Tracing - Just - In - Time compilation
(tracing JIT's)

- Please interrupt with questions!

1960s hinting at the idea of JITs

1990's the beginning of JIT compilers

prehistory

What is a just-in-time compiler? (JIT)

early 00's the beginning of tracing JIT's

What is a tracing JIT?

late 00's modern tracing JITs

history

Prehistory: Hinting at the idea of JIT's

McCarthy 1960's:

- created formalism for defining functions recursively
- in LISP
- programmer could select certain functions to be compiled into machine code and stored in core memory
- later, same functions executed from core memory
- values then computed 60x faster

1980's

- Smalltalk is popular
- Self develops out of Smalltalk
  - ↳ prototype-based, dynamic, object-oriented
  - ↳ had influences on JavaScript

Chambers and Ungar 1989

- Self JIT compiler
- first JIT compiler (wasn't called that)
- generate machine code on demand and cache for later use

McCarthy 1960's

Chambers and Ungar 1989

- both "generate code on demand and store for later use"
- 'Chambers and Ungar is the first JIT (but not called that)

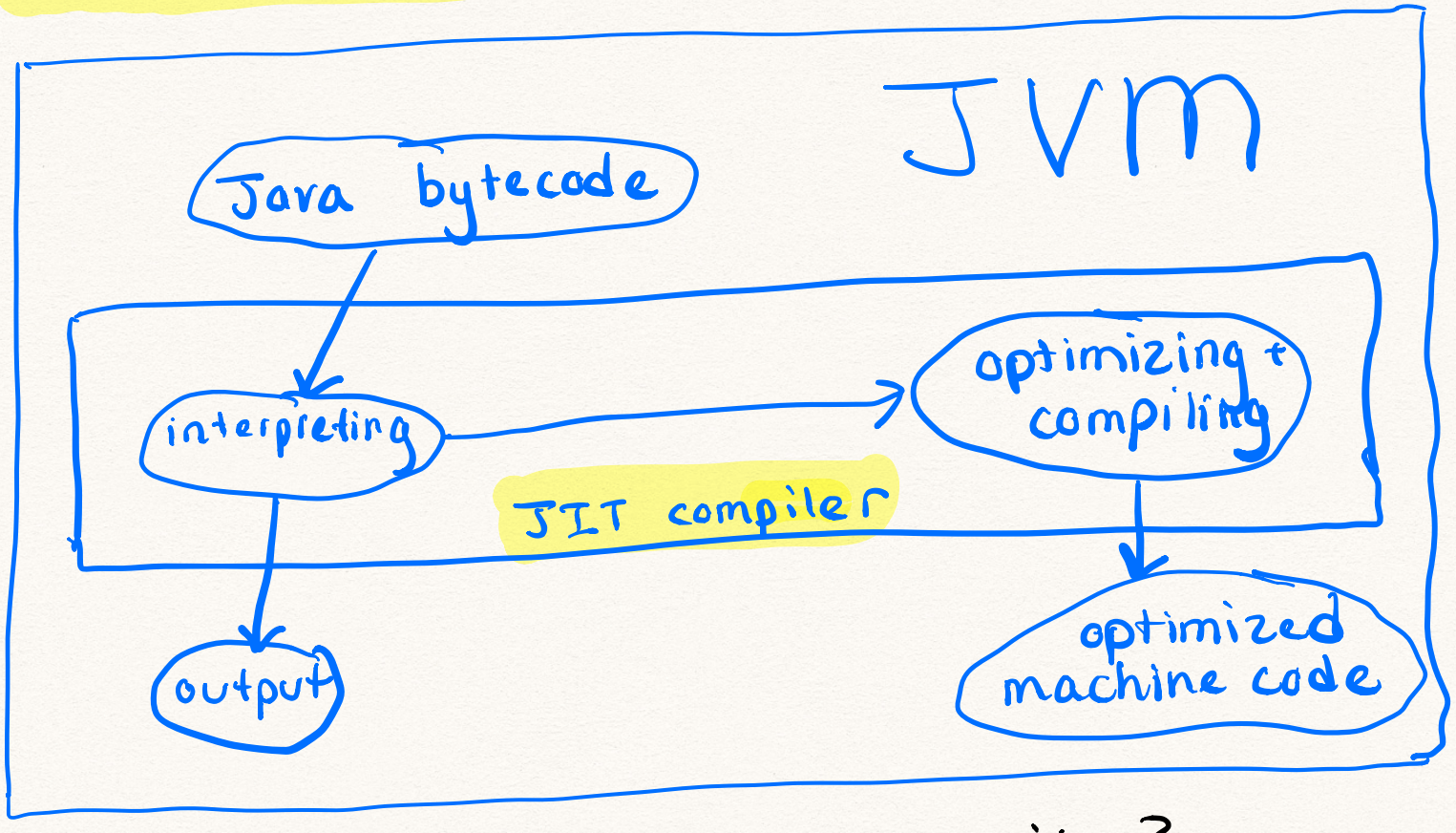
Prehistory: The beginning of JIT compilers

1996

- Sun switches from Self to Java
- Java is popular + interesting challenges
- wants to be portable, secure
  - ✗ interpreting is too slow
  - ✗ compiling breaks "portable, secure"
  - ✓ somewhere in-between

Cramer et. al 1997

- the Java Virtual Machine (JVM) is one of the first to call itself a JIT compiler



What is a JIT compiler?

