Static Elaboration of Recursion for Concurrent Software

Stephen A. Edwards and Jia Zeng

Columbia University

The Main Points

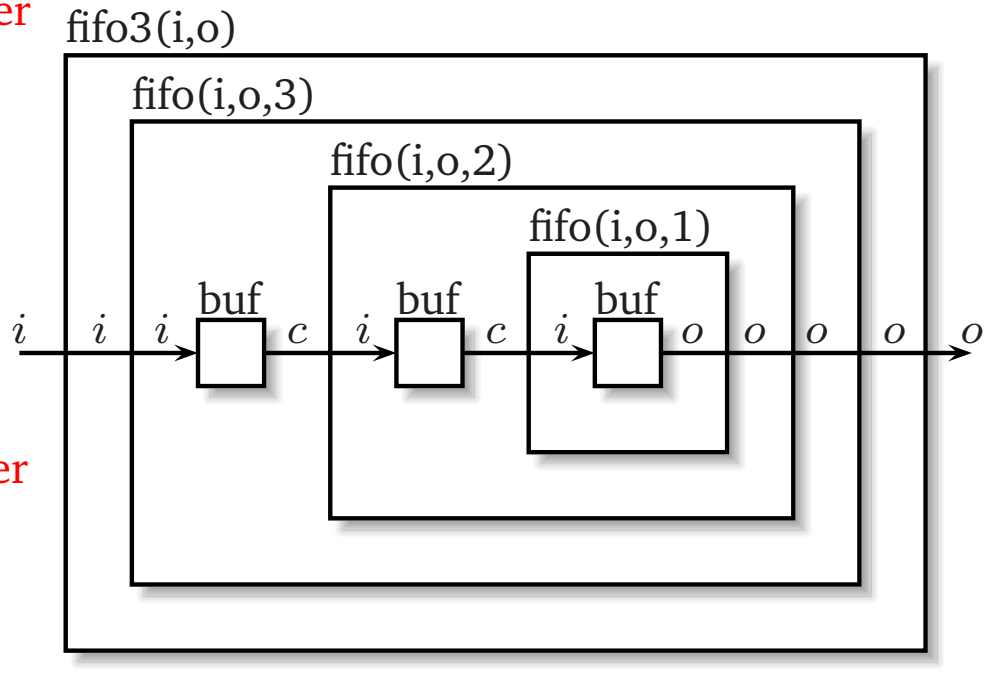
- You can build concurrent structures recursively
- We analyze and unroll recursion at compile-time
- Our Binding-time analysis:

  From recursive calls, consider control- and data-dependencies
- Our partial evaluation:

  Symbolic simulation on basic blocks
- This works

A Three-Place FIFO in SHIM

```
void fifo3(chan int i, chan int &o) {
  fifo(i, o, 3); // Three−place buffer
}

void fifo(chan int i, chan int &o,
          int n) {
  if (n > 1) {
    chan int c;
      buf(i, c); // Run a single buffer
    par // in parallel with an
      fifo(c, o, n−1); // n−1 buffer
  } else buf(i, o); // Base case
}

void buf(chan int i, chan in &o) {
  for (;;)
    next o = next i; // One−place buffer
}
```



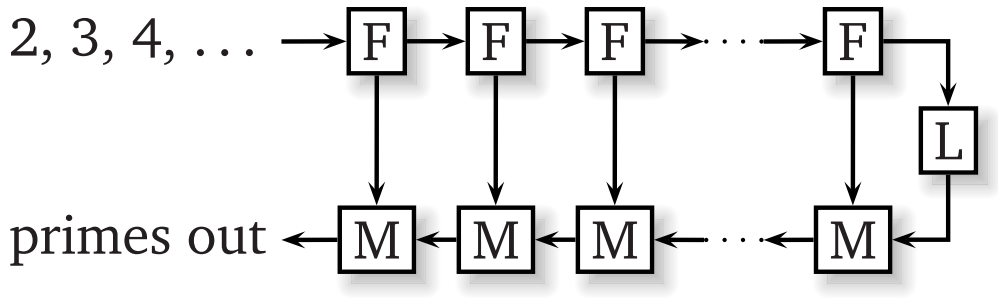
...Simplified By Our Technique

```
void fifo3(chan int i, chan int &o) {
  fifo(i, o, 3); // Three-place buffer
}

void fifo(chan int i, chan int &o,
          int n) {
  if (n > 1) {
    chan int c;
      buf(i, c); // Run a single buffer
    par // in parallel with an
      fifo(c, o, n-1); // n-1 buffer
  } else buf(i, o); // Base case
}

void buf(chan int i, chan in &o) {
  for (;;)
    next o = next i; // One-place buffer
}
```

```
void fifo3(chan int i, chan int &o) {
  chan int c1, c2, c3;
    buf(i, c1);
  par
    buf(c1, c2);
  par
    buf(c2, o);
}

void buf(chan int i, chan in &o) {
  for (;;)
    next o = next i;
}
```

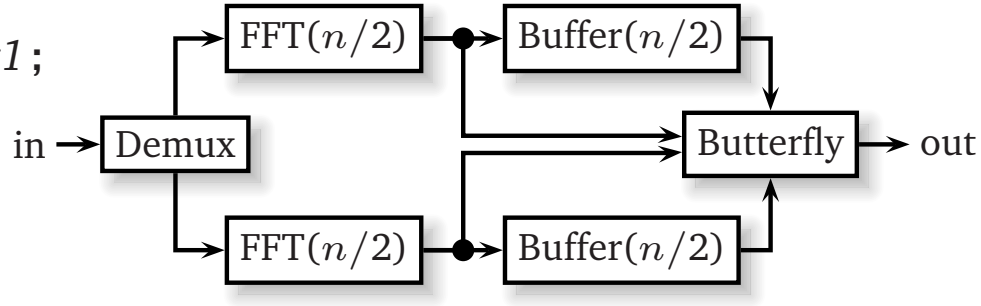
A Prime-Number Sieve

```
void sieve(chan int num_in, chan int &prime_out, int n) throws Done {
  if (n) {
    chan int prime, num_out, prime_in;
    {
      next prime = next num_in; // First to arrive is prime
      for (;;)
        if (next num_in % prime) // Then, discard multiples of our prime
          next num_out = num_in;
    } par {
      next prime_out = next prime; // Send our prime
      for (;;) next prime_out = next prime_in; // then pass the rest
    } par
        sieve(num_out, prime_in, n-1); // Recurse
  } else {
    next prime_out = next num_in;
    throw Done; // Stop pipeline
  }
}
```



An FFT: Divide-and-conquer

```
void fft(chan cplx in, chan cplx &out, uint n, int32 theta) {
  if (n == 1) {
    for (;;) next out = next in; // Trivial FFT
  } else {
    chan cplx even, odd, q, t, q1, t1;
        for (;;) { // Demux
          next even = next in;
          next odd = next in;
        }
    par fft(even, q, n/2, theta * 2); par fft(odd, t, n/2, theta * 2);
    par cplx_buffer(q, q1, n/2); par cplx_buffer(t, t1, n/2);
    par for (;;) { // Merge even and odd samples
       for (int i = 0 ; i < n/2 ; i++)
         next out = butterfly(next q, next t, theta * i / 2);
       for (int i = n/2 ; i < n ; i++)
         next out = butterfly(next q1, next t1, theta * i / 2);
    }
  }
}
```



Our Technique

1. Binding-time analysis using slicing

   Objective: track the fewest variables necessary

   Intuition: tracking more variables makes result bigger

2. Regenerate the program using symbolic simulation

   Objective: simulate and rewrite each basic block

   Intuition: predict the values of the tracked variables at compile time and leave remaining code alone

