-94,-27+t,a)&&t==2?_<13?
main(2,_+1,"%s %d %d\n"):9:16:t<0?t<-72?main(_,t,
"@n'+,#'/*{}w+/w#cdnr/+,{}r/*de}+,/*{*+,/w{%+,/w#q#n+,/#{l+,/n{n+,/+#n+,/#\
;#q#n+,/+k#;*+,/'r :'d*'3,}{w+K w'K:'+}e#';dq#'l \
q#'+d'K#!/+k#;q#'r}eKK#}w'r}eKK{nl]'/#;#q#n'){)#}w'){){nl]'/+#n';d}rw' i;#\
){nl]!/n{n#'; r{#w'r nc{nl]'/#{l,+'K {rw' iK{;[{nl]'/w#q#n'wk nw' \
iwk{KK{nl]!/w{%'l##w#' i; :{nl]'/*{q#'ld;r'}{nlwb!/*de}'c \
;;{nl'-{}rw]'/+,}##'*}#nc,',#nw]'/+kd'+e}+;#'rdq#w! nr'/ ') }+}{rl#'{n' ')# \
}'+}##(!!/")
:t<-50?_==*a?putchar(31[a]):main(-65,_,a+1):main((*a=='/')+t,_,a+1)
:0<t?main(2,2,"%s"):*a=='/'||main(0,main(-61,*a,
"!ek;dc i@bK'(q)-[w]*%n+r3#l,{}:\nuwloca-O;m .vpbks,fxntdCeghiry"),a+1);
}
```

What it does...

```
$ gcc -w obfus.c
$ ./a.out
On the first day of Christmas my true love gave to me
a partridge in a pear tree.

On the second day of Christmas my true love gave to me
two turtle doves
and a partridge in a pear tree.