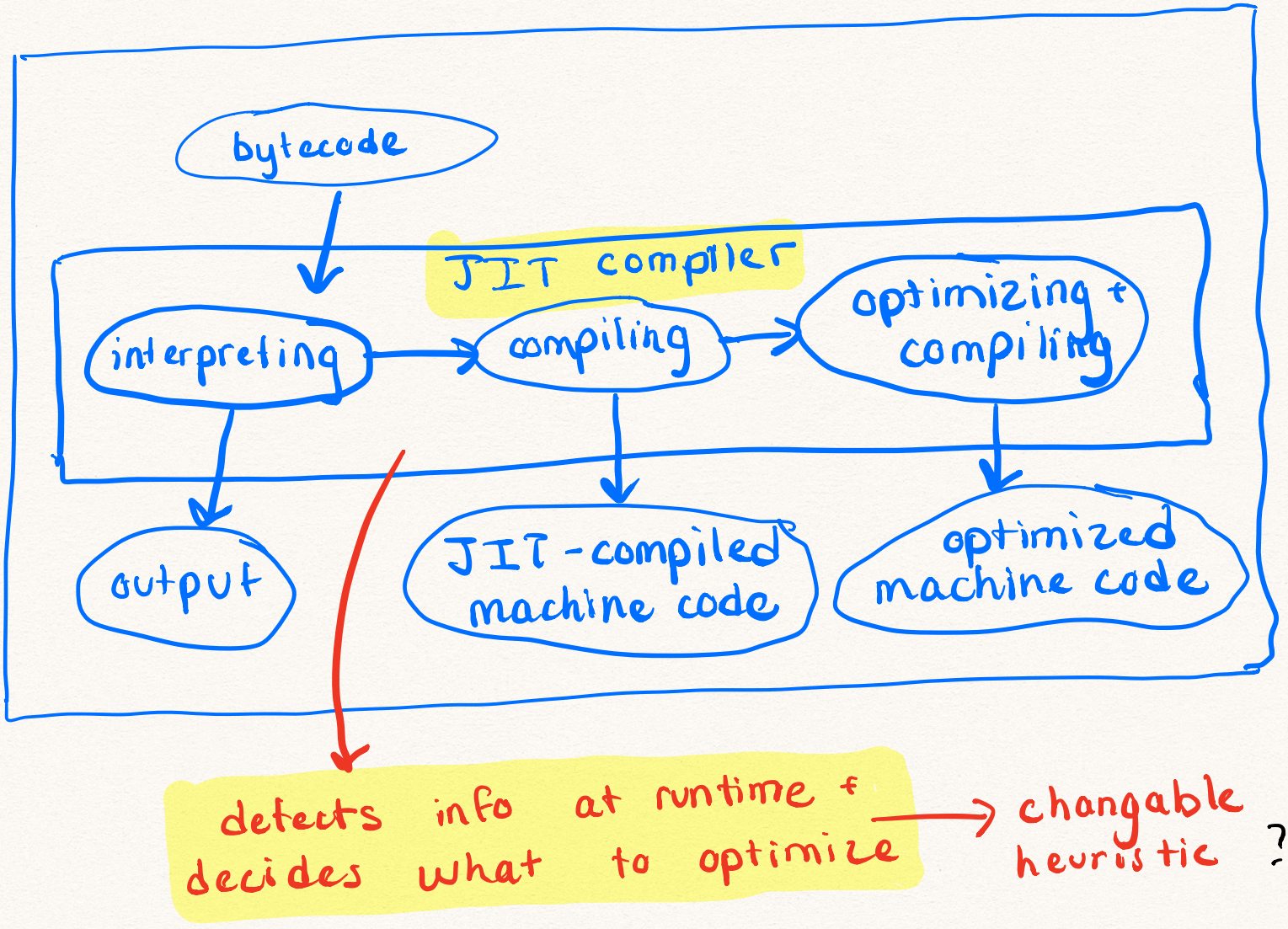
What is a just-in-time compiler (JIT)?

- combines AOT and interpreters

  speed from compiled code + Flexibility from interpreting

- code / bytecode → machine code / low-level representation

- "just-in-time" = runs after a program starts + uses runtime information



	interpreter	ahead-of-time AOT	just-in-time JIT
start-up time	fast	slow	fast-ish
execution time	fast or slow	fast	faster w/ more optimizations
use runtime info	yes-ish	no	yes

What kinds of optimizations?

- remove redundancy

```
class A;
public void foo(A bar){
    x = bar.value;
    // something that doesnt
    // involve x
    y = bar.value;
    return x+y;
}
```

```
class A;
public void foo(A bar){
    x = bar.value;
    // something
    // something

    return x+x;
}
```

- type specialization

```
Function foo(x: any) {
    return x+1;
}
foo(12);
foo(13);
```

```
Function foo(x: int) {
    return x+1;
}
foo(12);
foo(13);
```

- optimal bytecode reordering

```
iload y
iconst 2
iload z
imul
iadd
istore x
```

```
iconst 2
iload z
imul
iload y
iadd
istore x
```

- method inlining
- dead code elimination
- removing redundant array bounds checks

McCarthy 1960's

Chambers and Ungar 1989

Cramer et. al 1997

- Sun adapts "generate and store idea"
- provides security and portability to Java
- JIT idea expanded on
- now actually called "JIT"

The beginning of tracing JIT's

2000's • desire to optimize machine code

Bala et. al 2000

- Dynamo system compiles:

  machine code ⟶ optimized machine code

- ... using runtime info
- detect "hot" instruction sequences and optimise those

```
x = 100;
while (x > 0) {
    x = x - 1;
}
```