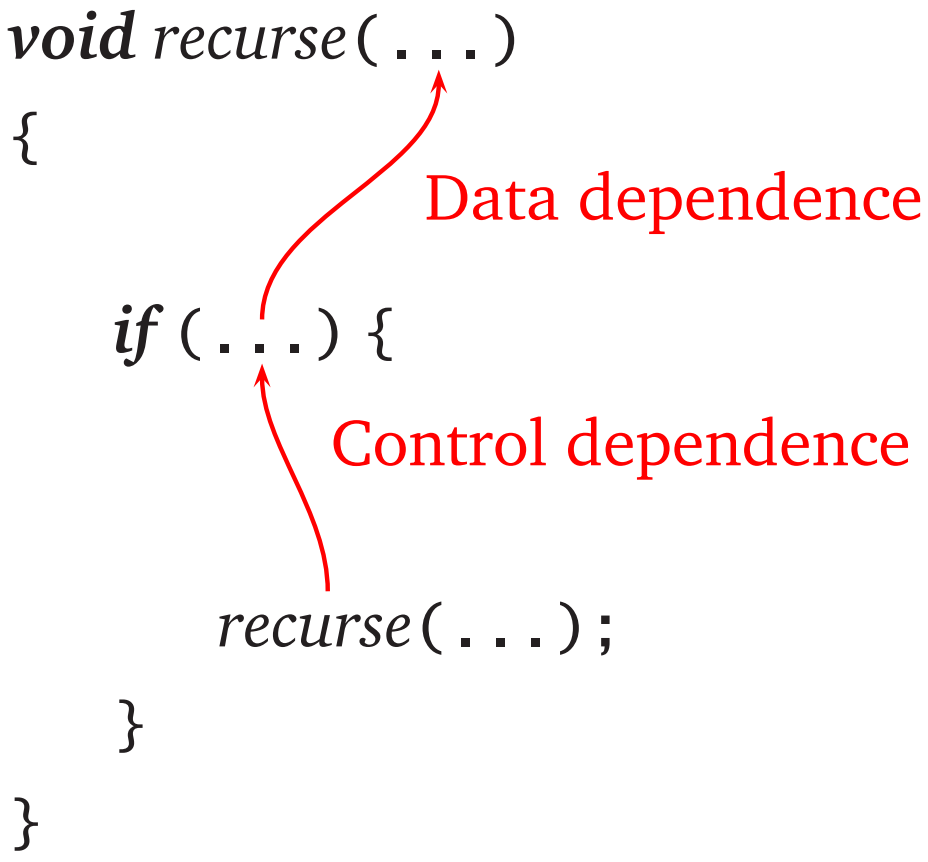
3. Inline “trivial” functions

   Objective: eliminate redundant code

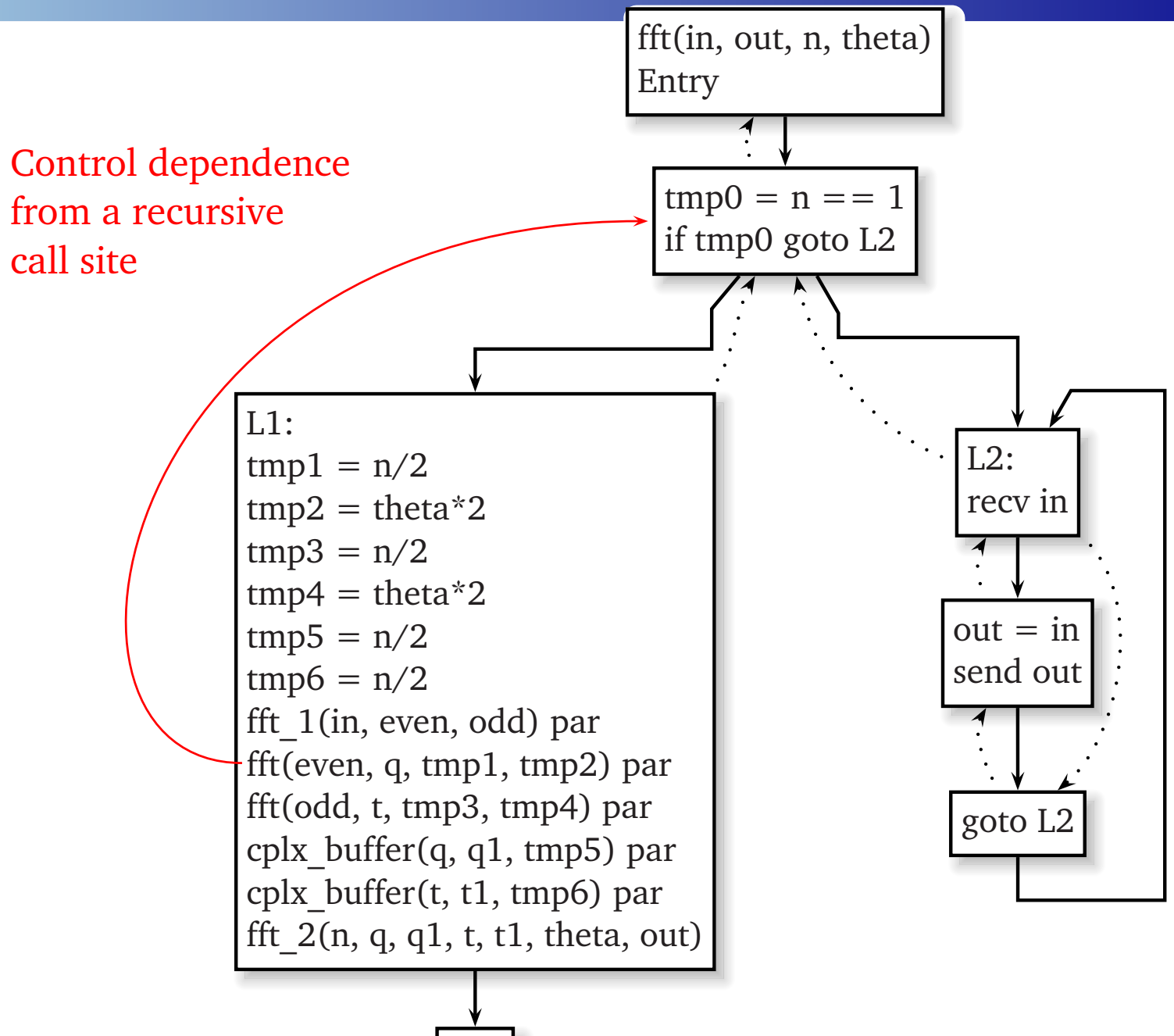
   Intuition: recursive functions often become just calls

Relevant Variables from Slicing

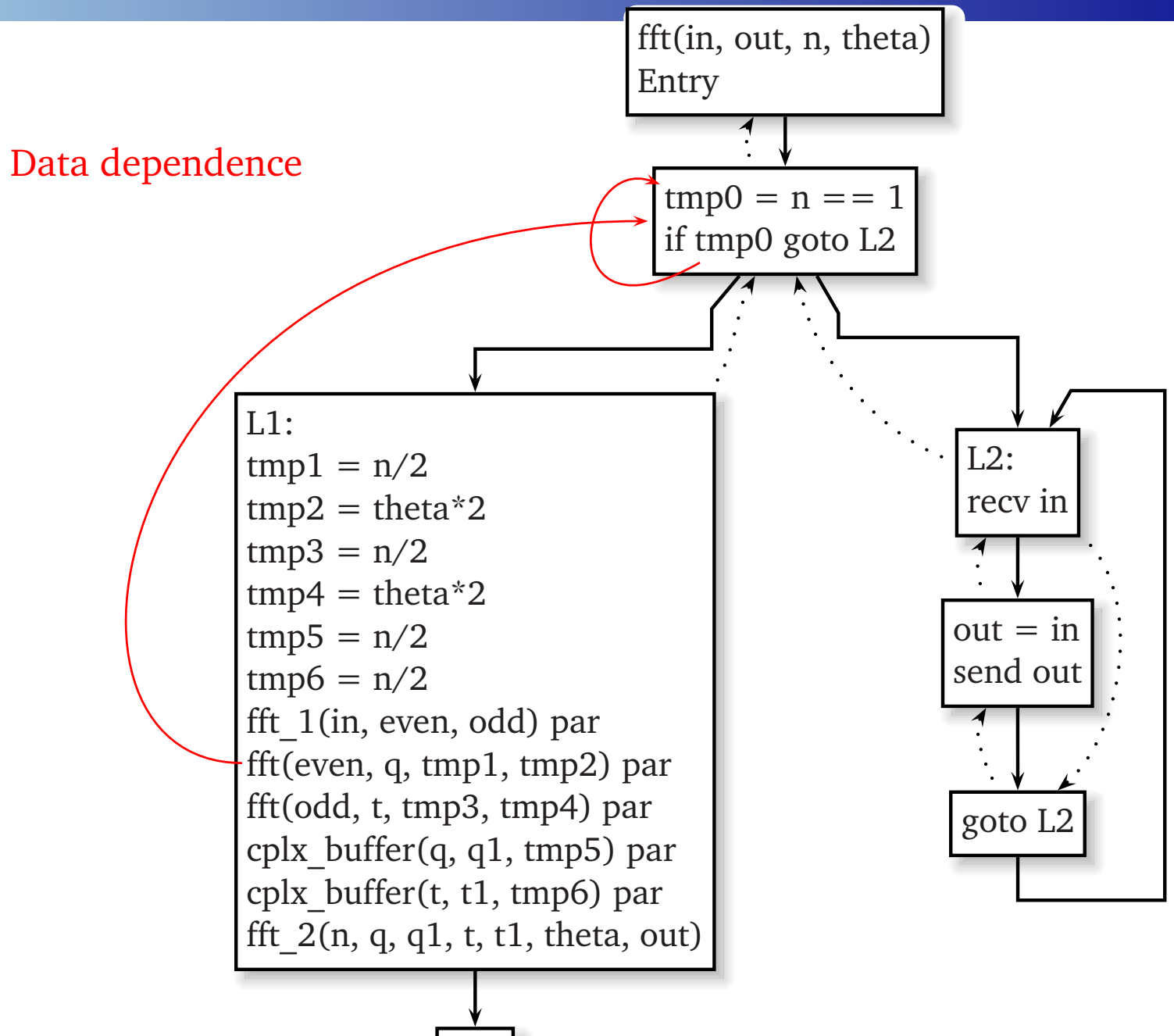
Insight: only conditional control flow stops recursion