On the third day of Christmas my true love gave to me
three french hens, two turtle doves
and a partridge in a pear tree.
```

Program understanding

- We know **what** the program does
- Our aim is to understand **how** it does it
- Before reverse engineering it, let's have a model in mind:
  - Gift t mentioned 13-t times in the poem (e.g. "five gold rings" occurs 13-5=8 times)
  - So 1+2+...+11+12 = 13*6 = 78 gift mentions (66 mentions of non-partridge gifts)
  - All verses except first have form

    On the <ordinal> day of Christmas my true love gave to me
    <list of gift phrases, from the ordinal day down to the second day>
    and a partridge in a pear tree.

    and first verse is

    On the first day of Christmas my true love gave to me
    a partridge in a pear tree.
  - Unique strings:
    - 3 strings for common structure ("On the", "day of Christmas...", "and a partridge ...")
    - 12 strings for the ordinals
    - 11 strings for the second through twelfth gifts.
    - ⇒approx. 3+12+11 = 26 unique strings in program, prints approx. 3*12 + 12 + 66 = 114 strings.

Model

- 12 days of Christmas (also 11, to catch "off-by-one" cases)
- 26 unique strings
- 66 occurrences of non-partridge-in-a-pear-tree presents
- 114 strings printed, and
- 2358 characters printed.

Program understanding

- First let's make it readable:

It's a single recursive function!

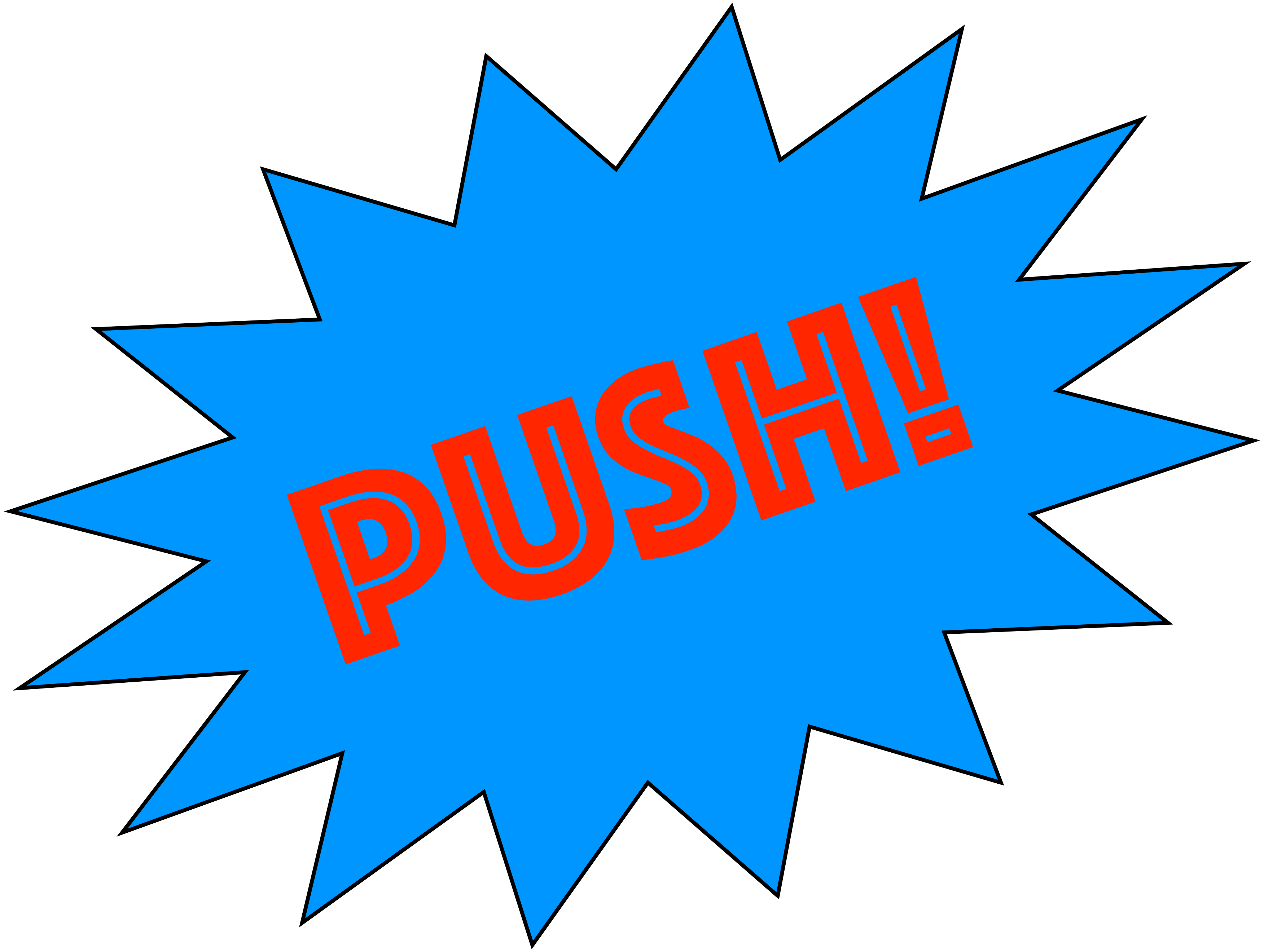
```
#include <stdio.h>
main(t,_,a) char *a;
{
     if ((!0) < t) {
[1]    if (t < 3)  main(-79,-13,a+main(-87,1-_,main(-86,0,a+1)+a));
[2]    if (t < _ ) main(t+1,_,a);
[3]    main(-94,-27+t,a);
[4]    if (t==2 && _ < 13 ) main(2,_+1,"");
     } else if (t < 0) {
[5]    if (t < -72)  main(_,t,LARGE_STRING);
       else if (t < -50 ) {
[6]      if (_ == *a) putchar(31[a]);
[7]      else         main(-65,_,a+1);
[8]    } else main((*a=='/')+t,_,a+1);
[9]  } else if (0 < t) main (2,2,"%s");
[10] else if (*a!='/') main(0,main(-61,*a,SMALL_STRING),a+1);
}
```

Path Profiling

- Count executions of paths of the function
  - E.g., path executed 2358 times likely involved in printing characters

Path ID	Frequency	Condition	Call Lines
main:0	1	`t == 1`	[9]
main:19	1	`t==2 && t >= _`	[1,3,4]
main:22	1	`t==2 && t < _ && _ >= 13`	[1,2,3]
main:23	10	`t==2 && t < _ && _ < 13`	[1,2,3,4]
main:9	11	`t >= 3 && t >= _`	[3]
main:13	55	`t >= 3 && t < _`	[2,3]
main:2	114	`t == 0 && *a == '/'`	*no call lines*
main:3	114	`t < -72`	[5]
main:1	2358	`t == 0 && *a != '/'`	[10]
main:7	2358	`t > -72 && t < -50 && _ == *a`	[6]
main:4	24931	`t < 0 && t >= -50`	[8]
main:5	39652	`t > -72 && t < -50 && _ != *a`	[7]

Table 2. Summary of the twelve executed paths in the readable obfuscated C program of Figure 2.