"hot" instruction sequence = frequently executed

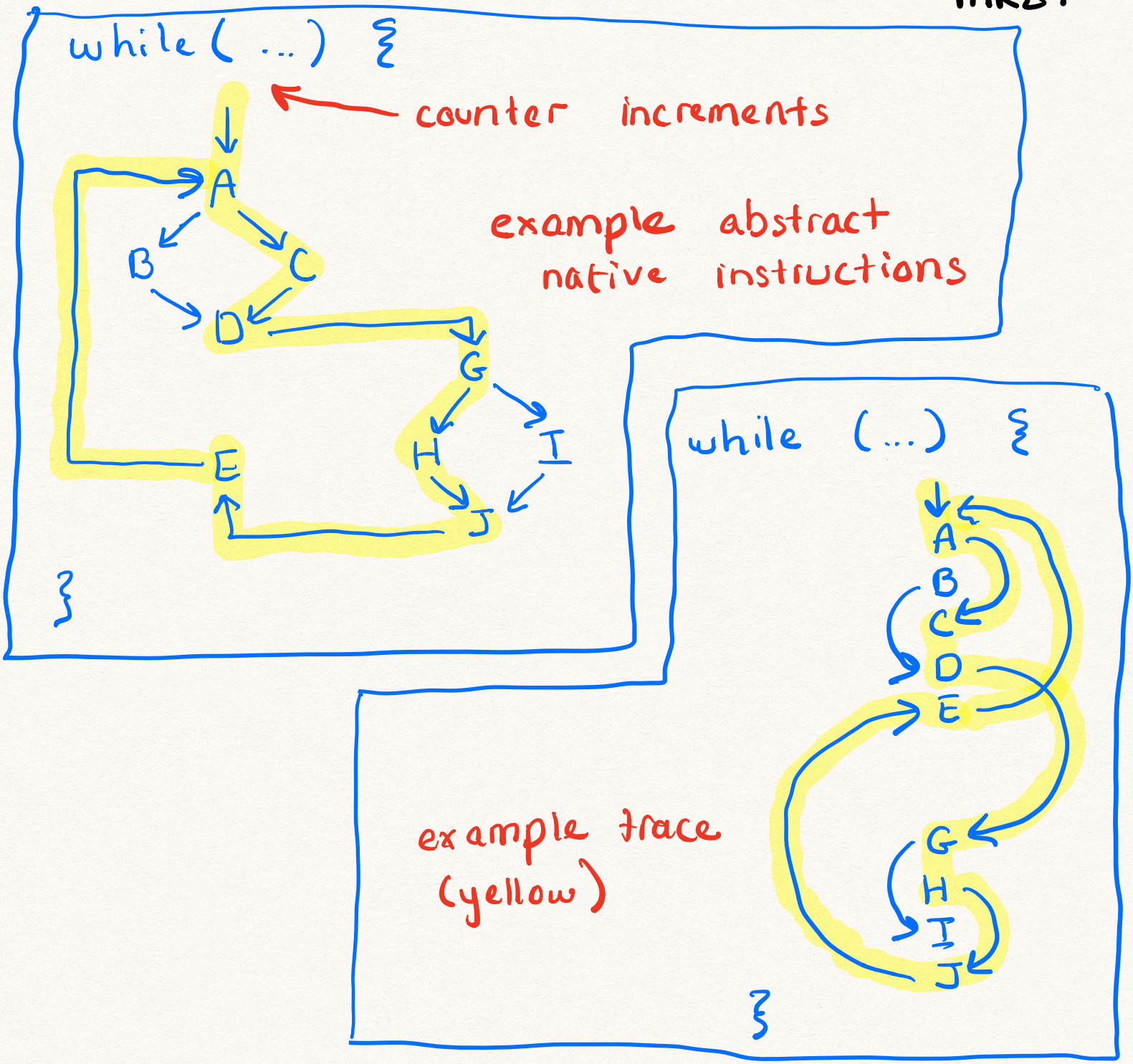
↳ called "trace"

How do we detect traces?

```
x = 100;
while (x > 0) {
    x = x - 1;
}
```

example user-facing program

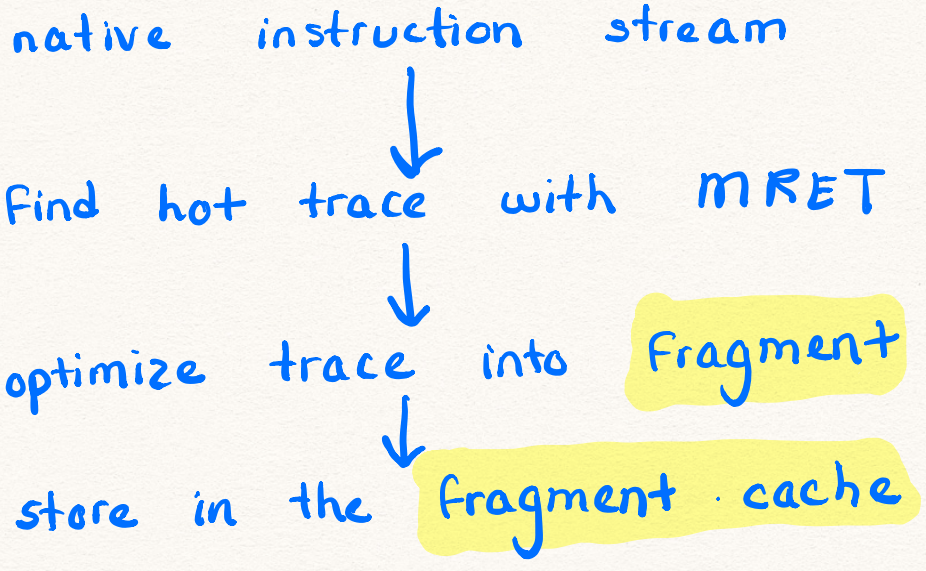
traces detected by "most recently executed tail" MRET