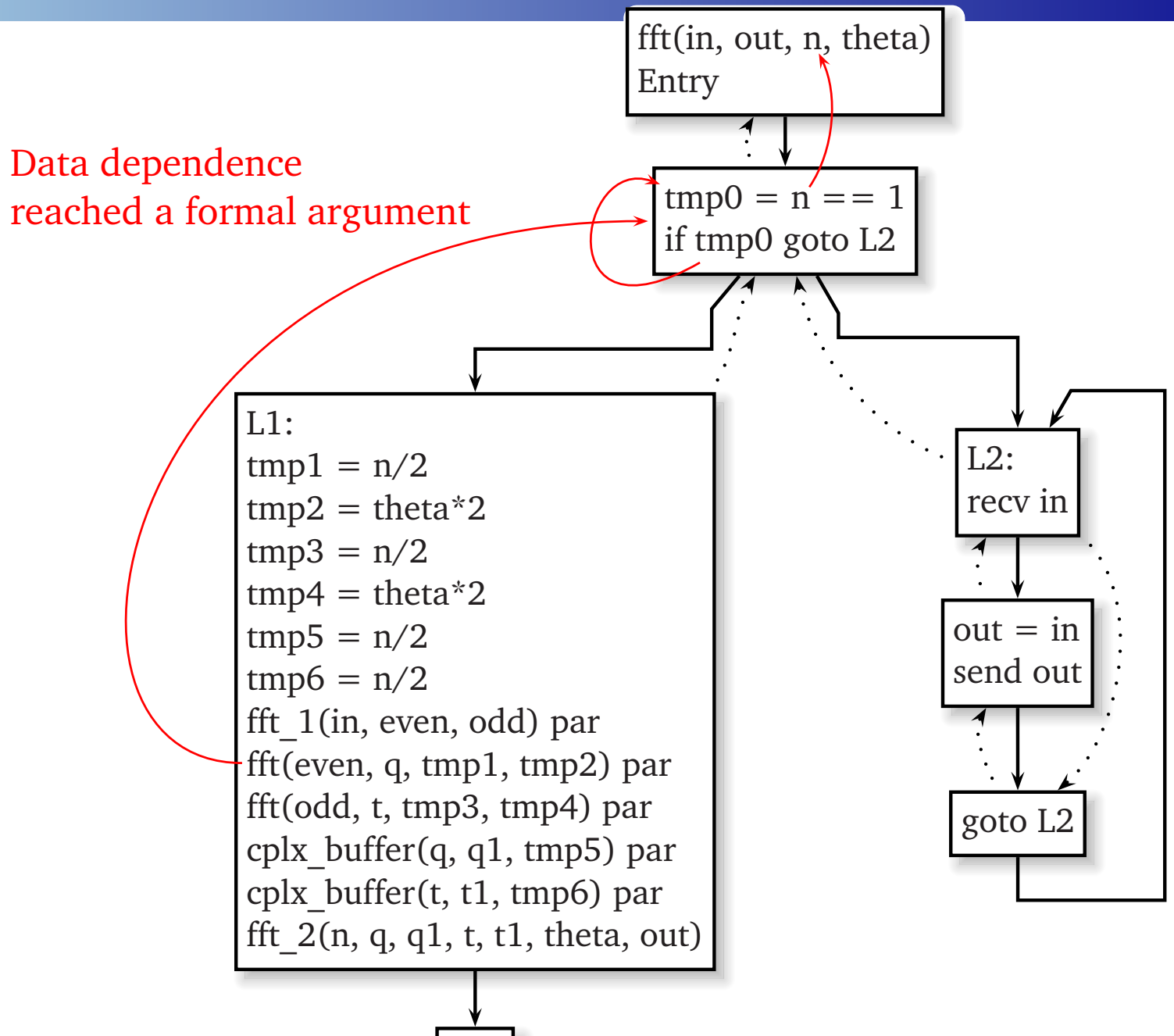
Binding-Time Analysis: Slicing



Binding-Time Analysis: Slicing

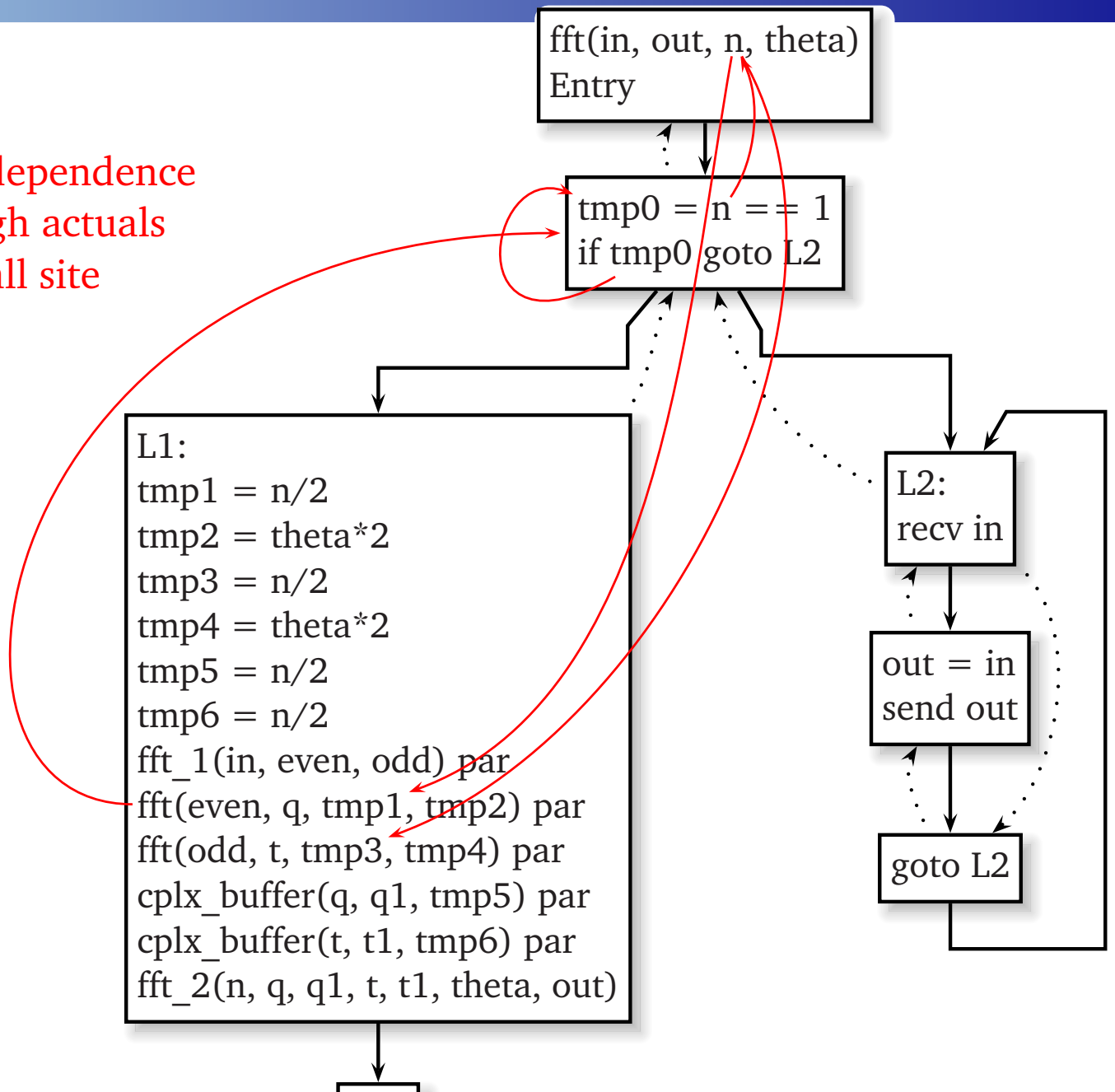


Binding-Time Analysis: Slicing

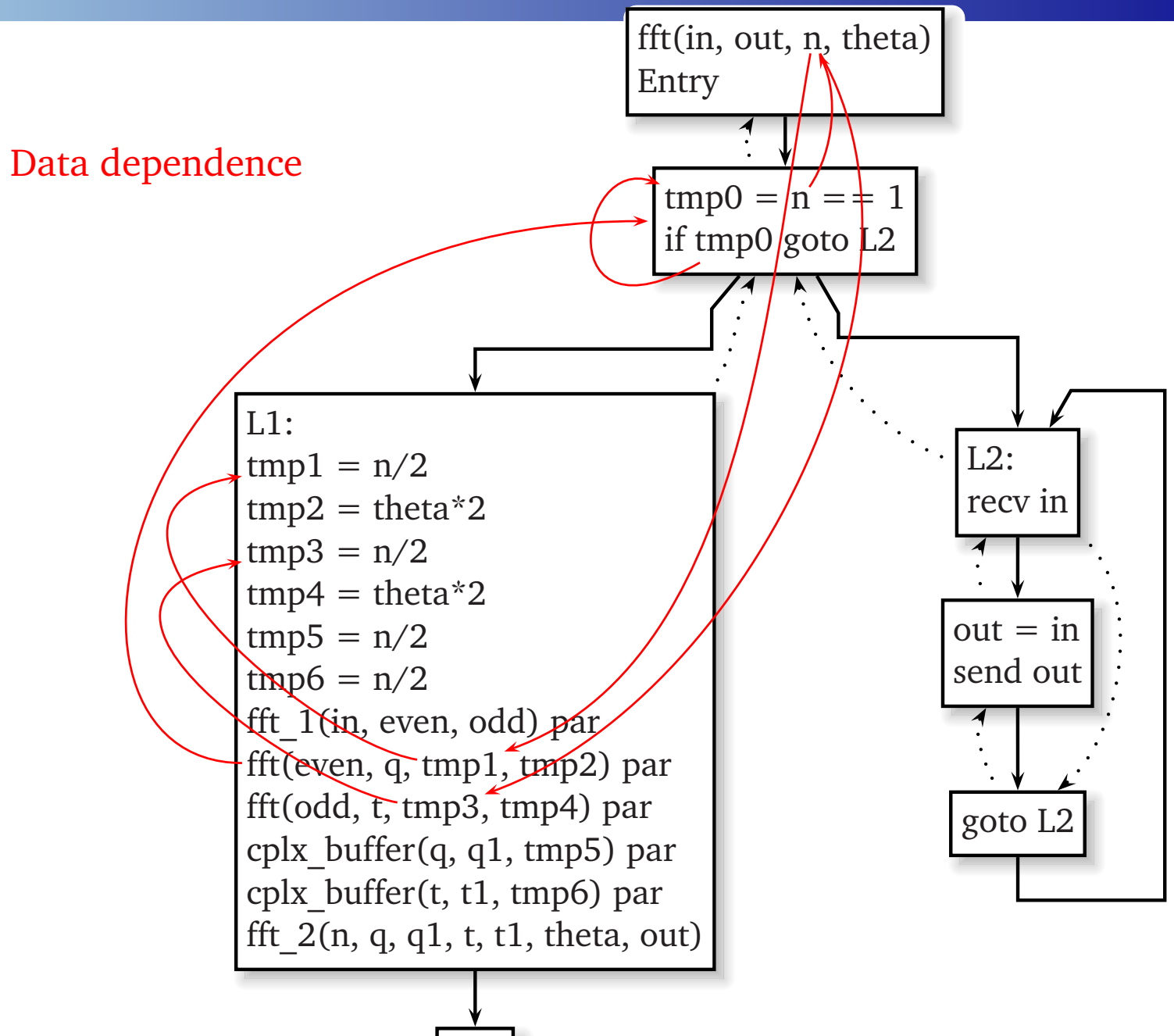


Binding-Time Analysis: Slicing

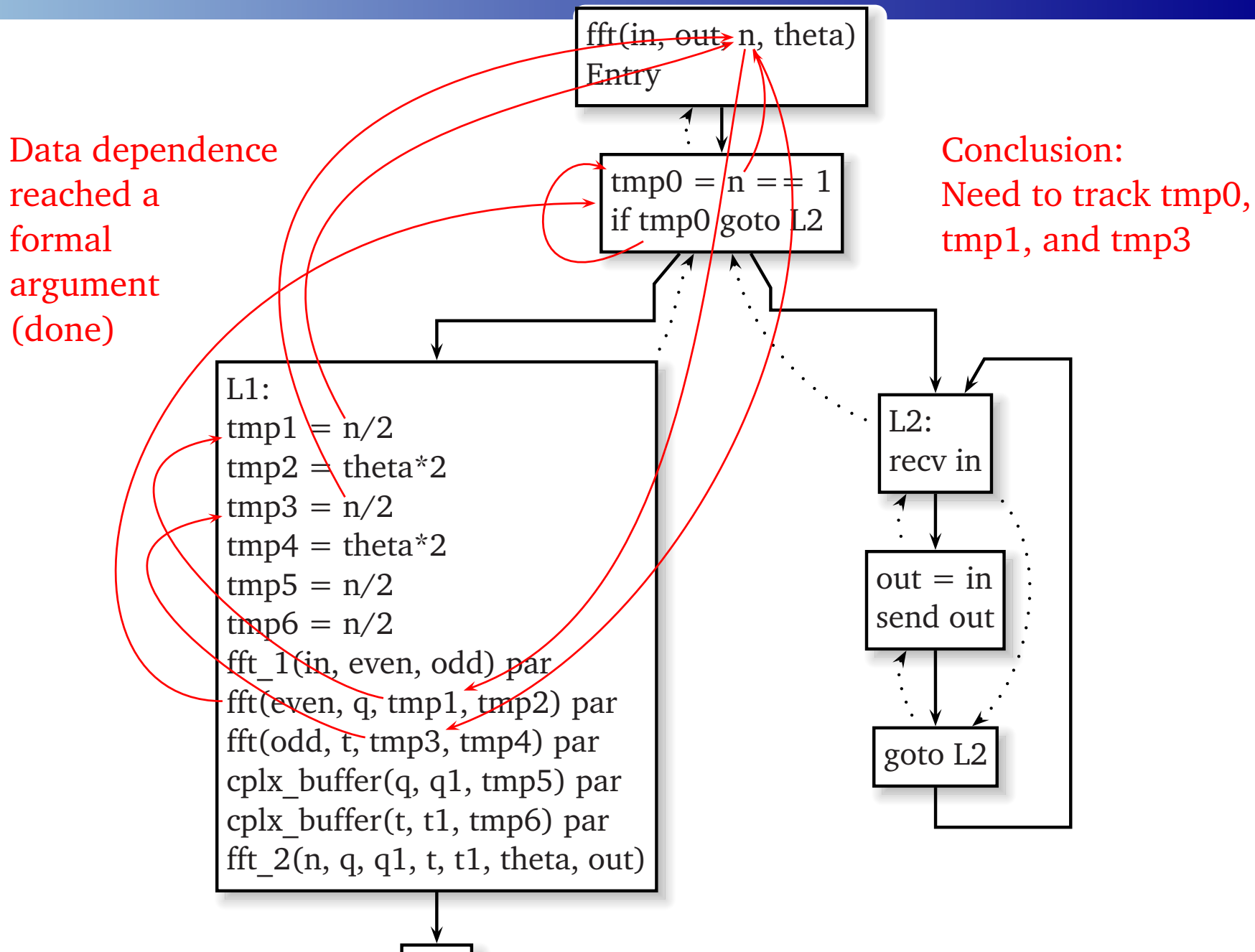
Data dependence
through actuals
at a call site



Binding-Time Analysis: Slicing



Binding-Time Analysis: Slicing



Partial Evaluation: Simulation

Insight: exhaustively simulate basic blocks with partial information about variables