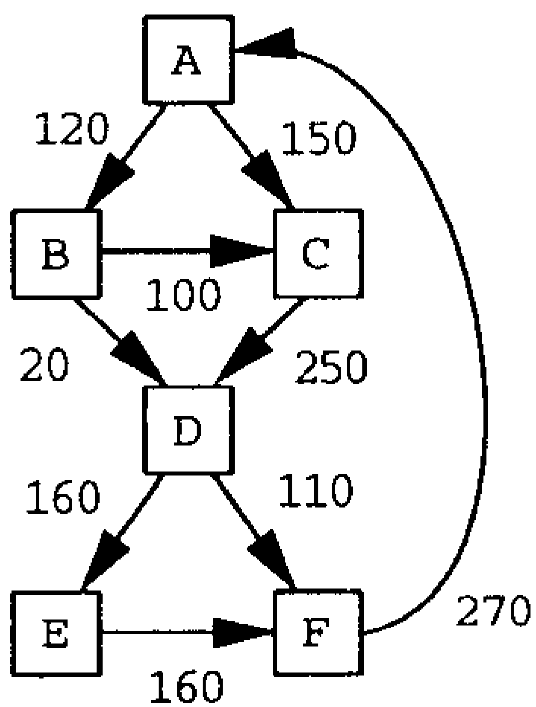


Path Profiling

Efficient Path Profiling, Ball and Larus, MICRO 1996

Problem: path profiling

- Which **paths** through a procedure are most common?
  - e.g., perform aggressive optimization on hot paths, make sure all paths are tested.
- Naive approach: count edge transitions

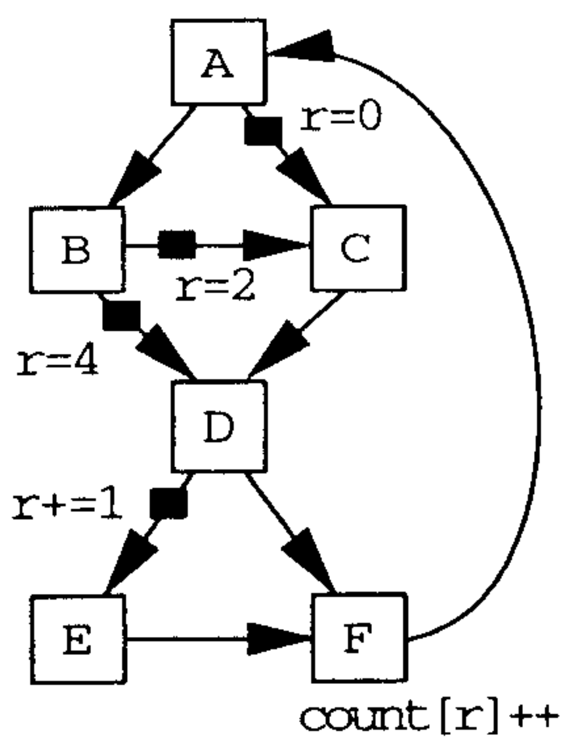


Path	Prof1	Prof2
ACDF	90	110
ACDEF	60	40
ABCDF	0	0
ABCDEF	100	100
ABDF	20	0
ABDEF	0	20

- Not enough information to determine paths!

Efficient Path Profiling

- (For DAGs)
- Encode each path as a unique integer and record path as state
  - i.e., at end of DAG, value of a register identifies path through DAG



Path	Encoding
ACDF	0
ACDEF	1
ABCDF	2
ABCDEF	3
ABDF	4
ABDEF	5

Algorithm overview

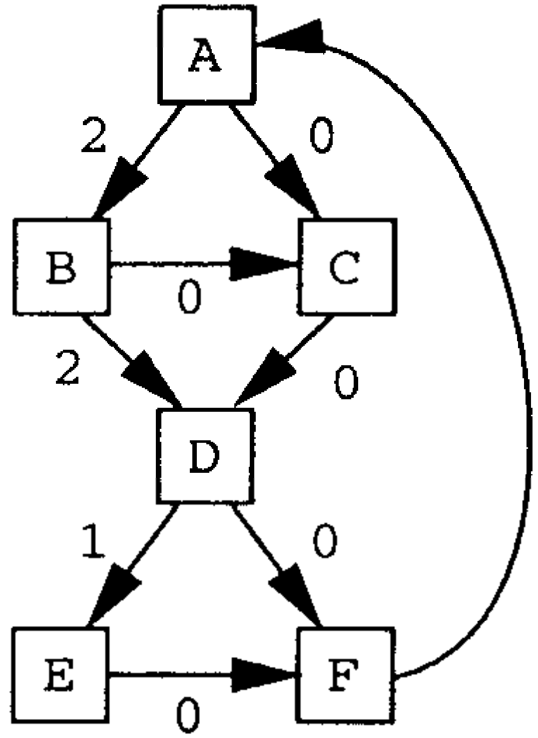
- 1. Number paths uniquely
- 2. Use spanning tree to select edges to instrument (and compute appropriate increment for each instrumented edge)
- 3. Select appropriate instrumentation
- 4. After profiling, given path number, figure out which path it corresponds to

Compact path numbering

- Aim: assign non-negative constant value to each edge such that sum of values along any path from ENTRY to EXIT is unique. Moreover, path sums should be in range 0..(NumPaths - 1) (i.e., minimal encoding)

Compact path numbering