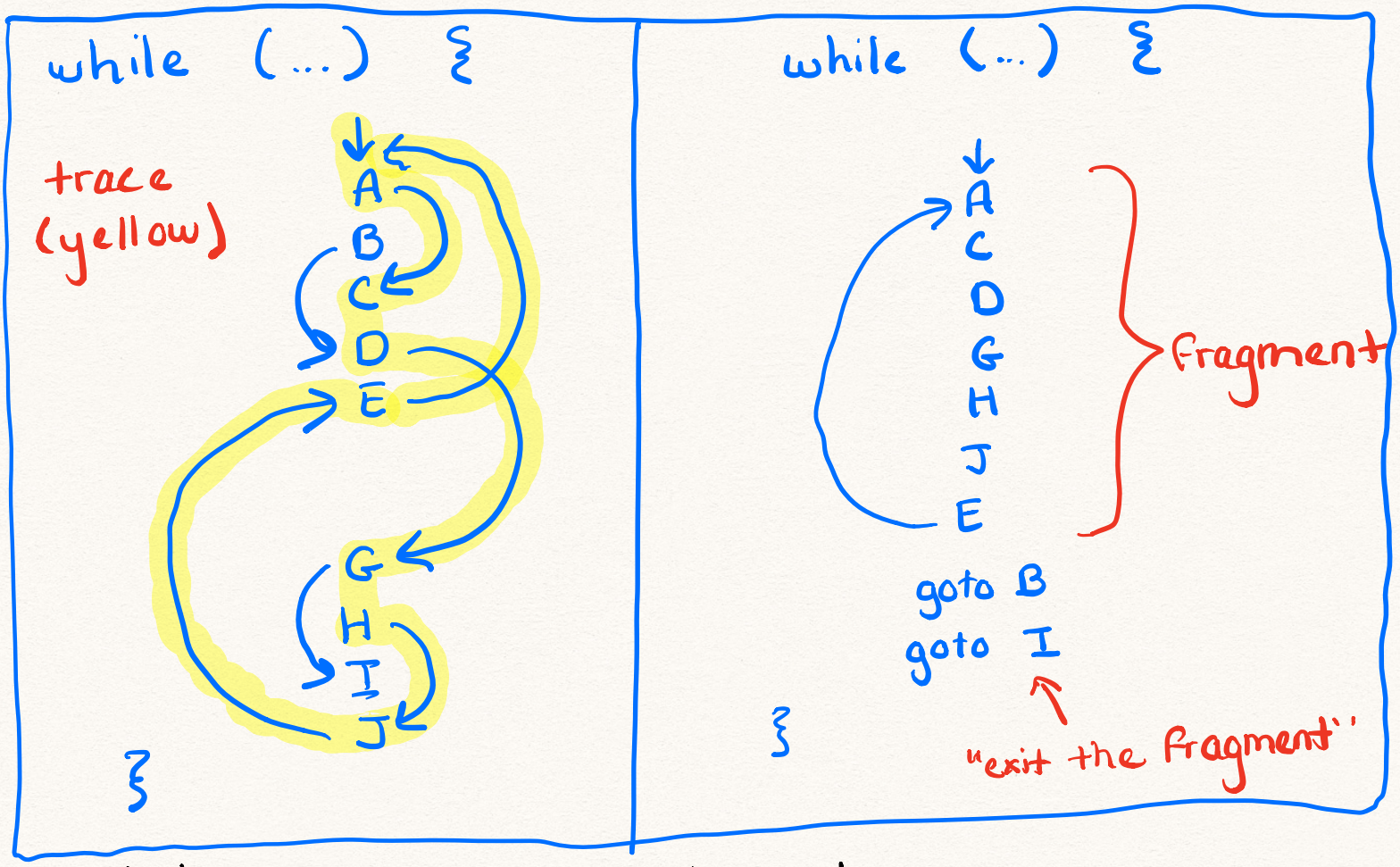
Traces are

- chunks of the native instruction stream
- discovered, not created
- evolved from systems
  - ↳ execution traces
  - ↳ "offline binary translator" profilers

How are traces used?



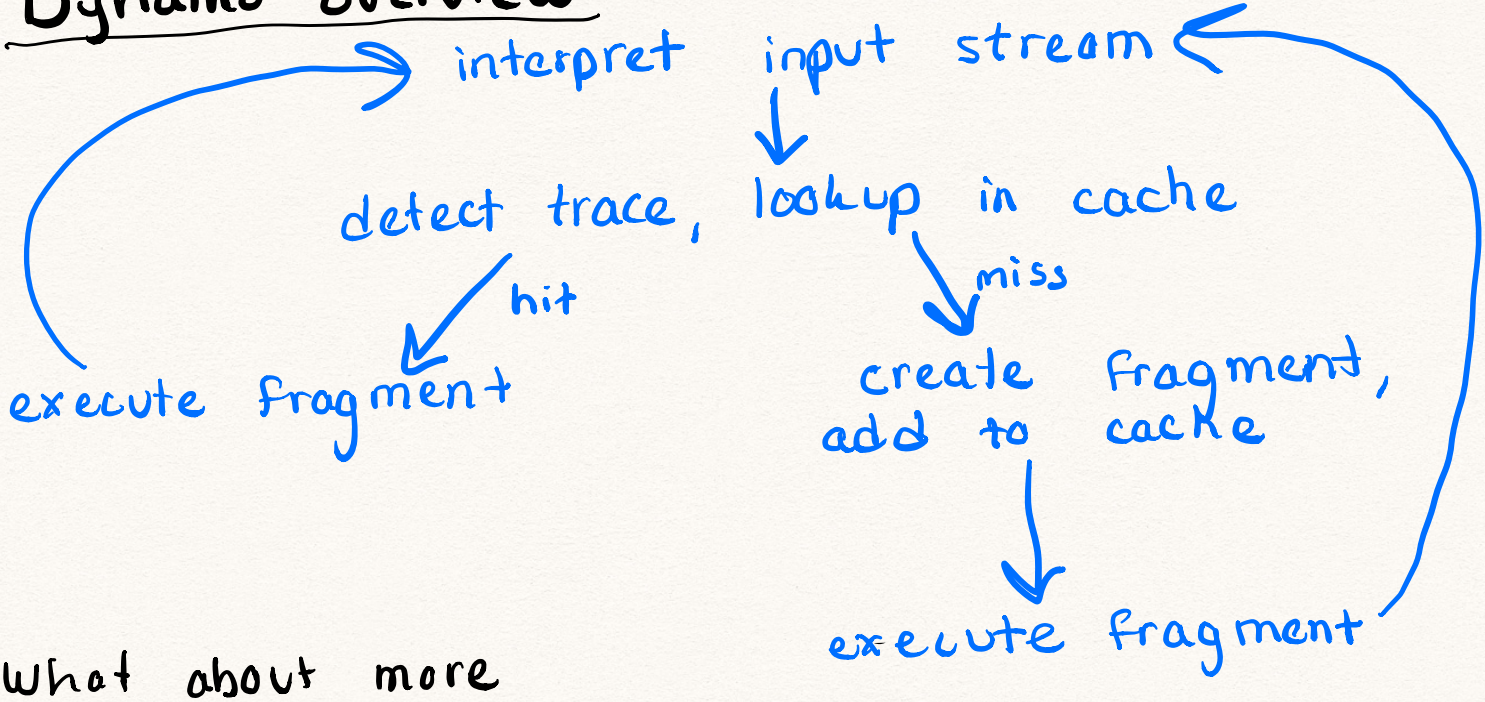
What is a fragment?
What is the fragment cache?
How does Dynamo use fragments?