Original block

```
recurse(i)
t1 = i == j
if t1 goto L1
```

Knowing i=3

```
recurse(3)
t1 = 3 == j
if t1 goto L1
```

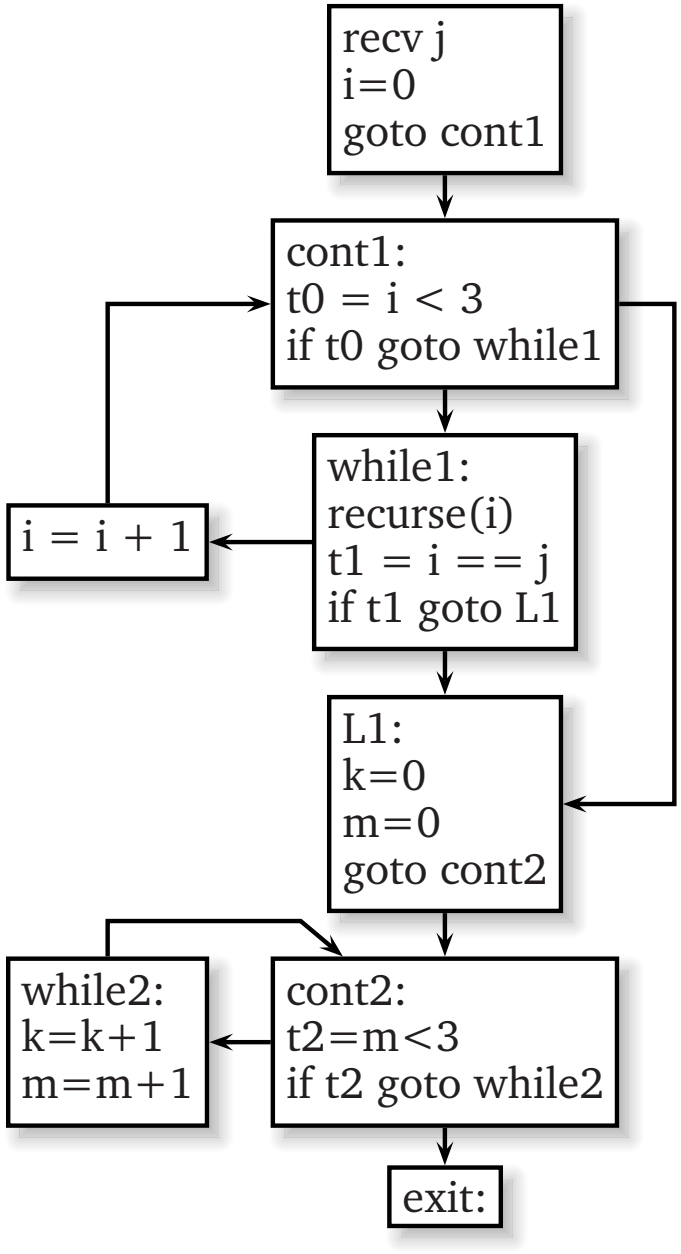
Knowing i=3 & j=3

```
recurse(3)


goto L1
```

Minimize duplication by forgetting variables when they cease to be live

Simulation Example: Basic Blocks

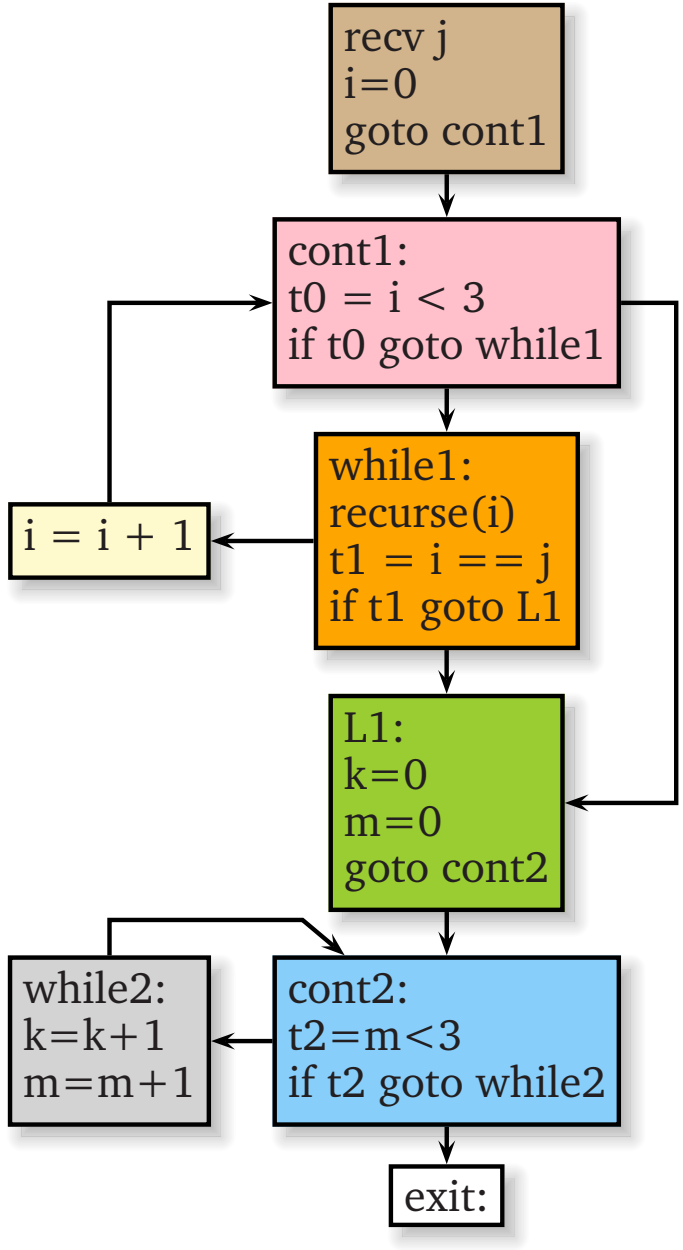
```
void foo(chan int j) {
 recv j;
 for (int i = 0 ; i < 3 ; i++) {
    recurse(i);
    if (i == j) break;
 }
 int k = 0;
 for (int m = 0 ; m < 3 ; m++)
   k += m;
}

void recurse(int i) {
   if (i) recurse (i–1);
}
```



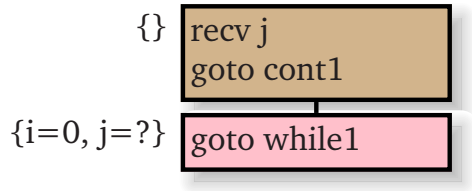
Simulation: i, j, and t0 relevant

{} recv j
goto cont1

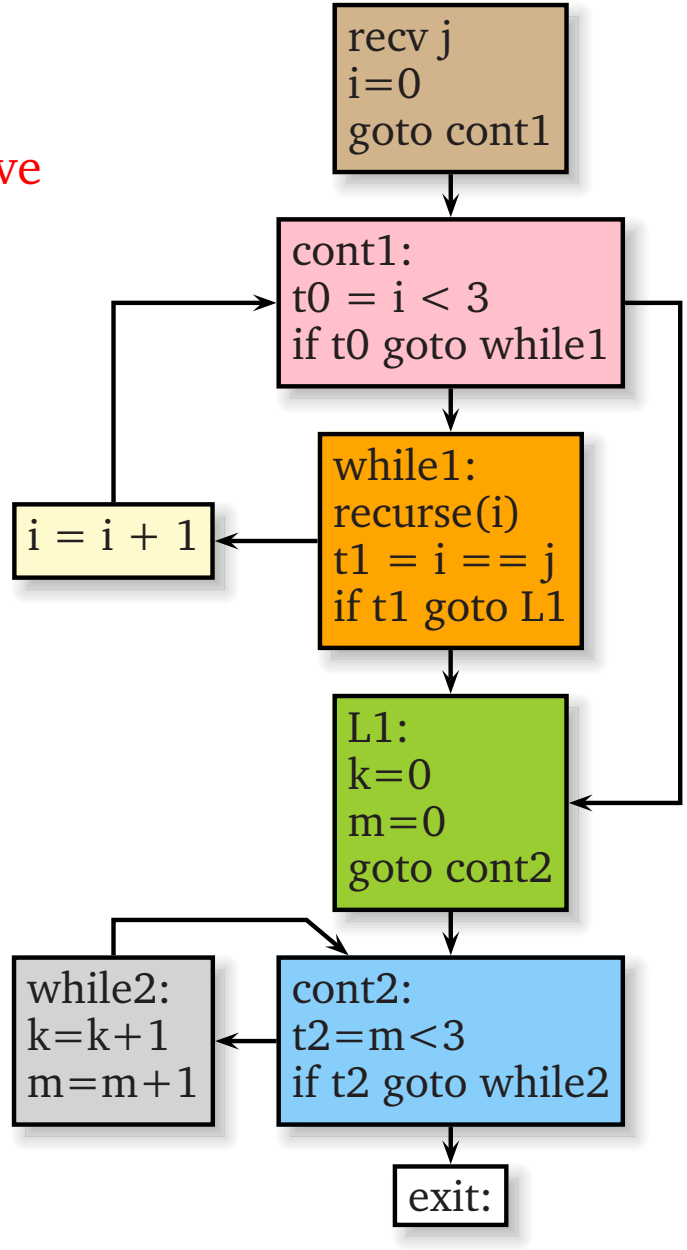
i becomes 0



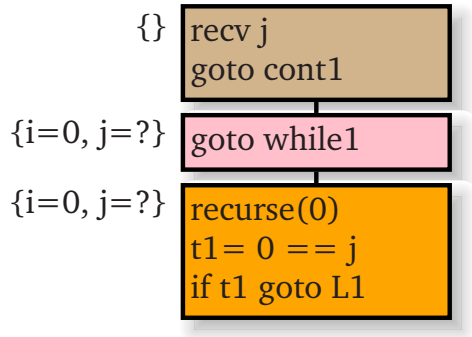
Simulation: i, j, and t0 relevant



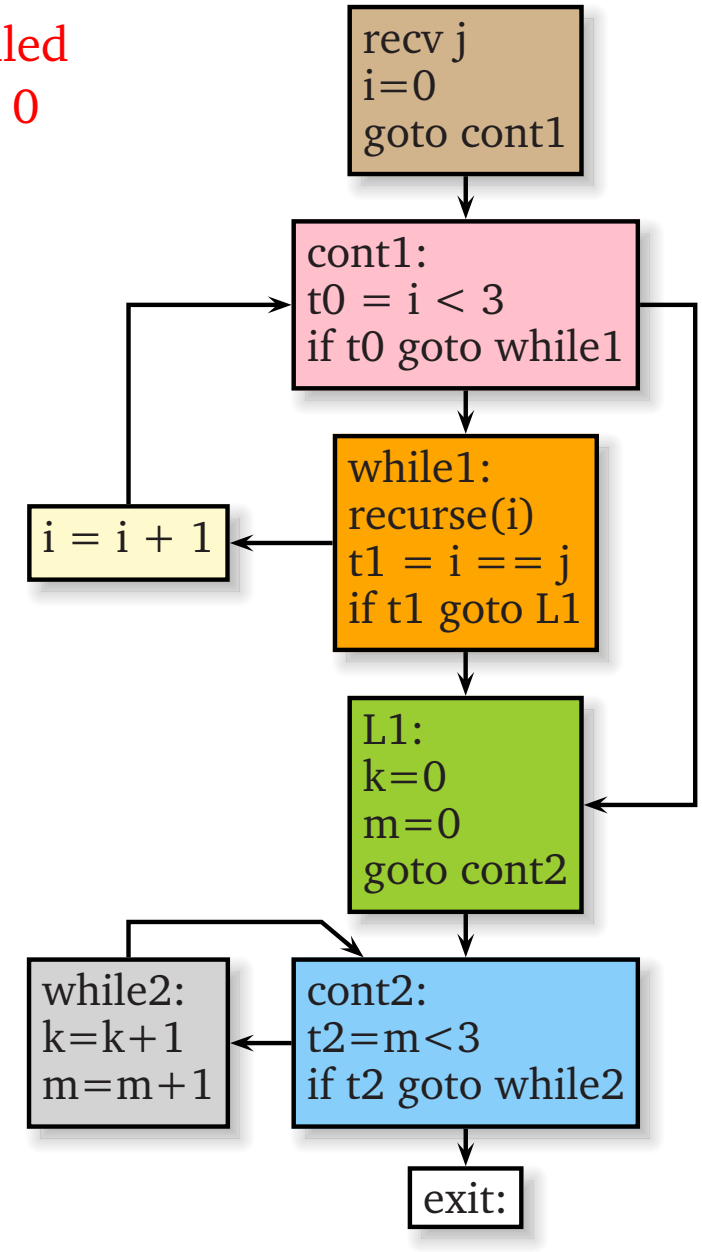
t0 evaluated
branch taken
t0 no longer live