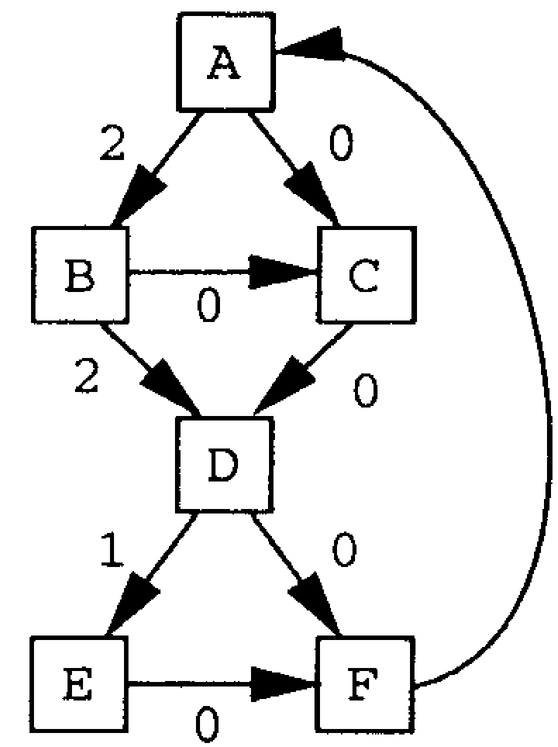
```
foreach vertex v in reverse topological order {
  if v is a leaf vertex {
    NumPaths(v) = 1;
  } else {
    NumPaths(v) = 0;
    for each edge e = v->w {
      Val(e) = NumPaths(v);
      NumPaths(v) = NumPaths(v) + NumPaths(w);
    }
  }
}
```

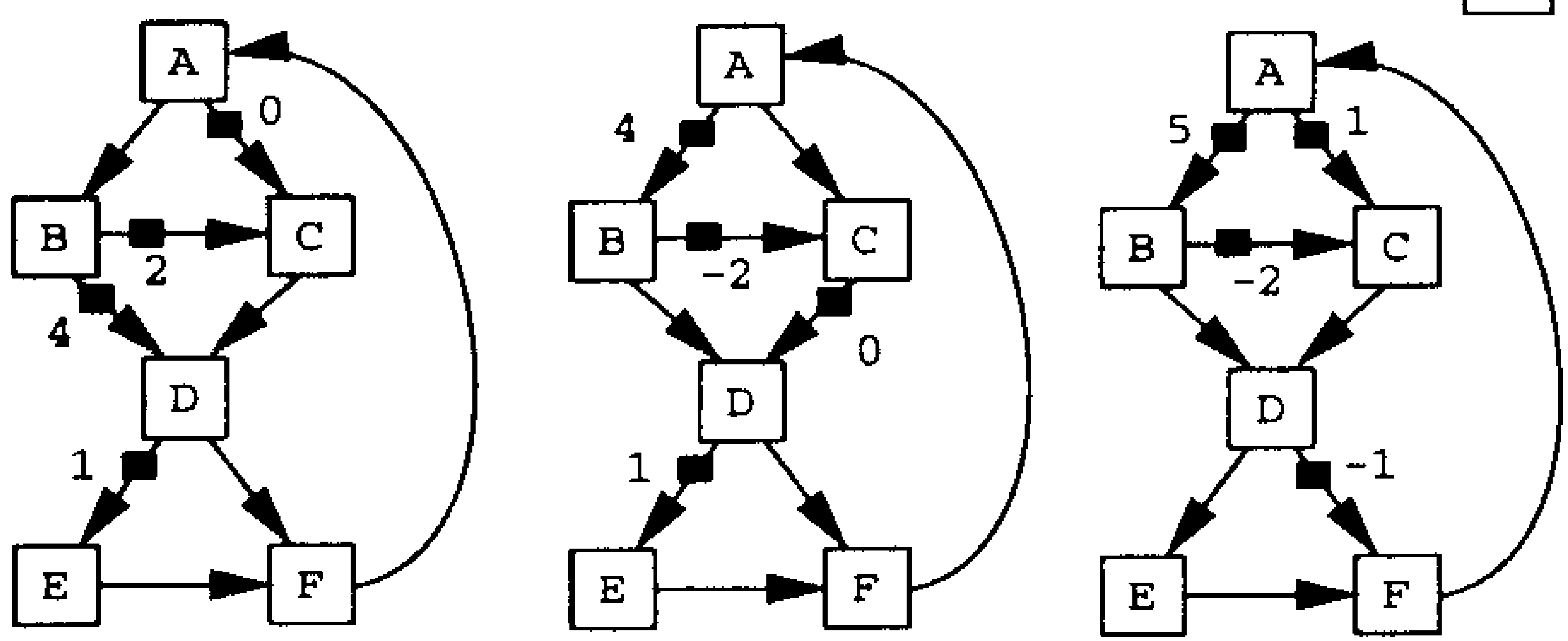


Vertex v	NumPaths(v)
A	6
B	4
C	2
D	2
E	1
F	1

Efficiently compute sums

- Find min-cost spanning tree (= max cost chord edges)
  - Chord edges will be instrumented
- Move weights from non-chord edges to chord edges



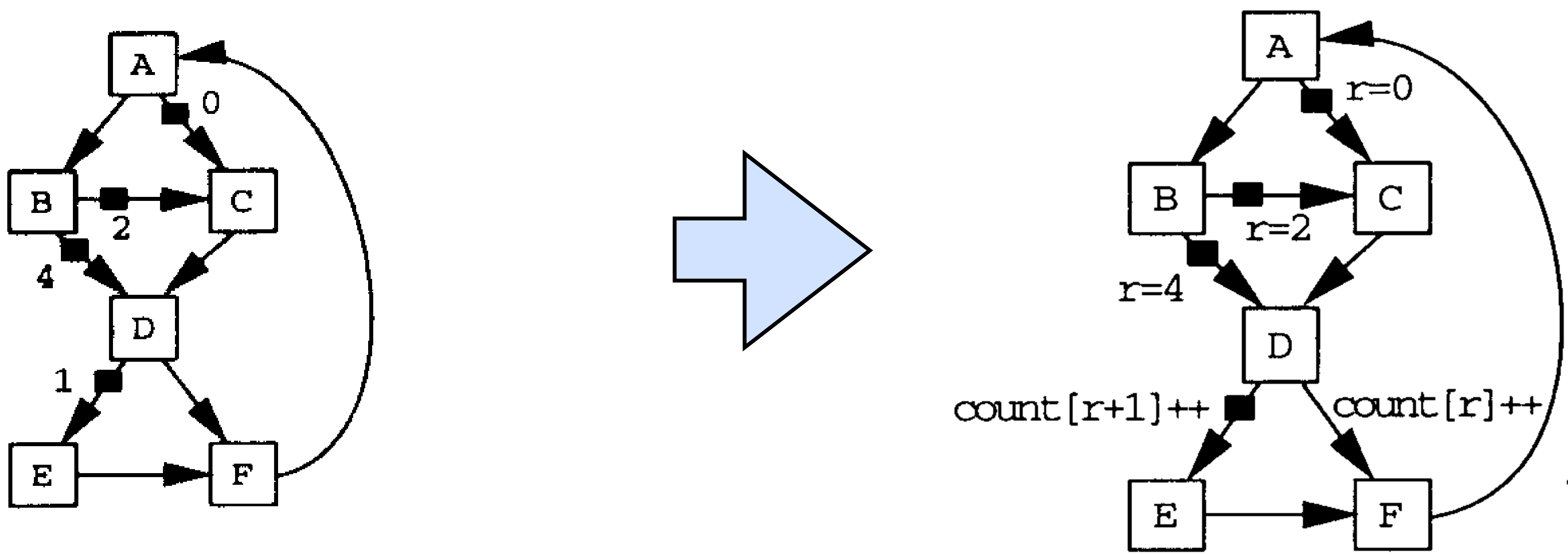


(a) (b) (c)

Instrumentation

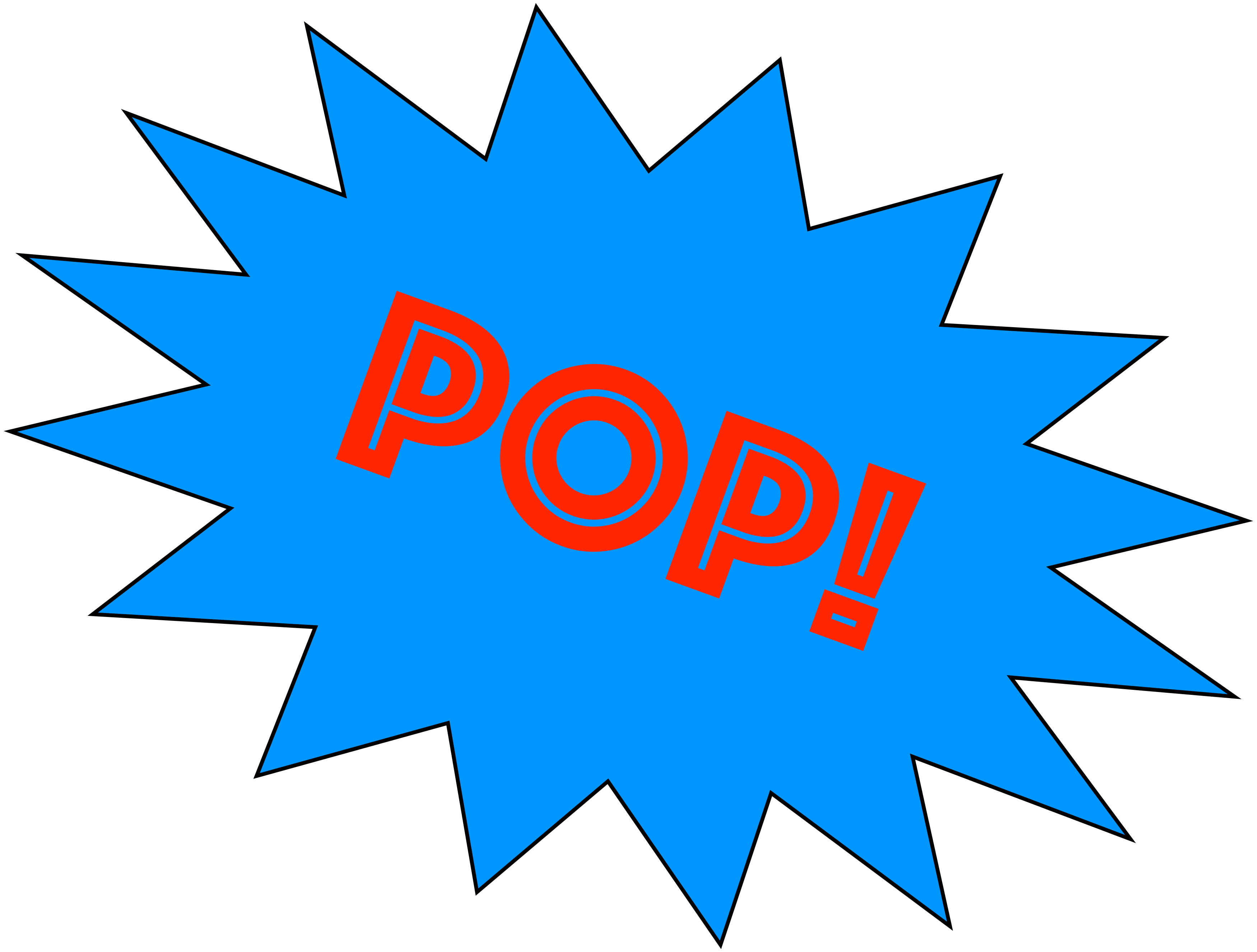
- Needed instrumentation:
  - Initialize path register (`r = 0`) at ENTRY
  - Increment register on instrumented edges (`r +=` Inc(e))
  - Record path's counter at EXIT (`count[r]++`)
- Can optimize:
  - Chord edge e can initialize counter (`r =` Inc(e)) iff first chord edge on every path from ENTRY to EXIT containing e
  - Chord edge e may increment path register and memory counter (`count[r+`Inc(e)`]++`) iff last chord edge on every path from ENTRY to EXIT containing e

Instrumentation



Details, details, details

- Algorithm works for DAGs
- Need to transform programs to be DAG like (and profile on DAG sub-graphs of cyclic graph)



Path Profiling

Path Profiling