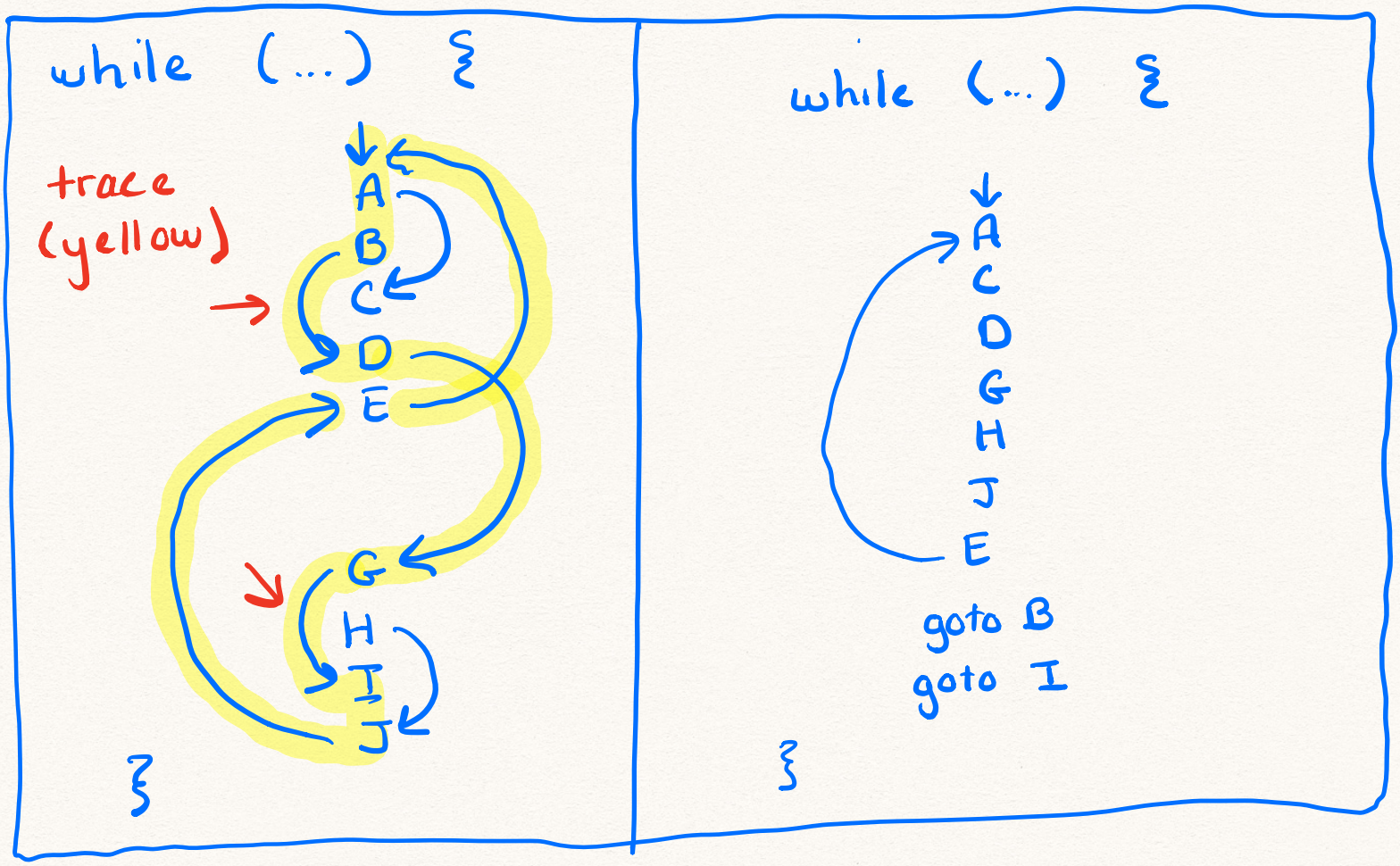
emitted to the fragment cache

- stores fragments for later execution on the processor
- indexed by address

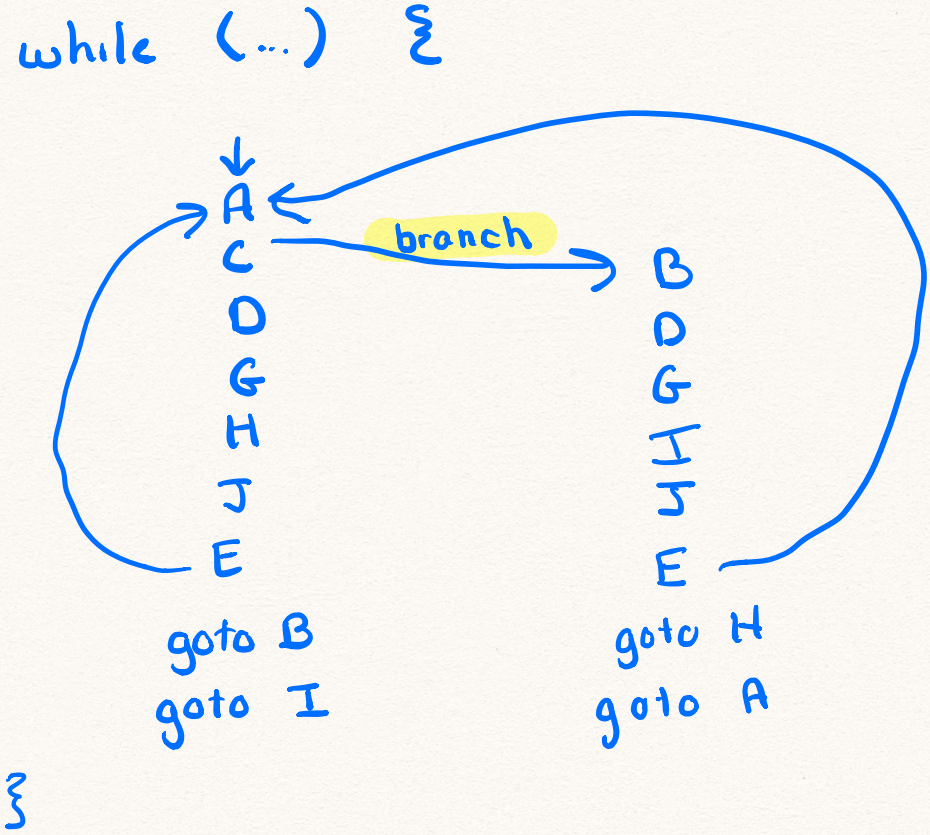
Dynamo overview:



What about more complex traces?



Fragment in the fragment cache updated via Fragment linking



The beginning of tracing JIT's

mid 2000's

- want to use Java on mobile phones
  - ✗ full-scale JIT compilers lack necessary performance
  - ✗ embedded JIT compilers provide performance, but are too specific

"embedded JIT" = just-in-time compiling VM implementation

Gal et. al 2006

- Hotpath VM

  Java bytecode ⟶ machine code

- lightweight JIT for embedded Java VM (Jamvm)
- uses tracing and Single Static Assignment (SSA)
- targets "hot" paths similar to Dynamo

Single Static Assignment = each variable assigned exactly once + defined before it is used

How is Hotpath VM similar to Dynamo? and different?

Similarities:

+ both use the same technique (MRET) for identifying traces
+ both operate on lower-level code Java bytecode / native instructions
+ same method of "fragment linking"

Differences:

X different definitions of "trace"

☆ most tracing JIT's have a unique definition of "trace" ☆

→ JIT = "car"

→ tracing JIT = one "make"

→ tracing JIT variations = different "models"

So what is HotpathVM's definition of "trace"?

non-sequential code turned into linear instructions

```
main (String[] args) {
    int i, k = 0;
    for (i=0; i<1000; ++i) {
        ++k;
    }
    print (k);
}
```

hotspot

```
    iconst_0
    istore_2
    iconst_0
    istore_1
A:  iload_1
    sipush 1000
    if_icmpge B
    iinc 2, 1
    iinc 1, 1
    goto A
B:  getstatic System.out
    iload_2
    invoke print(int)
    return
```

hotpath

trace

during execution, runtime info is saved to create a trace

- value of program counter
- opcode of the instruction
- trace value on top of the stack
- converted to SSA

Difference from Dynamo:

✗ Dynamo traces =
- chunks of the native input stream found at runtime and optimized
- equiv. to "hot path"

✗ HotpathVM traces =
- linear record of execution gathered at runtime
- not equiv to "hot path"
- influenced by work on parsing tree grammars

How are HotpathVM traces used?