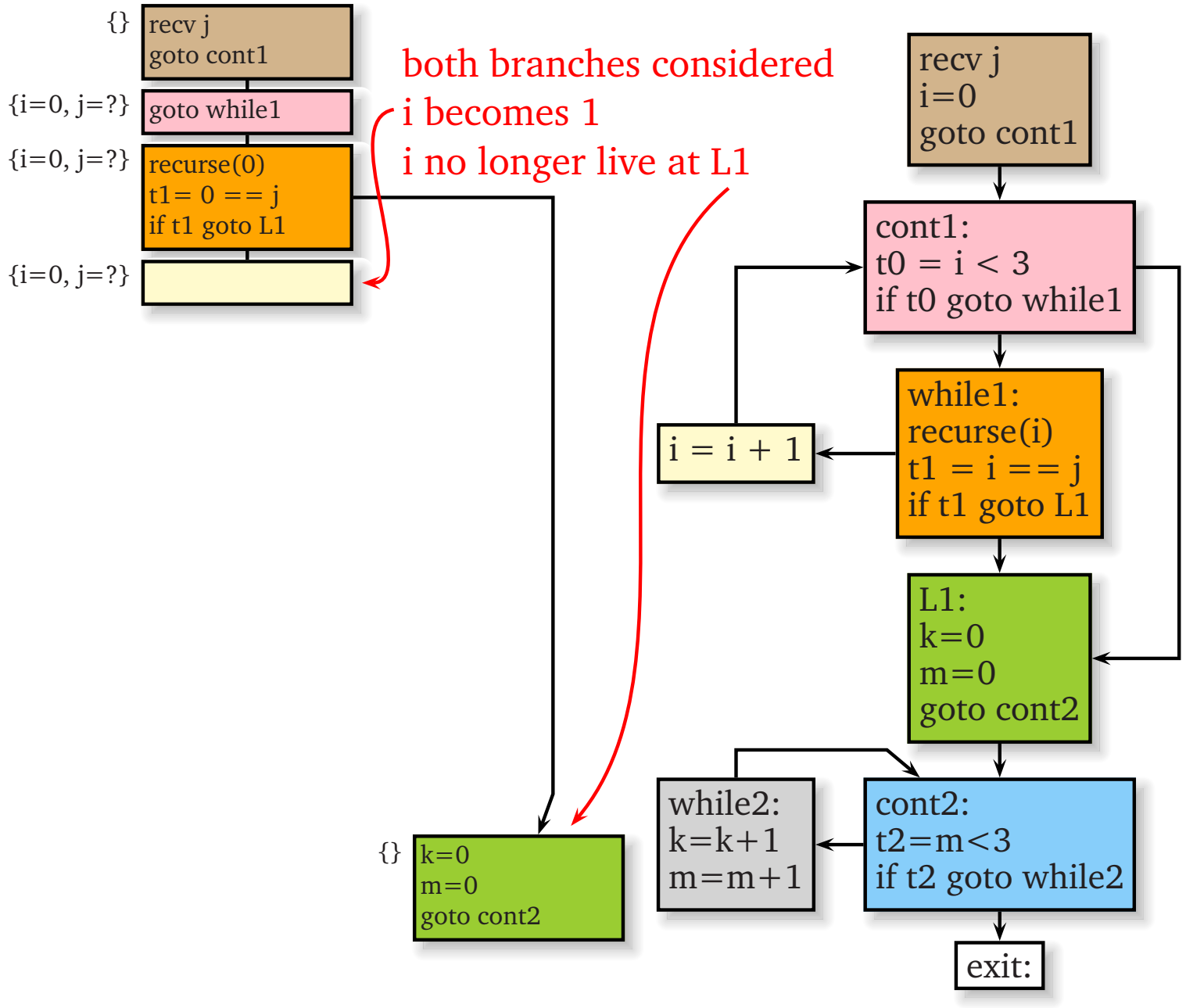
Simulation: i, j, and t0 relevant



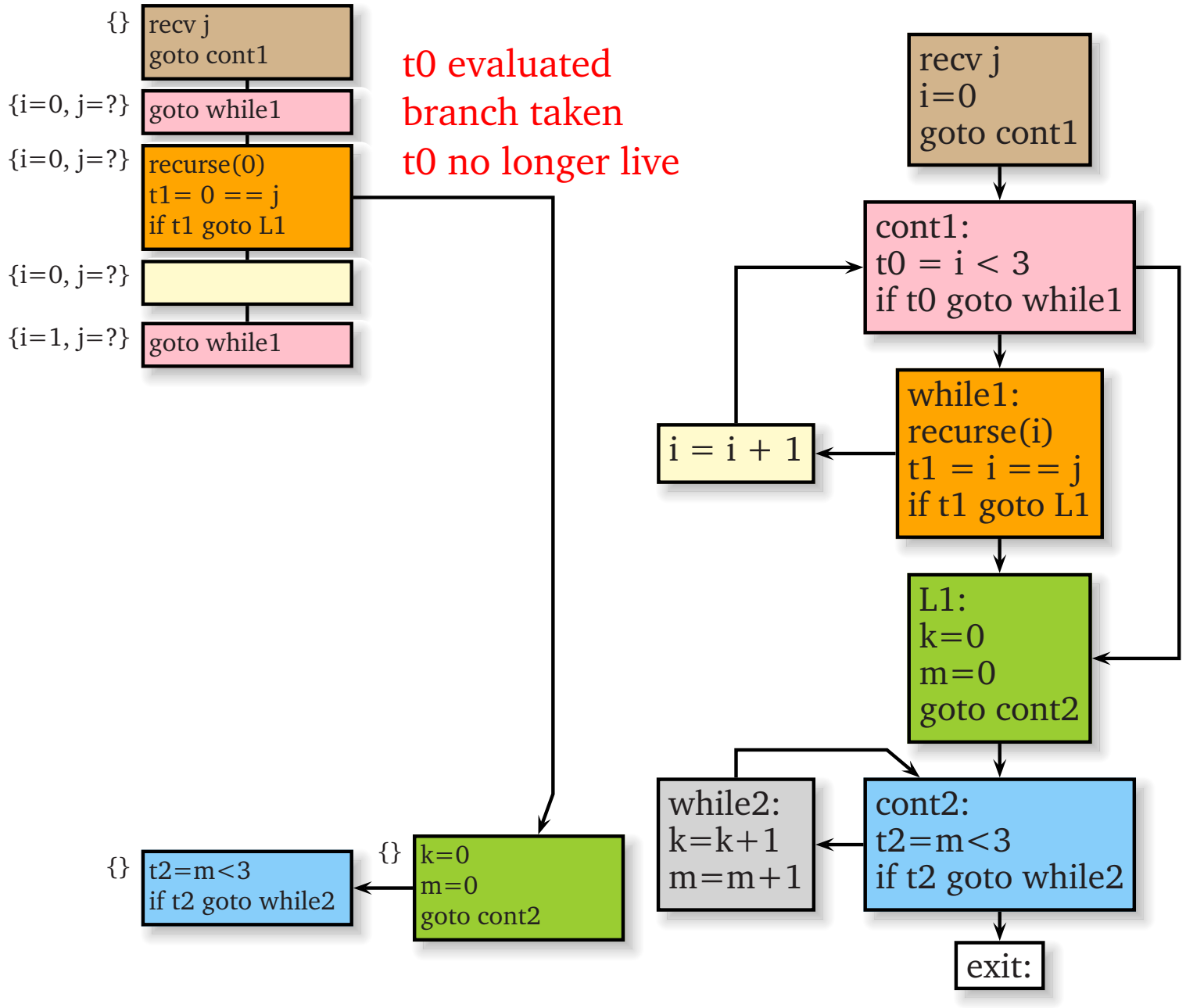
recurse(0) called
i known to be 0



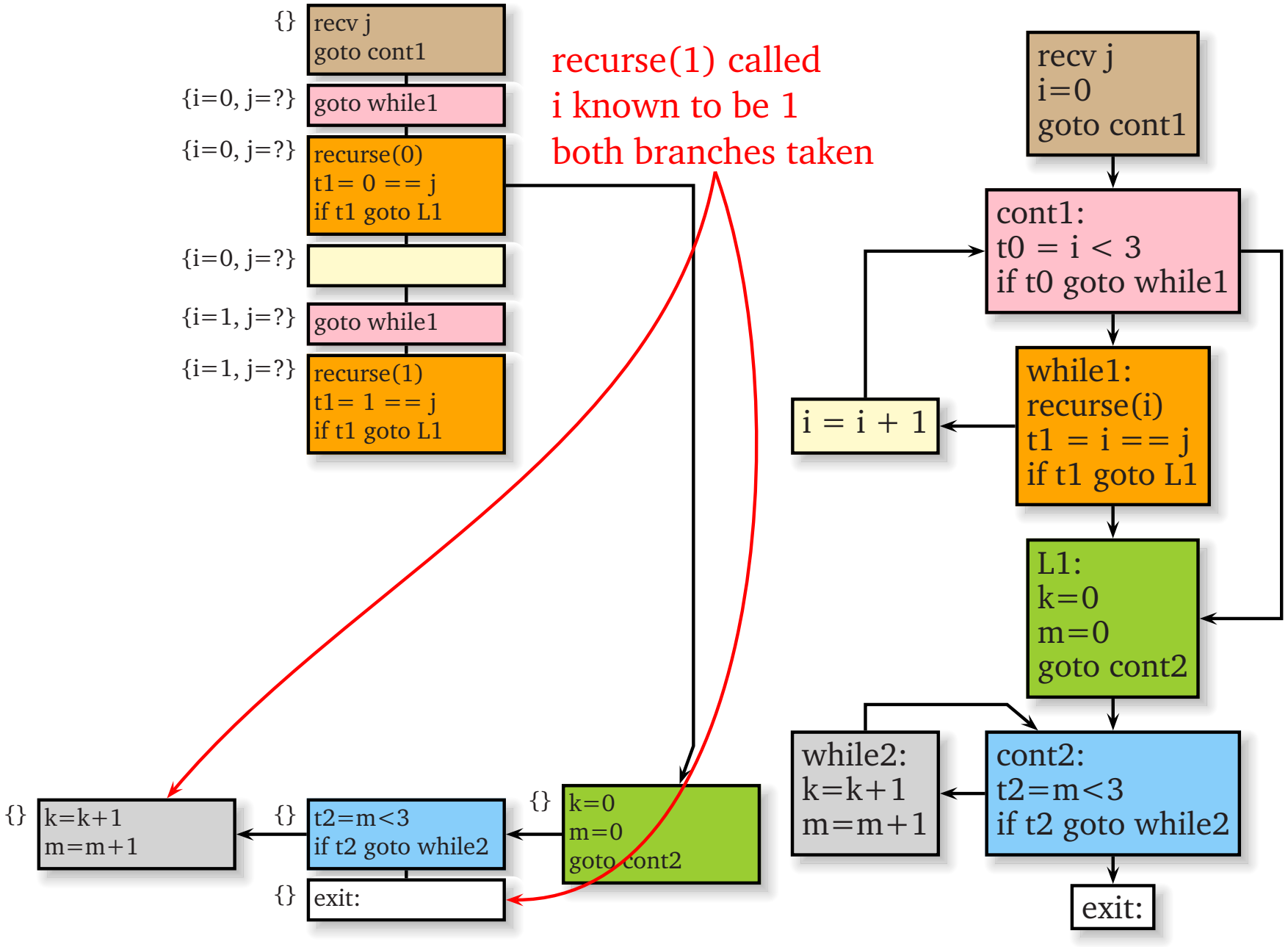
Simulation: i, j, and t0 relevant



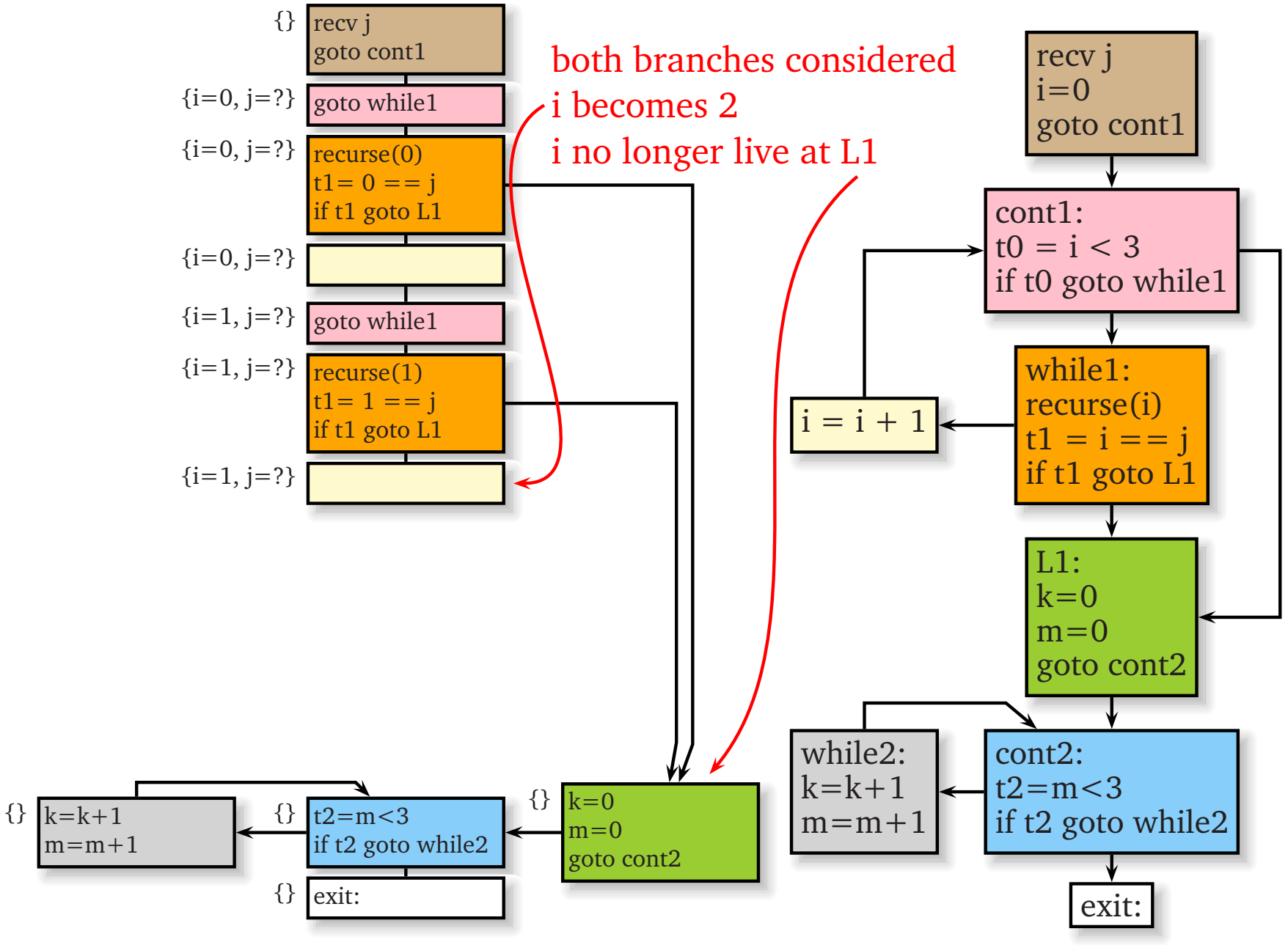
Simulation: i, j, and t0 relevant



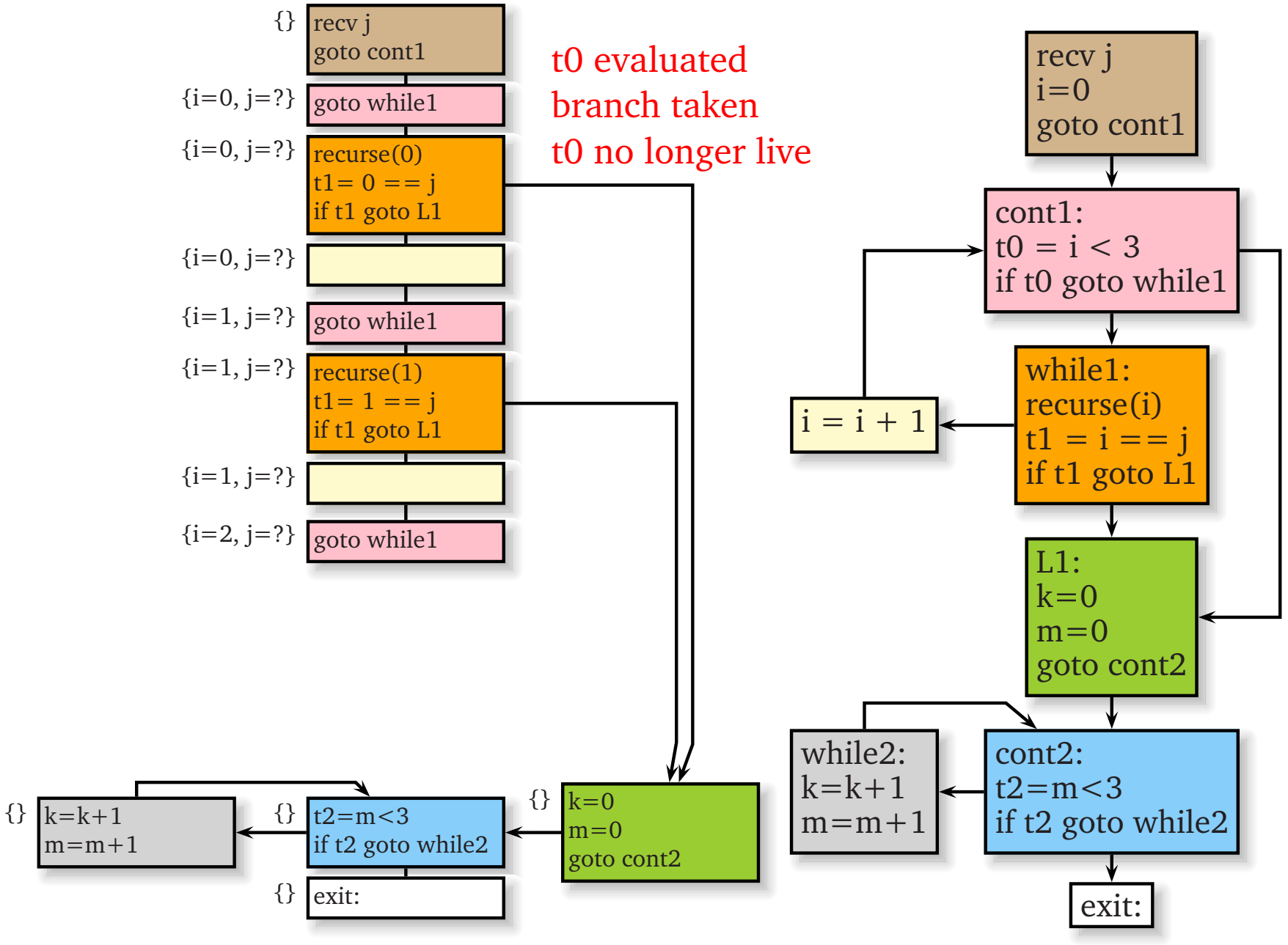
Simulation: i, j, and t0 relevant



