Programming Language Laboratory

Code Optimization Techniques for Graphics Processing Units

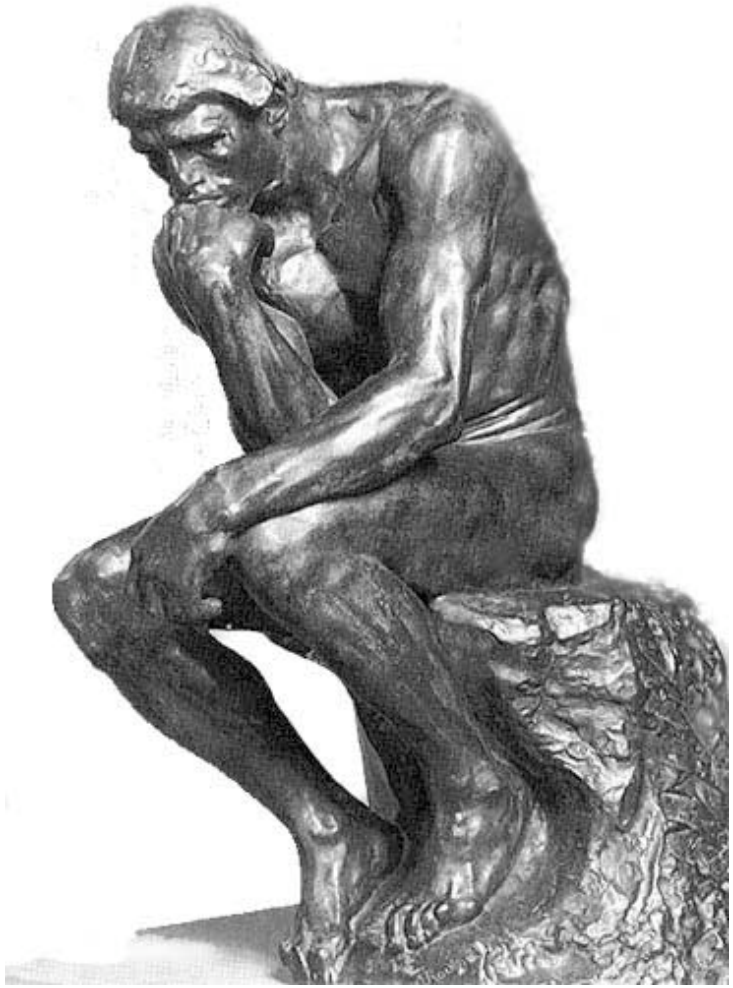
Fernando Magno Quintão Pereira

fernando@dcc.ufmg.br

fernando@dcc.ufmg.br

Divergence Analysis and Optimizations

- How does SIMD execution handle branches?
- How to detect divergences during program execution?
- How to predict the branches that will diverge?
- How to manually optimize code to mitigate their negative effects?



fernando@dcc.ufmg.br

The Kingdom of Paralland

- In a galaxy far, far away, live the **Locksteppers**.
- The locksteppers are cool, easy going creatures, who live in a *monastic* and *harmonious* society.
- But, they suffer from a weird **idiosyncrasy**…

fernando@dcc.ufmg.br

They **must** eat together…

fernando@dcc.ufmg.br

They **must** pray together...

fernando@dcc.ufmg.br

They **must** dance together…

fernando@dcc.ufmg.br

For that is the law:

"All the locksteppers who do something at a given point in time, must do the same thing."

The God of the Locksteppers

fernando@dcc.ufmg.br

But, sometimes they disagree…

- Then, we have two parties: **righties** and **lefties**.
- According to the law, **righties** must sleep, while **lefties** do stuff.
  - When the **lefties** are done, they must sleep, while the **righties** do stuff.
  - Until they re-converge.

fernando@dcc.ufmg.br

The Paralland Mission

Decrease the amount of time Locksteppers sleep!

- No easy task: Locksteppers disagree a lot.
- How to accomplish this mission?
- What does Paralland has to do with GPUs?

fernando@dcc.ufmg.br

Paralland and GPU's

- Each Lockstepper is a warp **thread**.
- Divergences – or disagreements – happen because of **branches**.