Java bytecode

↓

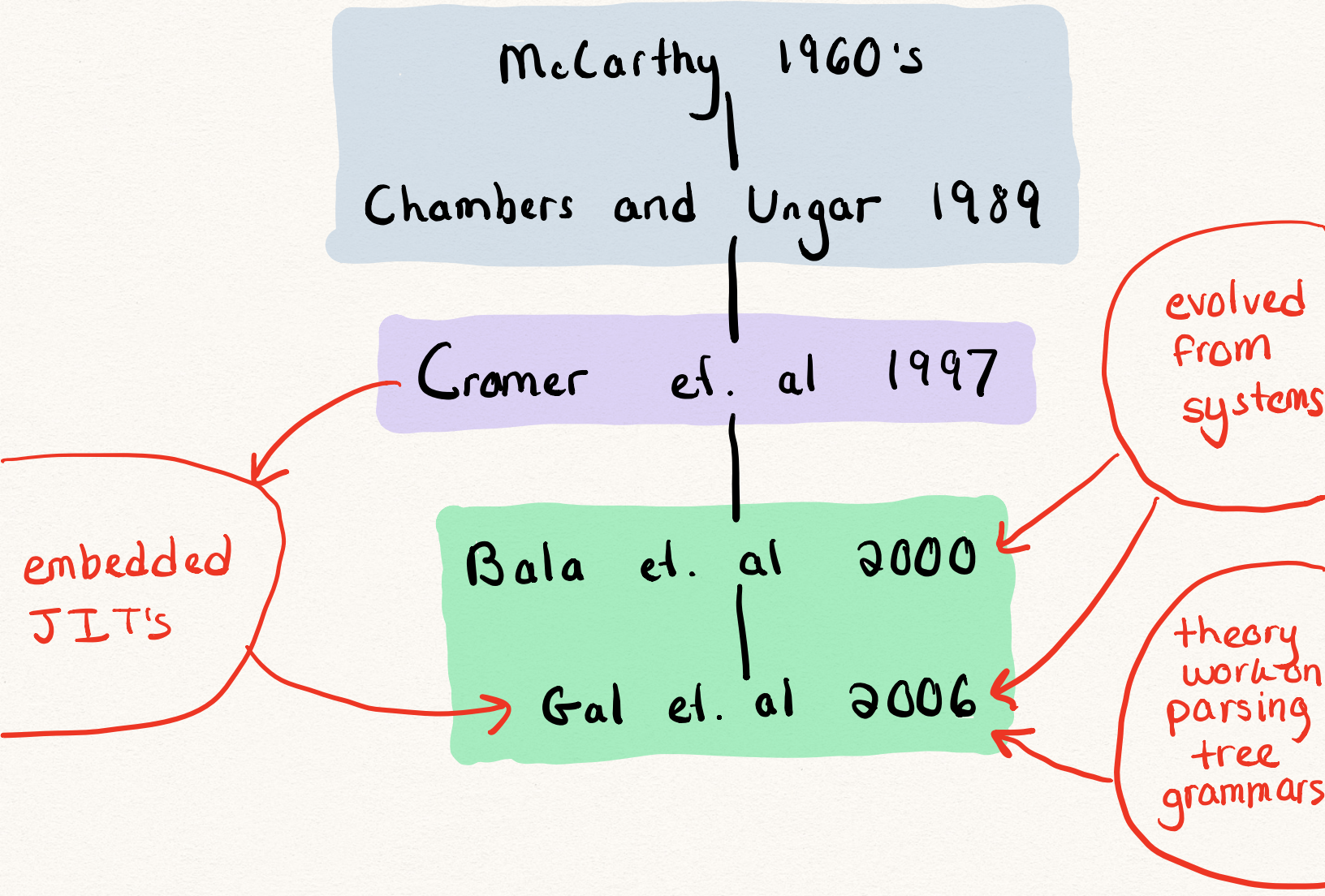
JIT-compile until "hot" path is detected

↓

record runtime info to gather trace

↓

optimize and compile to machine code



- Both borrow the idea of traces from adjacent fields
- Both apply it to existing work on JIT's

What is a tracing JIT?

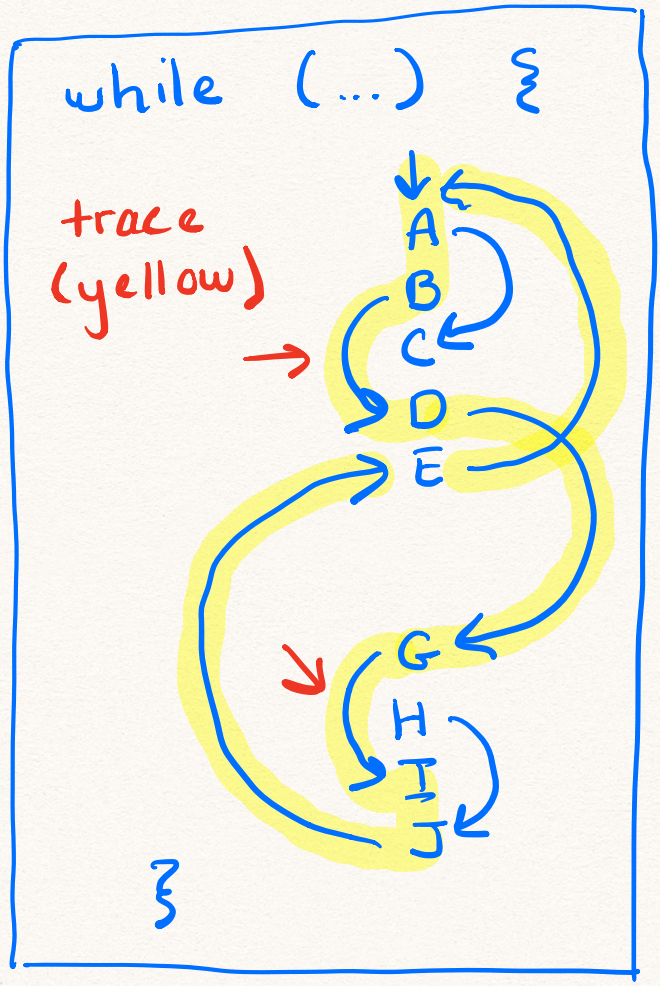
JIT that collects runtime info in the form of a linear sequence of frequently executed instructions

★ most tracing JIT's have a unique definition of "trace" ★

→ JIT = "car"

→ tracing JIT = one "make"

→ tracing JIT variations = different "models"




```
iconst_0
istore_2
iconst_0
istore_1
A: iload_1
   sipush 1000
   if_icmpge B
   iinc 2, 1
   iinc 1, 1
   goto A
B: getstatic System.out
   iload_2
   invoke print(int)
   return
```

hotpath

trace

Modern tracing JIT's

late 2000's

- Javascript is popular
  - ✗ dynamic langvages are difficult to compile as type information isnt known until runtime
  - ✓ JIT's can do type specialization at runtime

Gal et. al 2009

- tracing JIT for Javascript (TraceMonkey)
- starts running as a JS interpreter, then compiles "hot" sequences to fast native code
- assumptions:
  - → programs spend most of their time in "hot" loops
  - → most loops are type stable

Differences from Gal et. al 2006?