Path ID	Frequency	Condition	Call Lines
main:0	1	`t == 1`	[9]
main:19	1	`t==2 && t >= _`	[1,3,4]
main:22	1	`t==2 && t < _ && _ >= 13`	[1,2,3]
main:23	10	`t==2 && t < _ && _ < 13`	[1,2,3,4]
main:9	11	`t >= 3 && t >= _`	[3]
main:13	55	`t >= 3 && t < _`	[2,3]
main:2	114	`t == 0 && *a == '/'`	*no call lines*
main:3	114	`t < -72`	[5]
main:1	2358	`t == 0 && *a != '/'`	[10]
main:7	2358	`t > -72 && t < -50 && _ == *a`	[6]
main:4	24931	`t < 0 && t >= -50`	[8]
main:5	39652	`t > -72 && t < -50 && _ != *a`	[7]

12 days of Christmas (also 11, to catch "off-by-one" cases)
26 unique strings
66 occurrences of non-partridge-in-a-pear-tree presents
114 strings printed, and
2358 characters printed.

- With manual examination:
  - Path 0 (executed once) initializes the recursion with the call main(2,2,...).
  - Paths 19, 22, and 23 control printing of the 12 verses.
    - P19 first verse, P23 middle 10 verses, and P22 last verse.
  - Paths 9 and 13 control printing of non-partridge-gifts within verse. (Frequencies of P9 + P13 = 66)
  - Paths 2 and 3 responsible for printing out a string.
  - Paths 1 and 7 print out the characters in a string. (why two?)
  - Path 4 skips over n sub-strings in the large string, each sub-string terminated with '/'
  - Path 5 linearly scans the string that encodes the character translation

Reversed-engineered Program