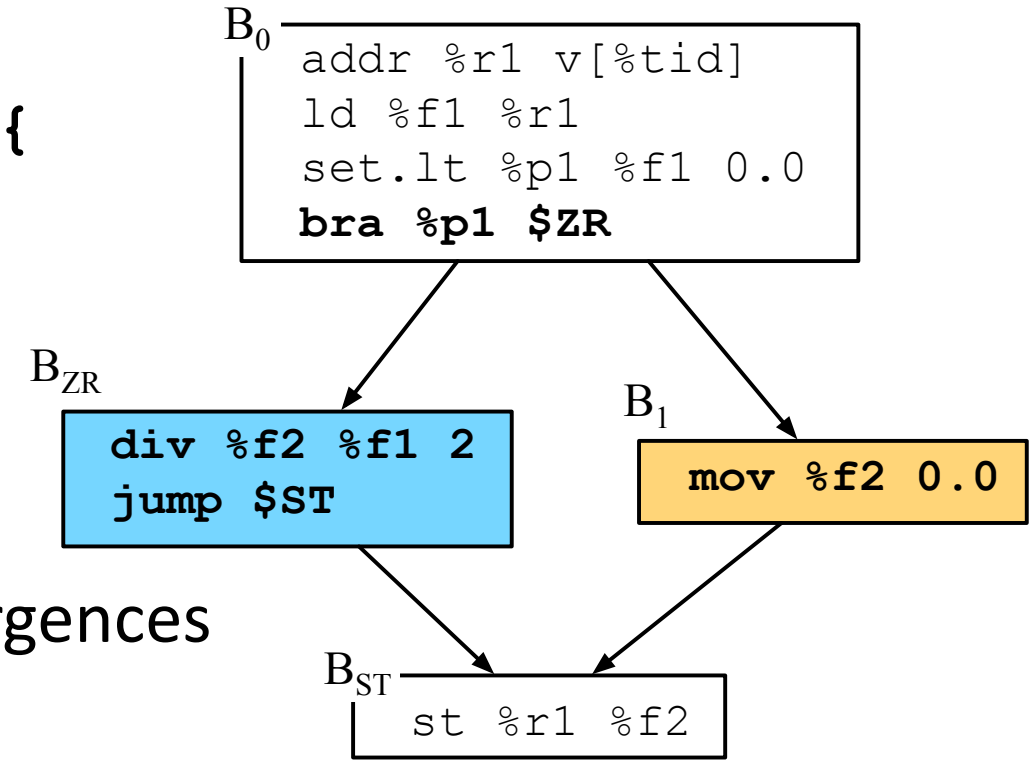
- We have a **SIMD** model of parallel execution.

fernando@dcc.ufmg.br

Example of Divergent CFG

- Below we have a simple *kernel*, and its **C**ontrol **F**low **G**raph:

```
__global__ void
ex (float* v) {
  if (v[tid] < 0.0) {
    v[tid] /= 2;
  } else {
    v[tid] = 0.0;
  }
}
```



- Why do we have divergences in this kernel?

fernando@dcc.ufmg.br

What does the divergence do?

program counter	label	op	def	use_1	use_2	ALU_1	ALU_2
1	B_0	addr	%r1	v	[%tid]	✓	✓
2		ld	%f1	%r1		✓	✓
3		set.lt	%p1	%f1	0.0	✓	✓
4		bra	%p1	$ZR		✓	✓
5	B_{ZR}	**div**	**%f2**	**%f1**	**2**	●	✓
6		**jump**		**$ST**		●	✓
7	B_1	**mov**	**%f2**	**0.0**		✓	●
8	B_{ST}	st		%r1	%f2	✓	✓

fernando@dcc.ufmg.br

To wrap up definitions…

- Why are divergences a problem in the GPU, while they are not even a concern in multi-core CPUs?
- How does latency and throughput suffer with divergences?
  - What are the bad cases in terms of performance degradation?
- Can you think about a way to detect divergences during the program execution?

fernando@dcc.ufmg.br

The Kernels of Samuel

- What is the best input for the kernel below?

```
__global__ void dec2zero(int* v, int N) {
  int xIndex = blockIdx.x*blockDim.x+threadIdx.x;
  if (xIndex < N) {
    while (v[xIndex] > 0) {
      v[xIndex]--;
    }
  }
}
```

From `decPar`, available in the course webpage.

fernando@dcc.ufmg.br

Trying different inputs

```
void vecIncInit(int* data, int size) {
  for (int i = 0; i < size; ++i) {
    data[i] = size - i - 1;
  }
}
```

1

```
void vecConsInit(int* data, int size) {
  int cons = size / 2;
  for (int i = 0; i < size; ++i) {
    data[i] = cons;
  }
}
```

2

```
void vecAltInit(int* data, int size) {
  for (int i = 0; i < size; ++i) {
    if (i % 2) {
      data[i] = size;
    }
  }
}
```

3

```
void vecRandomInit(int* data, int size) {
  for (int i = 0; i < size; ++i) {
    data[i] = random() % size;
  }
}
```

4

```
void vecHalfInit(int* data, int size) {
  for (int i = 0; i < size/2; ++i) {
    data[i] = 0;
  }
  for (int i = size/2; i < size; ++i) {
    data[i] = size;
  }
}
```

5

- What is the best way to initialize the kernel, thus pleasing Samuel?

fernando@dcc.ufmg.br

Samuelic array 1

```
void vecIncInit(int* data, int size) {
  for (int i = 0; i < size; ++i) {
    data[i] = size - i - 1;
  }
}
```

1

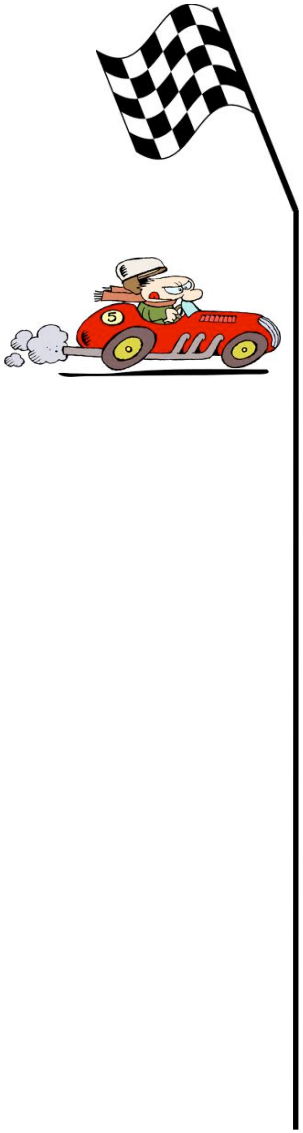
```
void vecConsInit(int* data, int size) {
  int cons = size / 2;
  for (int i = 0; i < size; ++i) {
    data[i] = cons;
  }
}
```

```
void vecAltInit(int* data, int size) {
  for (int i = 0; i < size; ++i) {
    if (i % 2) {
      data[i] = size;
    }
  }
}
```

```
void vecRandomInit(int* data, int
size) {
  for (int i = 0; i < size; ++i) {
    data[i] = random() % size;
  }
}
```

```
void vecHalfInit(int* data, int size) {
  for (int i = 0; i < size/2; ++i) {
    data[i] = 0;
  }
  for (int i = size/2; i < size; ++i) {
    data[i] = size;
  }
}
```