Similarities:

- ✓ trace-based JIT
- ✓ both target loops ("hot paths")

Differences:

- ✗ trace concept more fleshed-out
- ✗ traces specifically designed to provide type specialization

What is a TraceMonkey trace like?

- speculative
- inserts guards → do control flow
- tree-based

```
for (var i=2; i<100; ++i) {
  if (!primes[i])
    continue;
  for (var k=i+1; i<100; k+i)
    primes[k] = false;
}
```

"hot" path

loop counter

```
guard → i == int
      → k == int
      → primes = array
primes[k] = false
end
```

guards control flow in the trace and compiled machine code

What about the outer-loop?

Nested loops form "trace trees"

```
for (var i=2; i<100; ++i) {
    if (!primes[i])
        continue;
    for (var k=i+1; i<100; k+i)
        primes[k] = false;
}
```

loop counter

"hot" path

already compiled

```
guard → i == int
guard → primes == array
      → primes[i] == int
if (!primes[i])
    continue
```

```
guard → i == int
      → k == int
      → primes == array
primes[k] = false
end
```

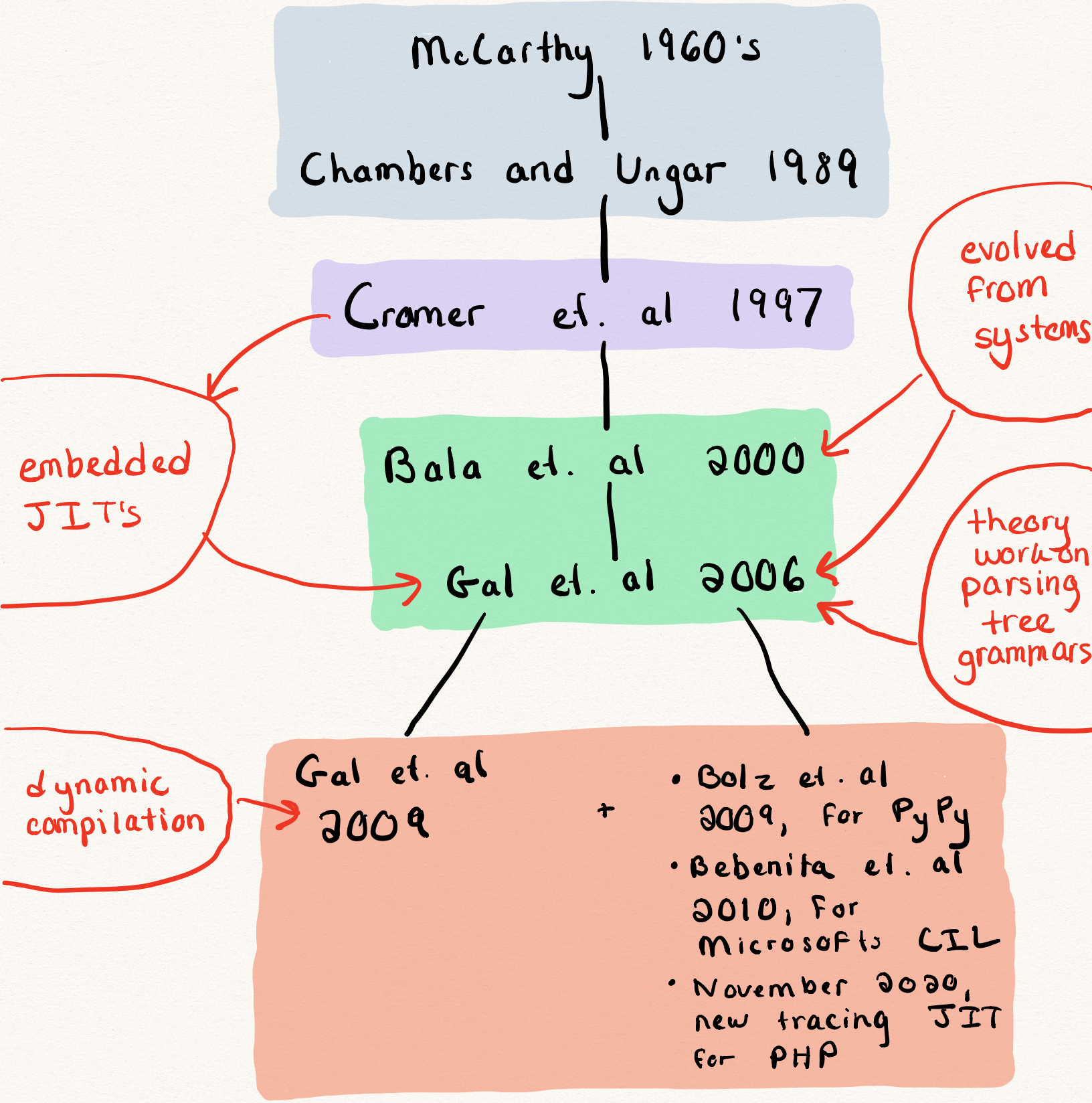
```
end
```

Discussion:

JIT that collects runtime info in the form of a linear sequence of frequently executed instructions

...but Gal et. al 2009 introduced trace trees. Does this change the definition?

+ fragment linking



- applied existing ideas (tracing JIT) to modern problem spaces
- Bala et. al paper particularly well-timed