```
#include <stdio.h>
static char *strings = LARGE_STRING;       /* the original set of strings */
static char *translate = SMALL_STRING;     /* the translation mapping */
#define FIRST_DAY  1
#define LAST_DAY  12

/* the original "indices" of the various strings */
enum { ON_THE = 0, FIRST = -1, TWELFTH = -12, DAY_OF_CHRISTMAS = -13,
       TWELVE_DRUMMERS_DRUMMING = -14, PARTRIDGE_IN_A_PEAR_TREE = -25
};

char* skip_n_strings(int n,char *s) { /* skip -n strings (separator is /), */
  if (n == 0)  return s;              /* where n is a negative value */
  if (*s=='/') return skip_n_strings(n+1,s+1);
  else         return skip_n_strings(n,s+1);
}

/* find the character in the translation buffer
   matching c and output the translation */
void translate_and_put_char(char c, char *trans) {
  if (c == *trans) putchar(trans[31]);
  else             translate_and_put_char(c,trans+1);
}
```

Reversed-engineered Program

```
void output_chars(char *s) {
  if (*s == '/') return;
  translate_and_put_char(*s,translate);
  output_chars(s+1);
}

/* skip to the "n^th" string and print it */
void print_string(int n) {  output_chars(skip_n_strings(n,strings)); }

/* print the list of gifts */
void inner_loop(int count_day, int current_day) {
  if (count_day < current_day) inner_loop(count_day+1,current_day);
  print_string(PARTRIDGE_IN_A_PEAR_TREE+(count_day-1));
}

void outer_loop(int current_day) {
  print_string(ON_THE);               /* "On the " */
  print_string(-current_day);         /* ordinal, ranges from -1 to -12 */
  print_string(DAY_OF_CHRISTMAS);     /* "day of Christmas ..." */
  inner_loop(FIRST_DAY,current_day); /* print the list of gifts */
  if (current_day < LAST_DAY)
    outer_loop(current_day+1);
}

void main() { outer_loop(FIRST_DAY); }
```


and now it's time for something
completely different

Adversarial Memory for Detecting Destructive Data Races

- By Flanagan and Freund, PLDI 2010
- A dynamic analysis to find **data races** in concurrent programs
  - What's a data race?

Program Model

$$
\begin{array}{rcll}
\alpha & \in & \mathit{Trace} & ::= \mathit{Operation}^* \\[1em]
a, b & \in & \mathit{Operation} & ::= \quad rd(t,x,v) \mid wr(t,x,v) \\
 & & & \quad \mid acq(t,m) \mid rel(t,m) \\
 & & & \quad \mid \mathit{fork}(t,u) \mid \mathit{join}(t,u)
\end{array}
$$

$$
s,t,u \in \mathit{Tid} \qquad x,y \in \mathit{Var} \qquad m \in \mathit{Lock} \qquad v \in \mathit{Value}
$$

- A multithreaded program has concurrently executing threads (each with thread identifier t ∈ Tid)
- Each thread manipulates variables and locks
- A trace lists sequence of operations performed by threads
  - Ignores everything except read/writes to variables, lock operations, and fork/join.

Happens-before relation and races

The *happens-before relation* $<_\alpha$ for a trace α is the smallest transitively-closed relation over the operations[3] in α such that the relation $a <_\alpha b$ holds whenever a occurs before b in α and one of the following holds:

- [PROGRAM ORDER] Both operations are by the same thread.
- [LOCKING ORDER]: a releases a lock that is later acquired by b.
- [FORK ORDER]: a is $\mathit{fork}(t, u)$ and b is by thread u.
- [JOIN ORDER]: a is by thread u and b is $\mathit{join}(t, u)$.

- Two operations *a* and *b* are *concurrent* if neither $a <_\alpha b$ nor $b <_\alpha a$
- A trace has a *race* if there are two memory accesses to the same variable, at least one of them is a write operation, and the accesses are concurrent

Races are bad

- Often cause errors only on certain rare executions
  - Hard to reproduce and reason about
- Exacerbated by multi-core processors and relaxed memory models
- BUT many races are benign
  - E.g., approximate counters, optimistic protocols
- Lots of work on race detection
  - Static: can be difficult to reason about all possible interleaving
  - Dynamic: interleavings with races may be rare
- This work:
  - standard dynamic analysis to detect "racy" variables
  - Then try to produce an *erroneous execution* that exhibits the race and produces observable incorrect behavior (e.g., crash, uncaught exception, etc.)

Double-checked locking example

```
class Point {
  double x, y;
  static Point p;

  Point() { x = 1.0; y = 1.0; }

  static Point get() {
    Point t = p;
    if (t != null) return t;
    synchronized (Point.class) {
      if (p==null) p = new Point();
      return p;
    }
  }