SUM: 20480000
TIME: 16250



fernando@dcc.ufmg.br

Samuelic array 2

```
void vecIncInit(int* data, int size) {
  for (int i = 0; i < size; ++i) {
    data[i] = size - i - 1;
  }
}
```

1

```
void vecConsInit(int* data, int size) {
  int cons = size / 2;
  for (int i = 0; i < size; ++i) {
    data[i] = cons;
  }
}
```

2

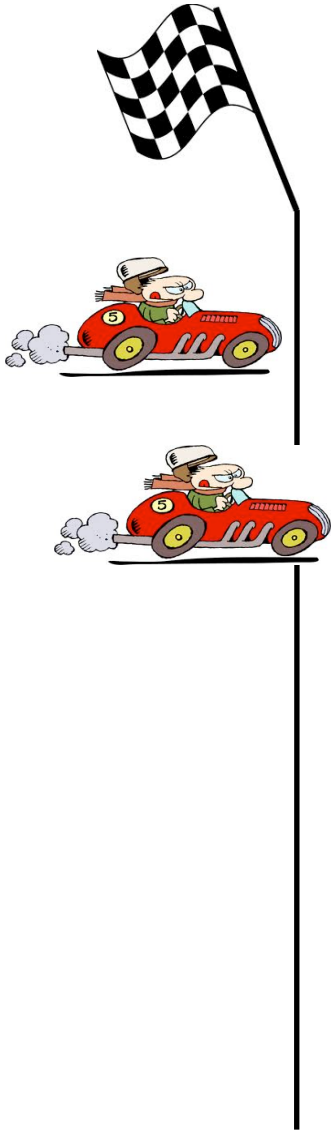
```
void vecAltInit(int* data, int size) {
  for (int i = 0; i < size; ++i) {
    if (i % 2) {
      data[i] = size;
    }
  }
}
```

```
void vecRandomInit(int* data, int
size) {
  for (int i = 0; i < size; ++i) {
    data[i] = random() % size;
  }
}
```

```
void vecHalfInit(int* data, int size) {
  for (int i = 0; i < size/2; ++i) {
    data[i] = 0;
  }
  for (int i = size/2; i < size; ++i) {
    data[i] = size;
  }
}
```

SUM: 20480000
TIME: 16250

SUM: 20480000
TIME: 16153



fernando@dcc.ufmg.br

Samuelic array 3

```
void vecIncInit(int* data, int size) {
  for (int i = 0; i < size; ++i) {
    data[i] = size - i - 1;
  }
}
```

1

```
void vecConsInit(int* data, int size) {
  int cons = size / 2;
  for (int i = 0; i < size; ++i) {
    data[i] = cons;
  }
}
```

2

```
void vecAltInit(int* data, int size) {
  for (int i = 0; i < size; ++i) {
    if (i % 2) {
      data[i] = size;
    }
  }
}
```

3

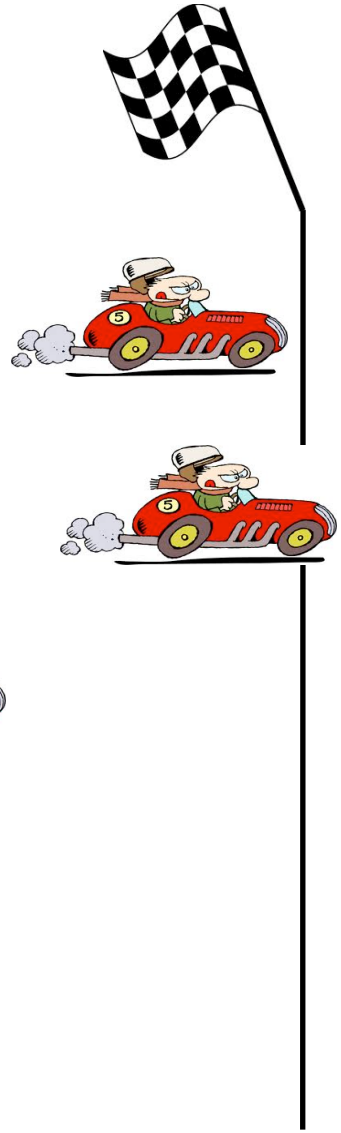
```
void vecRandomInit(int* data, int
size) {
  for (int i = 0; i < size; ++i) {
    data[i] = random() % size;
  }
}
```

```
void vecHalfInit(int* data, int size) {
  for (int i = 0; i < size/2; ++i) {
    data[i] = 0;
  }
  for (int i = size/2; i < size; ++i) {
    data[i] = size;
  }
}
```