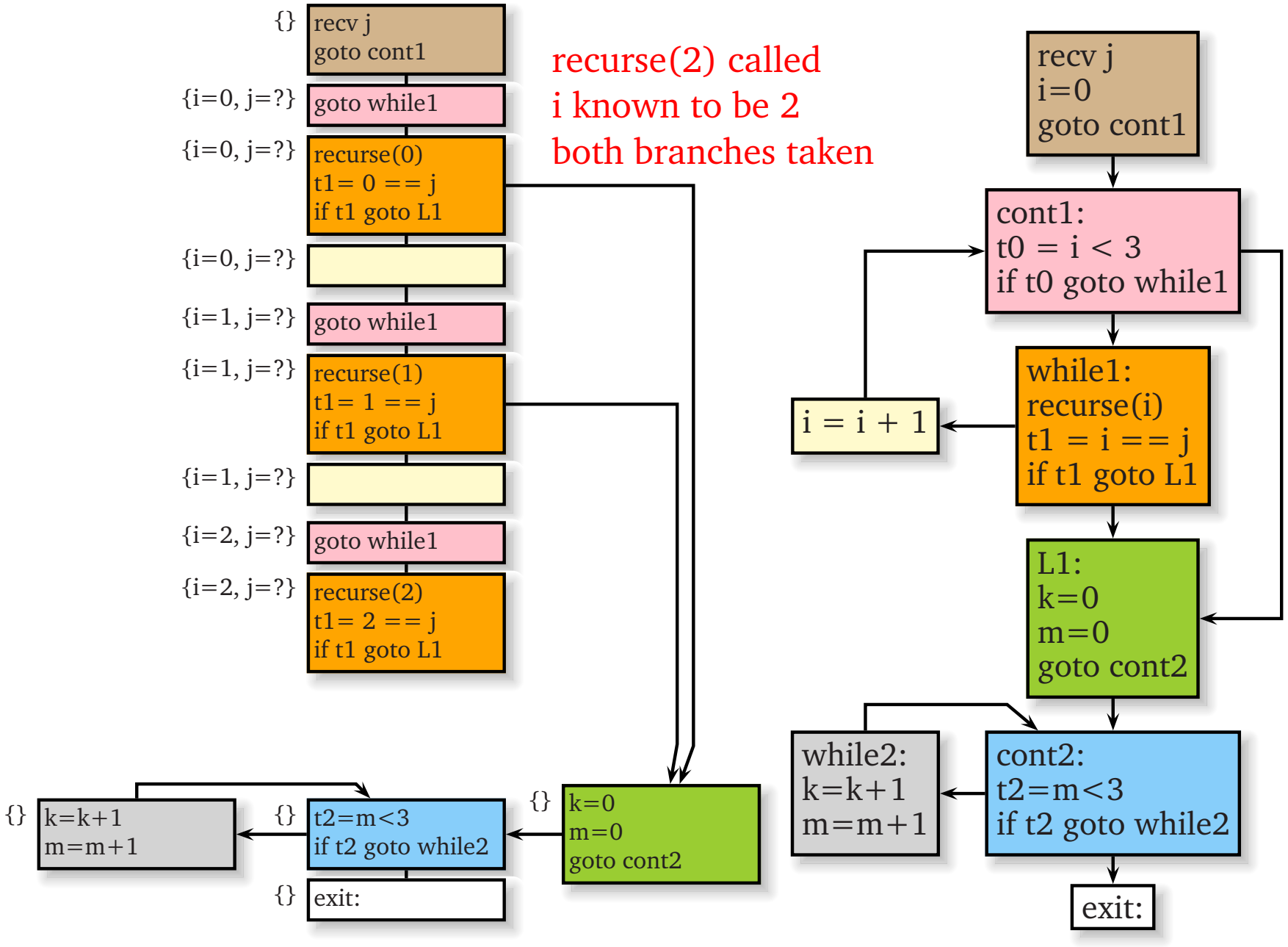
Simulation: i, j, and t0 relevant



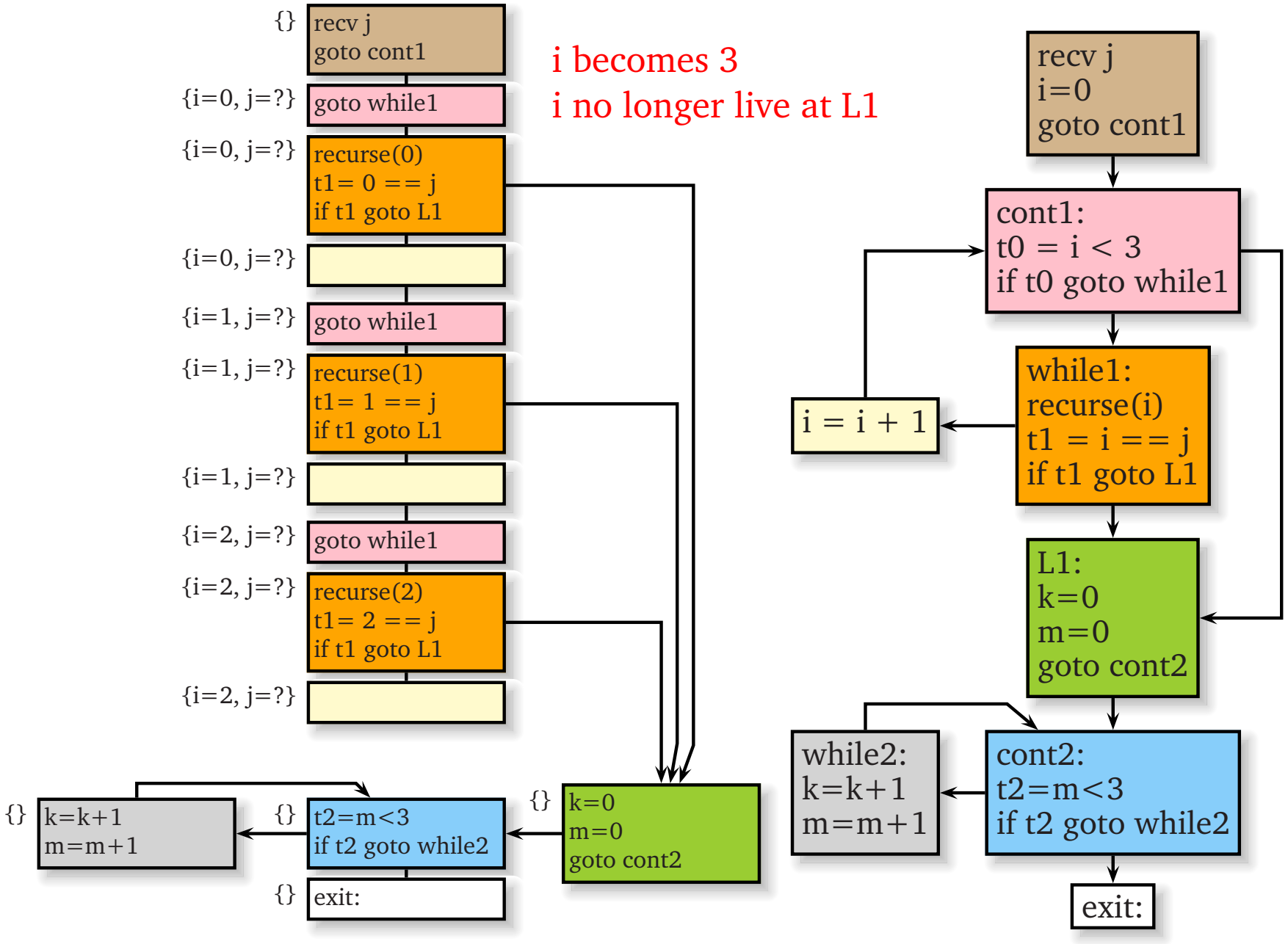
Simulation: i, j, and t0 relevant



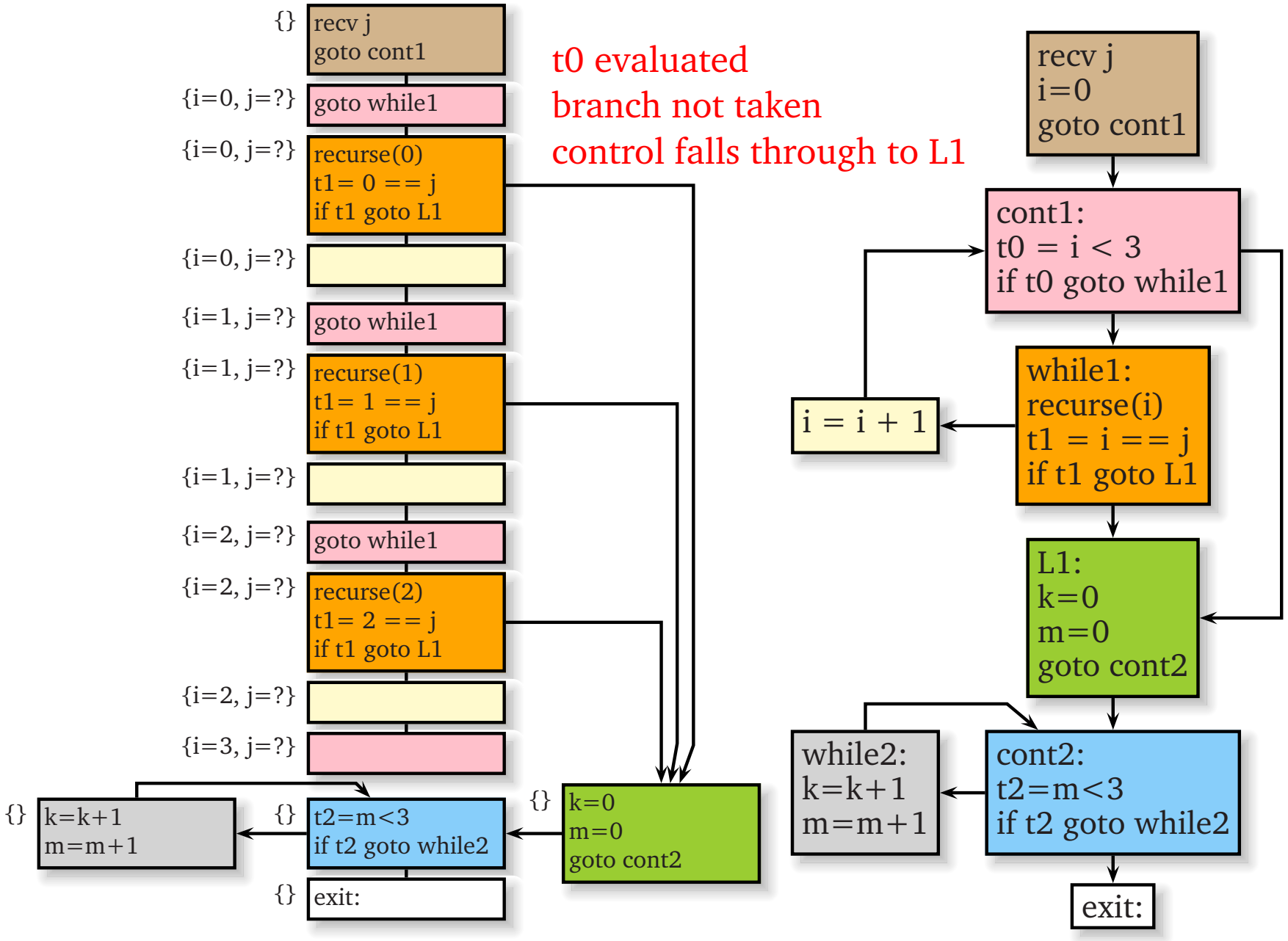
Simulation: i, j, and t0 relevant



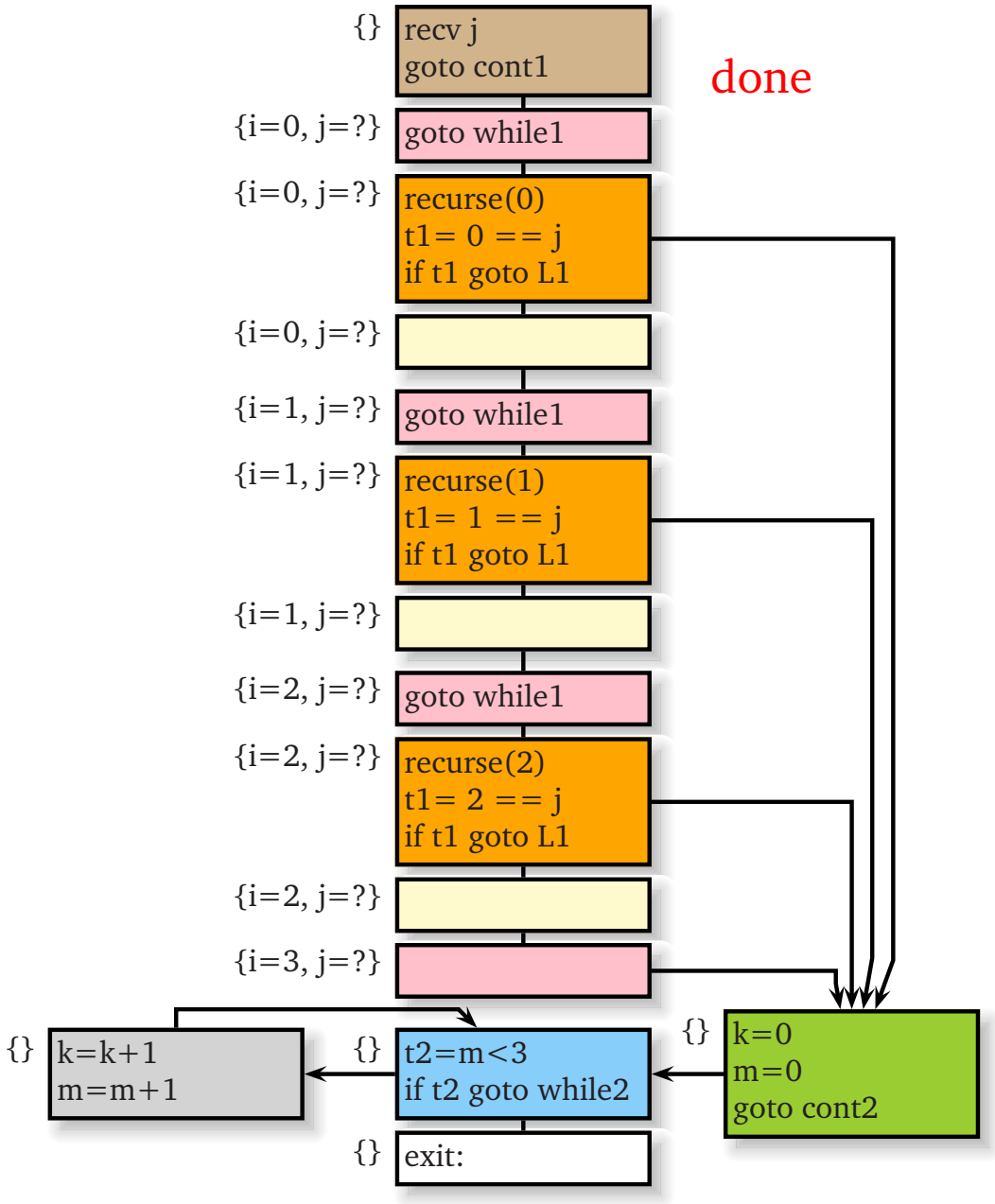
Simulation: i, j, and t0 relevant

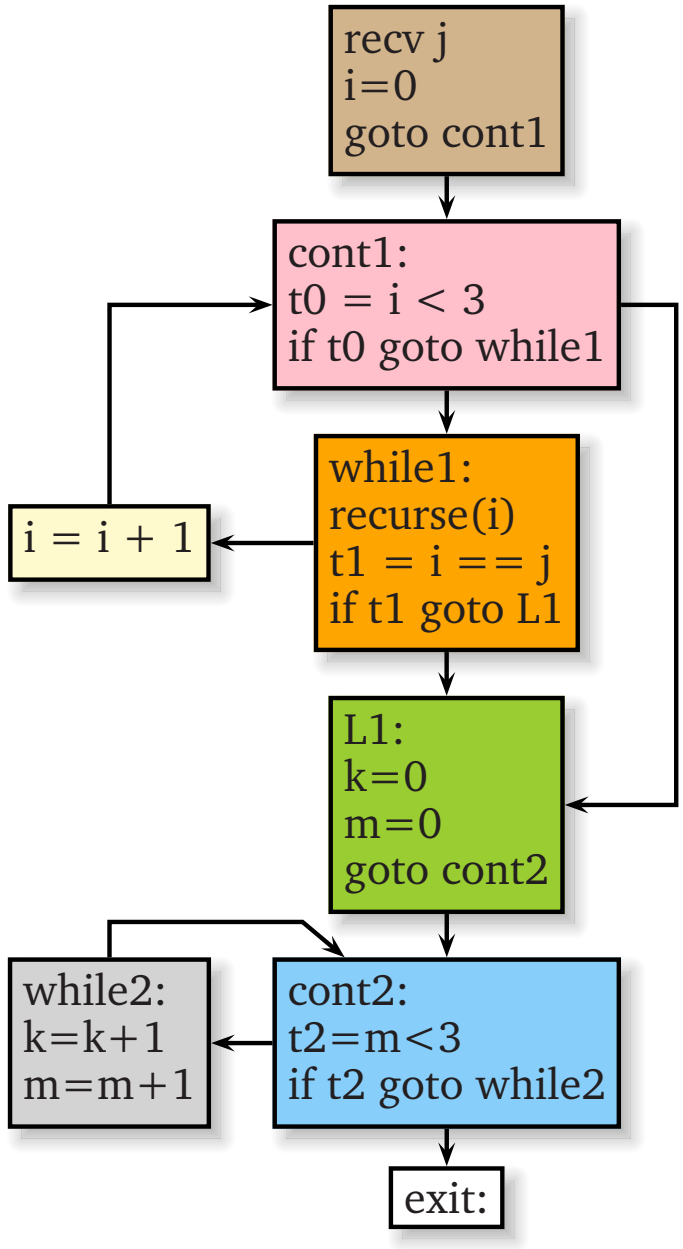