  static double slope() {
    return get().y / get().x;
  }

  public static void main(String[] args) {
    fork { System.out.println( slope() ); }
    fork { System.out.println( slope() ); }
  }
}
```

- Relaxed memory model means that get().x could evaluate to zero.
  - (Thus, the race on p is *destructive*, i.e., non-benign)
- But most of the time, destructive behavior not exhibited

Adversarial Memory

- Exploits full flexibility of relaxed memory model to try and cause crashes
- Tool tracks memory and synchronization operations of execution, and keeps a **write buffer** recording history of writes to racy variables
- When a thread asks for a value, return older (but still legal) values whenever possible

Memory Models

- **Sequential Memory Model:** read operation $a = rd(t,x,v)$ in trace α may only return the value of the most recent write to that variable in α
  - Intuitive but limits optimization by compiler, virtual machine, and hardware.
- **Happens-Before Memory Model:** read operation $a = rd(t,x,v)$ in trace α may return the value of any write operation $b = wr(u, x, v)$ provided:
  1. b does not happen after a; and
  2. no intervening write c to x where $b <_\alpha c <_\alpha a$

Out-of-thin-air

- Consider

$$x := y \mid\mid y := x$$

- Assume x and y are initially zero.
- Under happens-before memory model, the following trace is possible:

  rd(t1, x, 42) wr(t1, y, 42) rd(t2, y, 42) wr(t2, x, 42)
- Where did 42 come from??!?
- Java Memory Model extends the happens-before memory model with a *causality requirement* to preclude non-sensical traces as above
- This paper uses **Progressive Java Memory Model:** read operation a = *rd(t,x,v)* in trace α may return the value of any write operation b = *wr(u, x, v)* provided:

  1. b is before a in trace α; and
  2. no intervening write c to x where $b <_\alpha c <_\alpha a$

Adversarial Memory Implementation

- Uses **vector clocks** to record time stamps of write operations
  - Vector clocks can be used to determine the happens-before relation
- Read operation for x at time C_t can return any value so long as it satisfies the Progressive Java Memory Model
  - i.e., a write in the write buffer for x that happened at time K_i such that there is no write at time K_j where $K_i \sqsubseteq K_j \sqsubseteq C_t$

Adversarial Memory Heuristics

- **Sequentially consistent:** always return most recently written value
- **Oldest:** chose "most stale" value. (occasionally return most-recent value to satisfy fairness assumptions)
- **Oldest-but-different:** return oldest element that is different from the last value read
- **Random:** return a random value from the permitted values
- **Random-but-different:** return a random value from the permitted value that is dfferent from the last value read

Effectiveness

Program	Field	No Jumble	Jumble Configurations: *Sequentially Consistent*	*Oldest*	*Oldest-But-Different*	*Random*	*Random-But-Different*	Destructive Race?
		Erroneous Behavior Observation Rate (%)						
Figure 8	`x`	**0**	**0**	**0**	**83**	**84**	**92**	**Yes**
Figure 2	`p`	0	0	0	0	0	0	No
Figure 2	`p.x`	**0**	**0**	**60**	**52**	**32**	**30**	**Yes**
Figure 2	`p.y`	**0**	**0**	**48**	**53**	**27**	**30**	**Yes**
jbb	`Company.elapsed_time`	**0**	**0**	**100**	**0**	**15**	**5**	**Yes**
jbb	`Company.mode`	**0**	**0**	**100**	**100**	**95**	**98**	**Yes**
montecarlo	`Universal.UNIVERSAL_DEBUG`	0	0	0	0	0	0	No
mtrt	`RayTracer.threadCount`	0	0	0	0	0	0	No
raytracer	`JGFRayTracerBench.checksum1`	**0**	**0**	**100**	**100**	**100**	**100**	**Yes**
tsp	`TspSolver.MinTourLen`	0	0	100	100	100	100	QoS
sor	array index `[0]` and `[1]`	**0**	**0**	**100**	**100**	**100**	**100**	**Yes**
lufact	array index `[0]` and `[1]`	**0**	**0**	**100**	**100**	**100**	**100**	**Yes**
moldyn	array index `[0]` and `[1]`	**0**	**0**	**100**	**100**	**100**	**100**	**Yes**

Performance

Program	Size (lines)	Num. Threads	Field	Base Time (s)	Slowdown		Num. Instances	Num. Writes	Max. Buffer Size	
					Empty	Jumble			No Comp.	With Comp.
jbb	30,491	5	`Company.elapsed_time`	74.4	1.3	1.3	1	2	2	2
jbb	30,491	5	`Company.mode`	74.4	1.3	1.4	2	10	8	4
montecarlo	3,669	4	`Universal.UNIVERSAL_DEBUG`	1.6	1.2	1.2	1	40,005	40,005	5
mtrt	11,317	5	`RayTracer.threadCount`	0.5	4.5	4.9	1	10	10	5
raytracer	1,970	4	`JGFRayTracerBench.checksum`	5.6	1.1	1.1	1	6	6	5
tsp	742	5	`TspSolver.MinTourLen`	0.7	2.3	4.0	1	26	26	23
sor	883	4	array index `[0]` and `[1]`	0.6	3.9	5.8	2,106	104,620	255	32
lufact	1,627	4	array index `[0]` and `[1]`	0.4	4.1	4.2	1,108	14,526	2,047	7
moldyn	1,407	4	array index `[0]` and `[1]`	0.9	4.1	8.9	62	53,433	16,383	32

Figure 10. Performance of JUMBLE under the *Sequentially-Consistent* configuration.