SUM: 20480000
TIME: 16250

SUM: 20480000
TIME: 16153

SUM: 20476800
TIME: 32193



fernando@dcc.ufmg.br

Samuelic array 4

```
void vecIncInit(int* data, int size) {
  for (int i = 0; i < size; ++i) {
    data[i] = size - i - 1;
  }
}
```

1

```
void vecConsInit(int* data, int size) {
  int cons = size / 2;
  for (int i = 0; i < size; ++i) {
    data[i] = cons;
  }
}
```

2

```
void vecAltInit(int* data, int size) {
  for (int i = 0; i < size; ++i) {
    if (i % 2) {
      data[i] = size;
    }
  }
}
```

3

```
void vecRandomInit(int* data, int
size) {
  for (int i = 0; i < size; ++i) {
    data[i] = random() % size;
  }
}
```

4

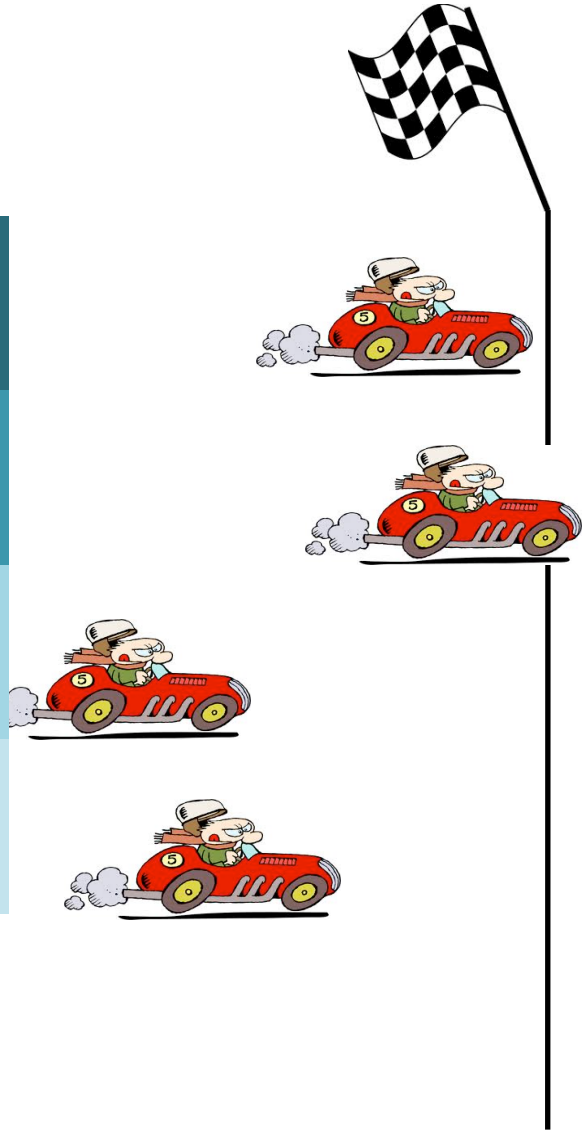
```
void vecHalfInit(int* data, int size) {
  for (int i = 0; i < size/2; ++i) {
    data[i] = 0;
  }
  for (int i = size/2; i < size; ++i) {
    data[i] = size;
  }
}
```

SUM: 20480000
TIME: 16250

SUM: 20480000
TIME: 16153

SUM: 20476800
TIME: 32193

SUM: 20294984
TIME: 30210



fernando@dcc.ufmg.br

Samuelic array 5

```
void vecIncInit(int* data, int size) {
  for (int i = 0; i < size; ++i) {
    data[i] = size - i - 1;
  }
}
```

1

```
void vecConsInit(int* data, int size) {
  int cons = size / 2;
  for (int i = 0; i < size; ++i) {
    data[i] = cons;
  }
}
```

2

```
void vecAltInit(int* data, int size) {
  for (int i = 0; i < size; ++i) {
    if (i % 2) {
      data[i] = size;
    }
  }
}
```

3

```
void vecRandomInit(int* data, int
size) {
  for (int i = 0; i < size; ++i) {
    data[i] = random() % size;
  }
}
```

4

```
void vecHalfInit(int* data, int size) {
  for (int i = 0; i < size/2; ++i) {
    data[i] = 0;
  }
  for (int i = size/2; i < size; ++i) {
    data[i] = size;
  }
}
```