Simulation: i, j, and t0 relevant



Simulation: i, j, and t0 relevant





Experiments

Ran on six small examples in paper

Compared

Source Original, recursive code

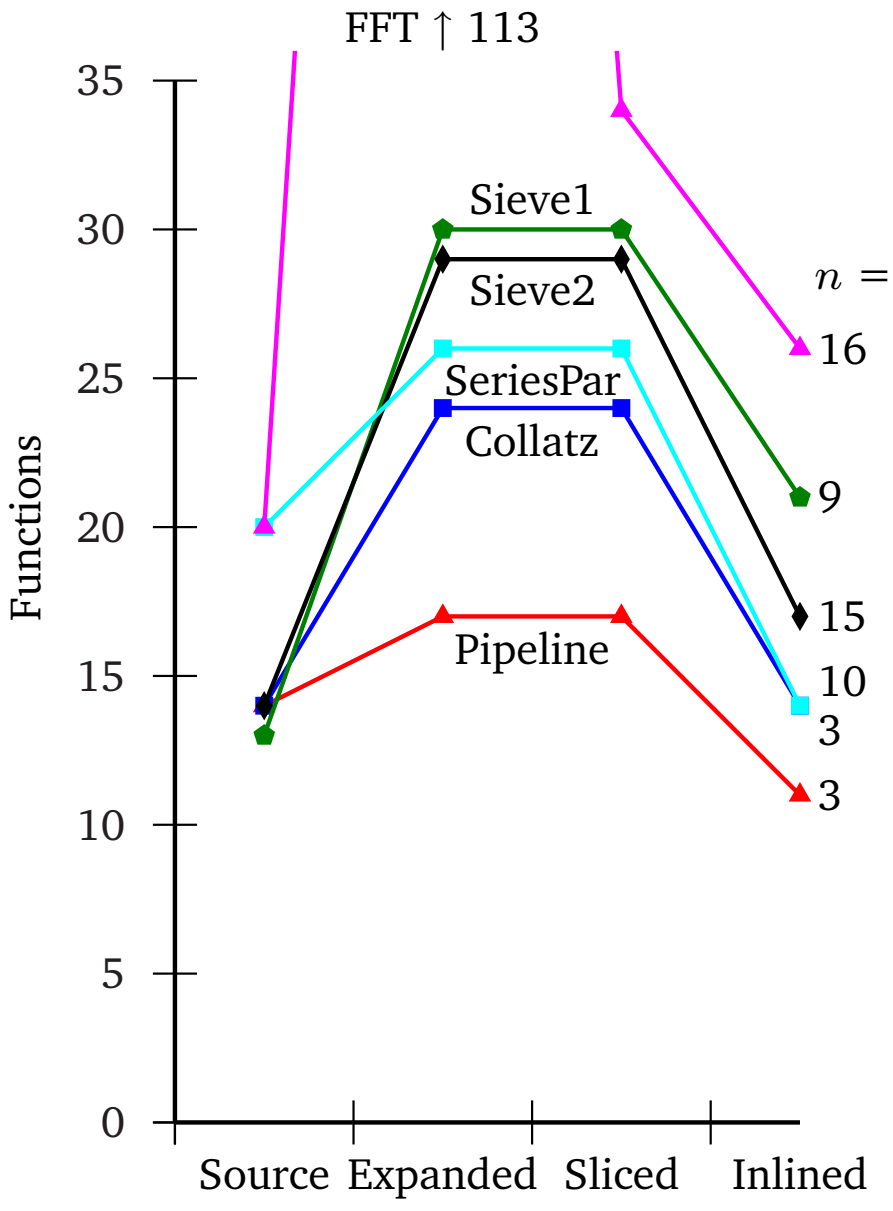
Expanded All variables relevant, depth limited to 32

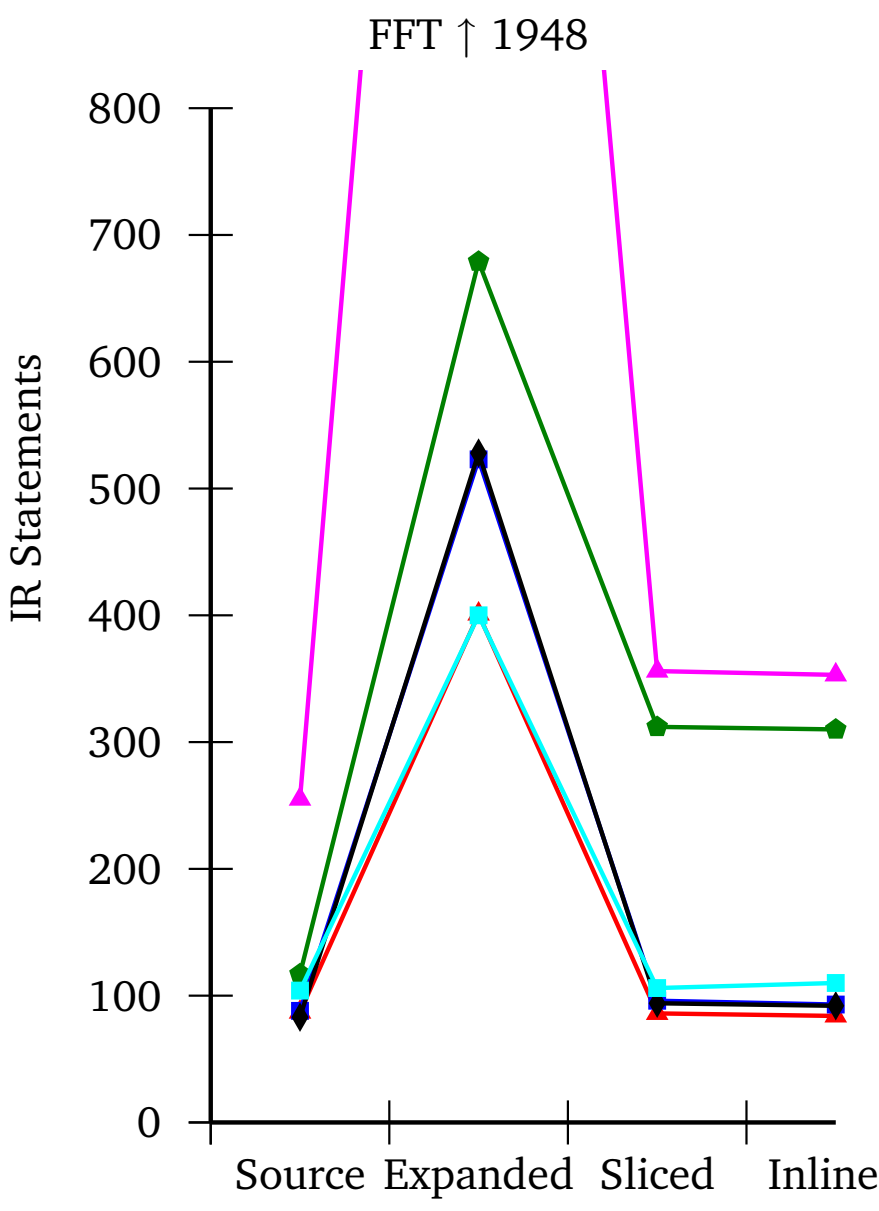
Sliced Only consider variables deemed relevant by slicing

Inlined Post-processed to remove “just call” functions

Runtimes were all under a second

Experimental Results





Conclusions

- Recursion useful for building parallel structures
- Our technique can statically resolve “structural” recursion for, e.g., hardware synthesis
- Slicing-based binding-time analysis
- Simulation-based Partial Evaluation
- Technique is fast and code does not explode