5

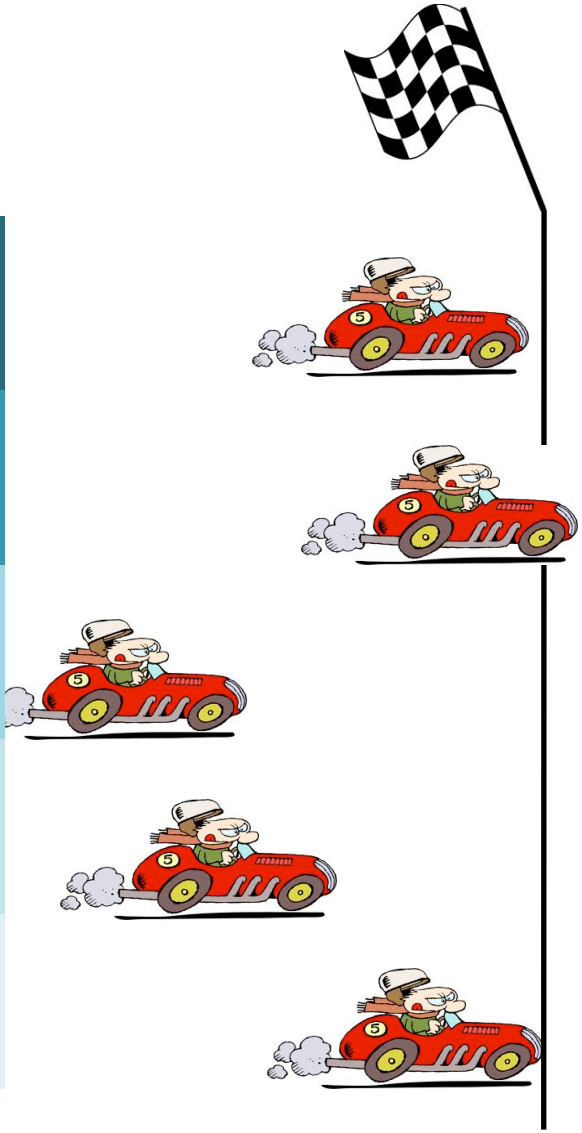
SUM: 20480000
TIME: 16250

SUM: 20480000
TIME: 16153

SUM: 20476800
TIME: 32193

SUM: 20294984
TIME: 30210

SUM: 20480000
TIME: 16157



fernando@dcc.ufmg.br

Profilers

- Has anyone hear of `gprof`, what about `valgrind`?
  - What do these tools measure?
- What can we do with traditional profilers?
- Profilers are mostly dynamic tools. However, there are static profilers too. Has anyone heard of anything like this?
- What is interesting to profile on GPUs?

fernando@dcc.ufmg.br

Profiling for Divergences

Given a program, and its entry, how many times each branch has been visited, and how many divergences have happened per branch?

- How to implement a profiler that answers this question?

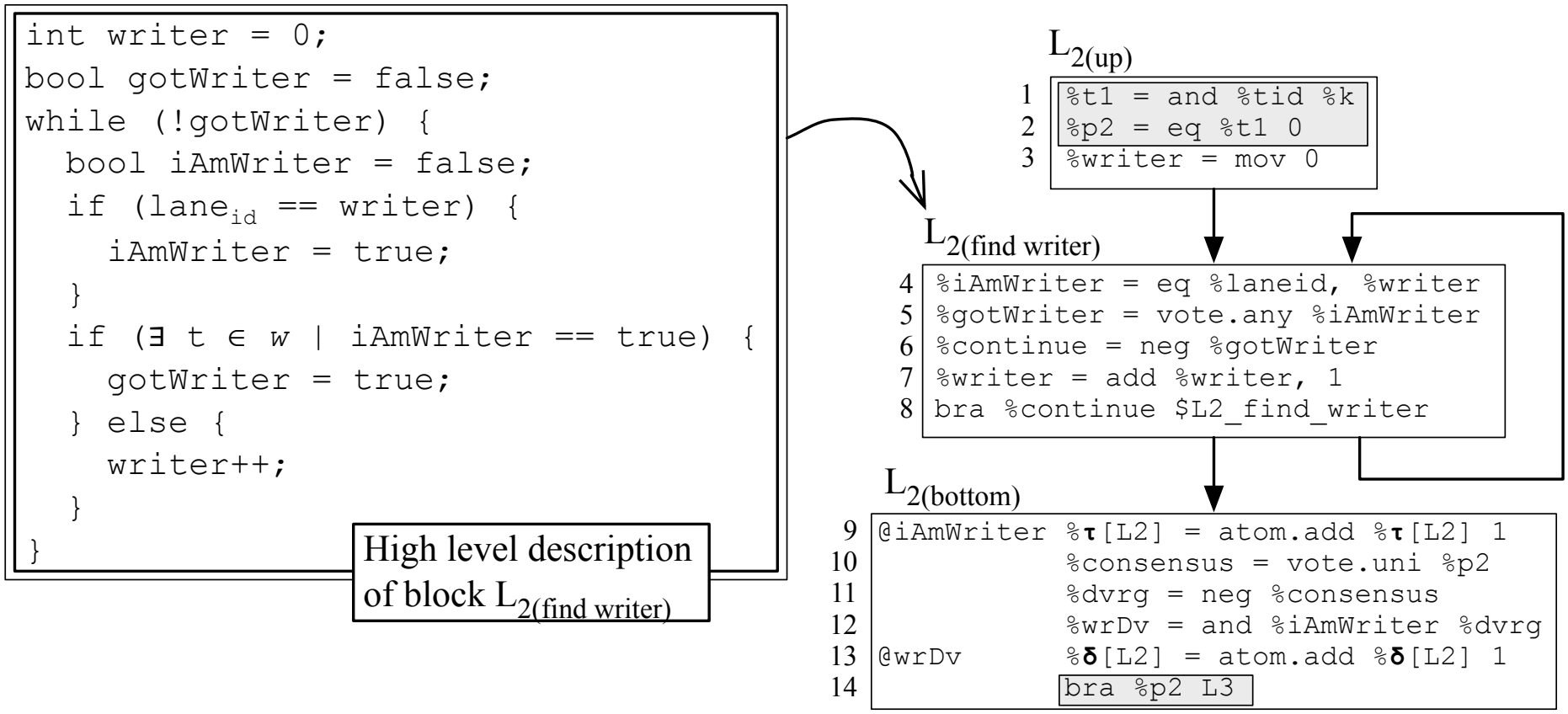
fernando@dcc.ufmg.br

A Simple Solution

- We can measure divergences with a *profiler* that works via instrumentation
- At each divergent path, we do a referendum among all the warp threads.
  - If they all vote together, then there is no divergence.
  - If they don't, then there is divergence.
- But we must find a writer…

Why is it difficult to find a writer?

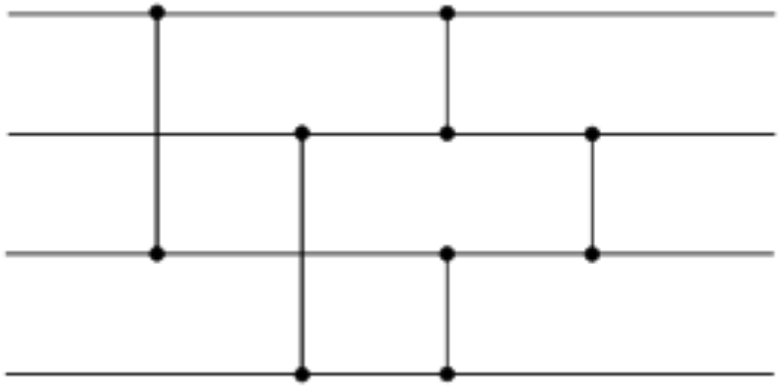
- What is the asymptotic complexity of profiling via instrumentation?

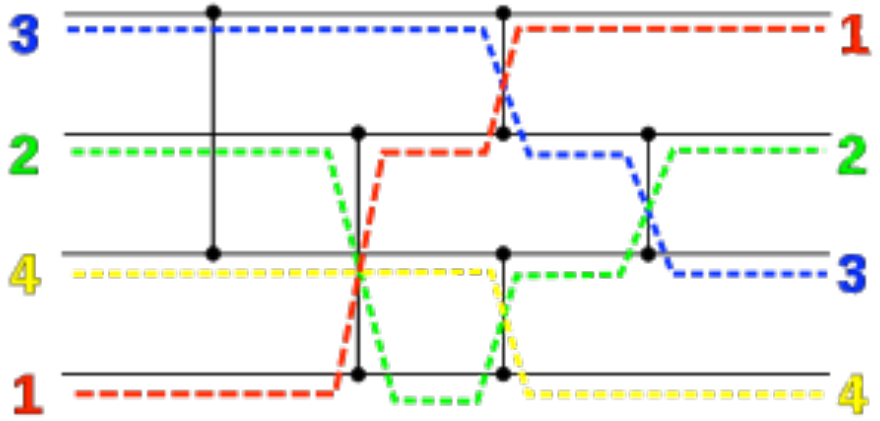


fernando@dcc.ufmg.br

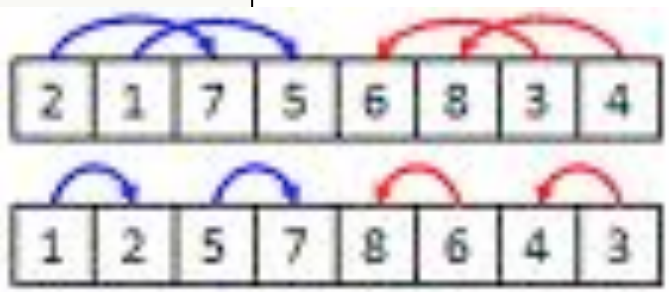
Example: Bitonic Sort

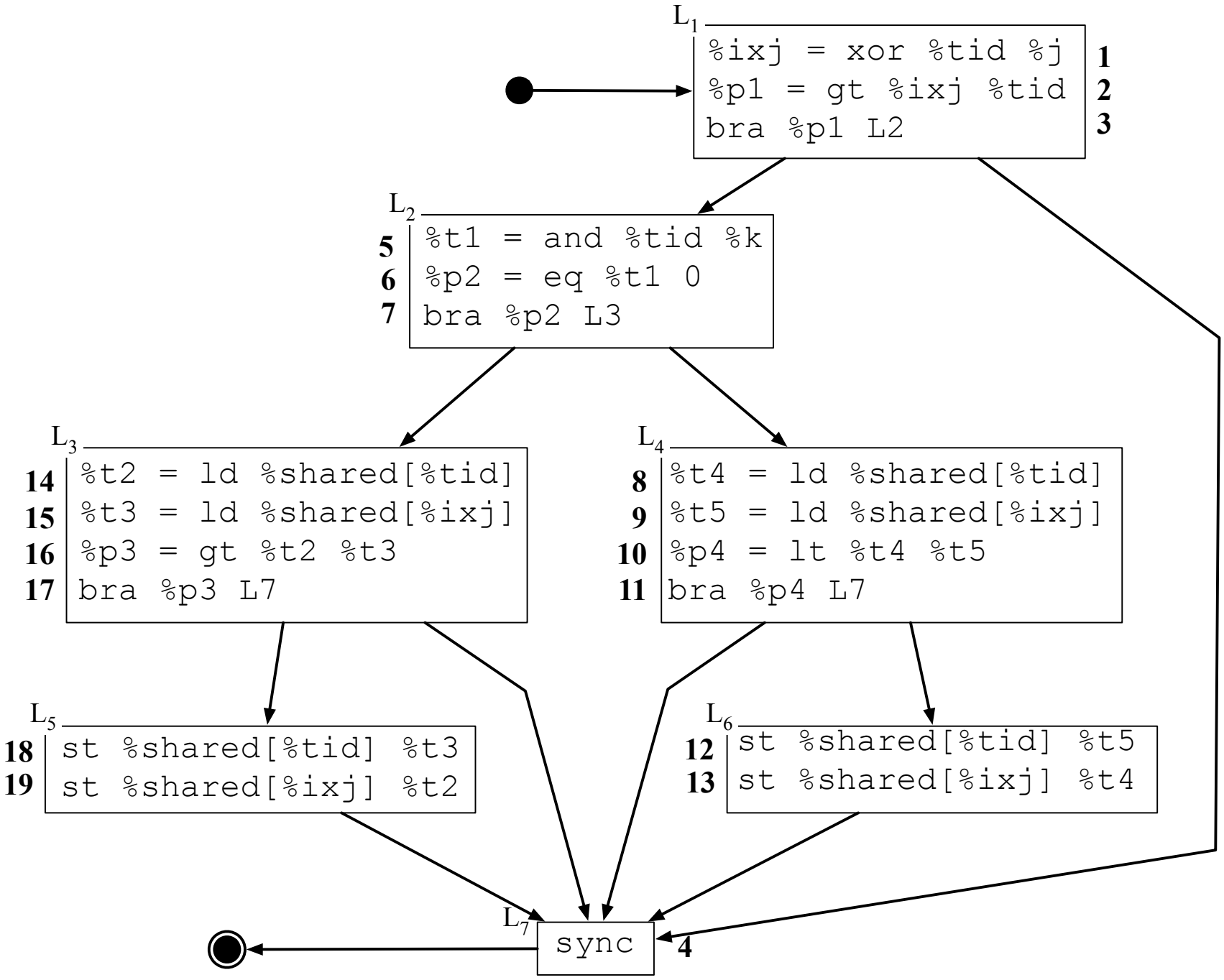
- How to sort efficiently in parallel?
- Has anyone heard about sorting networks?
  - Shell Sort (with bubble-sort)?
  - Bitonic Sort?





```
__global__ static void bitonicSort(int * values) {
  extern __shared__ int shared[];
  const unsigned int tid = threadIdx.x;
  shared[tid] = values[tid];
  __syncthreads();
  for (unsigned int k = 2; k <= NUM; k *= 2) {
    for (unsigned int j = k / 2; j>0; j /= 2) {
      unsigned int ixj = tid ^ j;
      if (ixj > tid) {
        if ((tid & k) == 0) {
          if (shared[tid] > shared[ixj]) {
            swap(shared[tid], shared[ixj]);
          }
        } else {
          if (shared[tid] < shared[ixj]) {
            swap(shared[tid], shared[ixj]);
          }
        }
      }
      __syncthreads();
    }
  }
  values[tid] = shared[tid];
}
```




L1
%ixj = xor %tid %j
%p1 = gt %ixj %tid
bra %p1 L2
1
2
3
L2
%t1 = and %tid %k
%p2 = eq %t1 0
bra %p2 L3
5
6
7
L3
%t2 = ld %shared[%tid]
%t3 = ld %shared[%ixj]
%p3 = gt %t2 %t3
bra %p3 L7
14
15
16
17
L4
%t4 = ld %shared[%tid]
%t5 = ld %shared[%ixj]
%p4 = lt %t4 %t5
bra %p4 L7
8
9
10
11
L5
st %shared[%tid] %t3
st %shared[%ixj] %t2
18
19
L6
st %shared[%tid] %t5
st %shared[%ixj] %t4
12
13
L7
sync
4

fernando@dcc.ufmg.br

The Control Flow Graph

- What are the boxes?
- What are the boldface numbers in front of the boxes?
- What are the edges?
- Can you make a correlation between source code and CFG?
- Can you think about a systematic way to produce CFGs from source code?
- **Go back to the CFG and answer me: what would be the most serious divergences?**

fernando@dcc.ufmg.br

Warp Trace

cycle	label	opcode	def	use_1	use_2	t_0	t_1	t_2	t_3
1	L_1	xor	%ixj	%tid	%j	ι_1	ι_1	ι_1	ι_1
2		gt	%p1	%ixj	%tid	ι_2	ι_2	ι_2	ι_2
3		bra		%p1	L_2	ι_3	ι_3	ι_3	ι_3
4	L_2	and	%t1	%tid	%k	ι_5	•	ι_5	•
5		eq	%p2	%t1	0	ι_6	•	ι_6	•
6		bra		%p2	L_3	ι_7	•	ι_7	•
7	L_3	load	%t2	%shared	%tid	ι_{14}	•	•	•
8		load	%t3	%shared	%ixj	ι_{15}	•	•	•
9		gt	%p3	%t2	%t3	ι_{16}	•	•	•
10		bra		%p3	L_7	ι_{17}	•	•	•
11	L_4	load	%t4	%shared	%tid	•	•	ι_8	•
12		load	%t5	%shared	%ixj	•	•	ι_9	•
13		lt	%p4	%t4	%t5	•	•	ι_{10}	•
14		bra		%p3	L_7	•	•	ι_{11}	•
15	L_5	store		%tid	%t3	ι_{18}	•	•	•
16		store		%tid	%t2	ι_{19}	•	•	•
17	L_6	store		%tid	%t5	•	•	ι_{12}	•
18		store		%tid	%t4	•	•	ι_{13}	•
19	L_7	sync				ι_4	ι_4	ι_4	ι_4

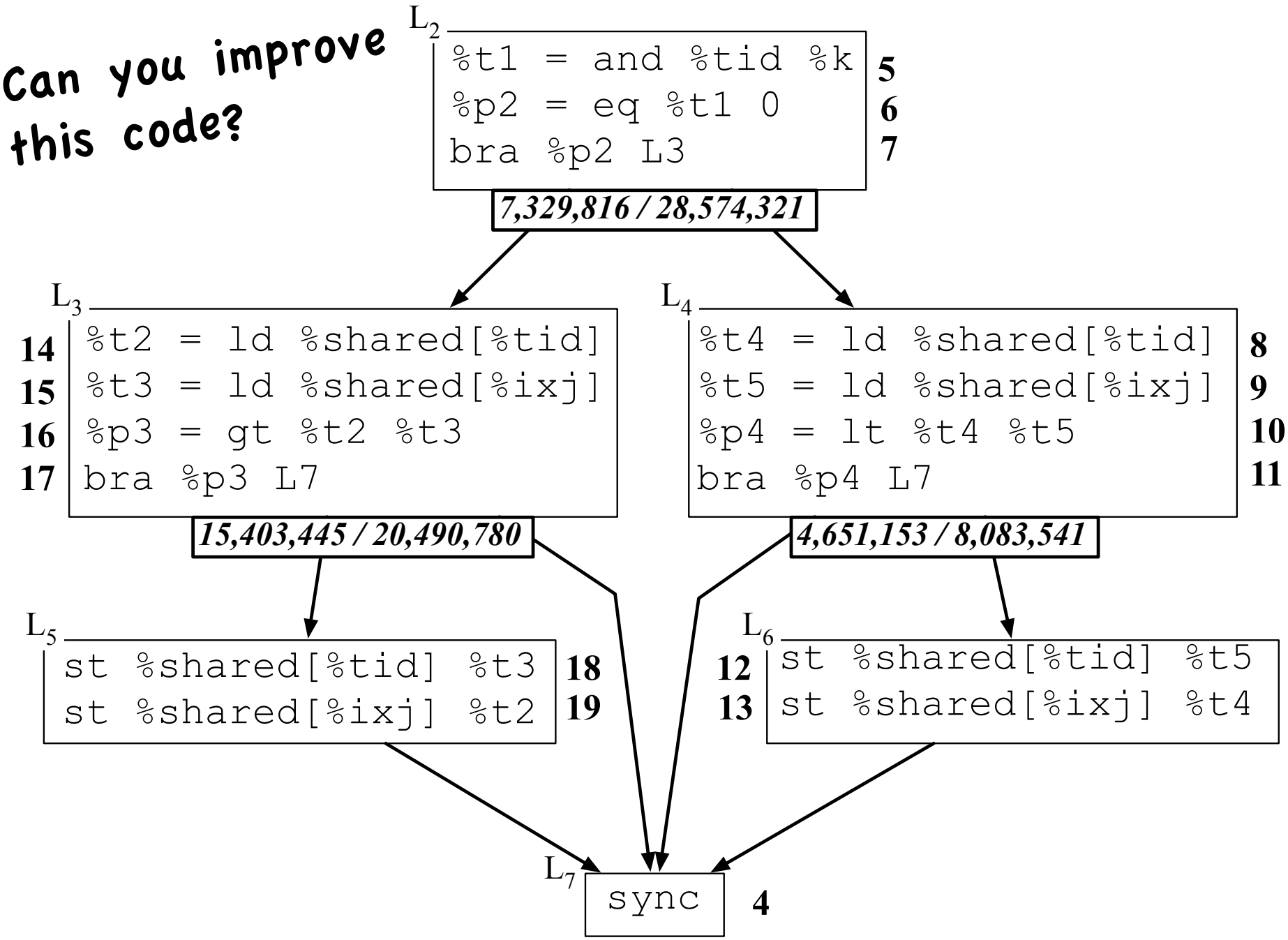
```
__global__ static void bitonicSort(int * values) {
  extern __shared__ int shared[];
  const unsigned int tid = threadIdx.x;
  shared[tid] = values[tid];
  __syncthreads();
  for (unsigned int k = 2; k <= NUM; k *= 2) {
    for (unsigned int j = k / 2; j>0; j /= 2) {
      unsigned int ixj = tid ^ j;
      if (ixj > tid) {
        if ((tid & k) == 0) {
          if (shared[tid] > shared[ixj]) {
            swap(shared[tid], shared[ixj]);
          }
        } else {
          if (shared[tid] < shared[ixj]) {
            swap(shared[tid], shared[ixj]);
          }
        }
      }
      __syncthreads();
    }

    values[tid] = shared[tid];
}
```

7,329,816 / 28,574,321

15,403,445 / 20,490,780

4,651,153 / 8,083,541

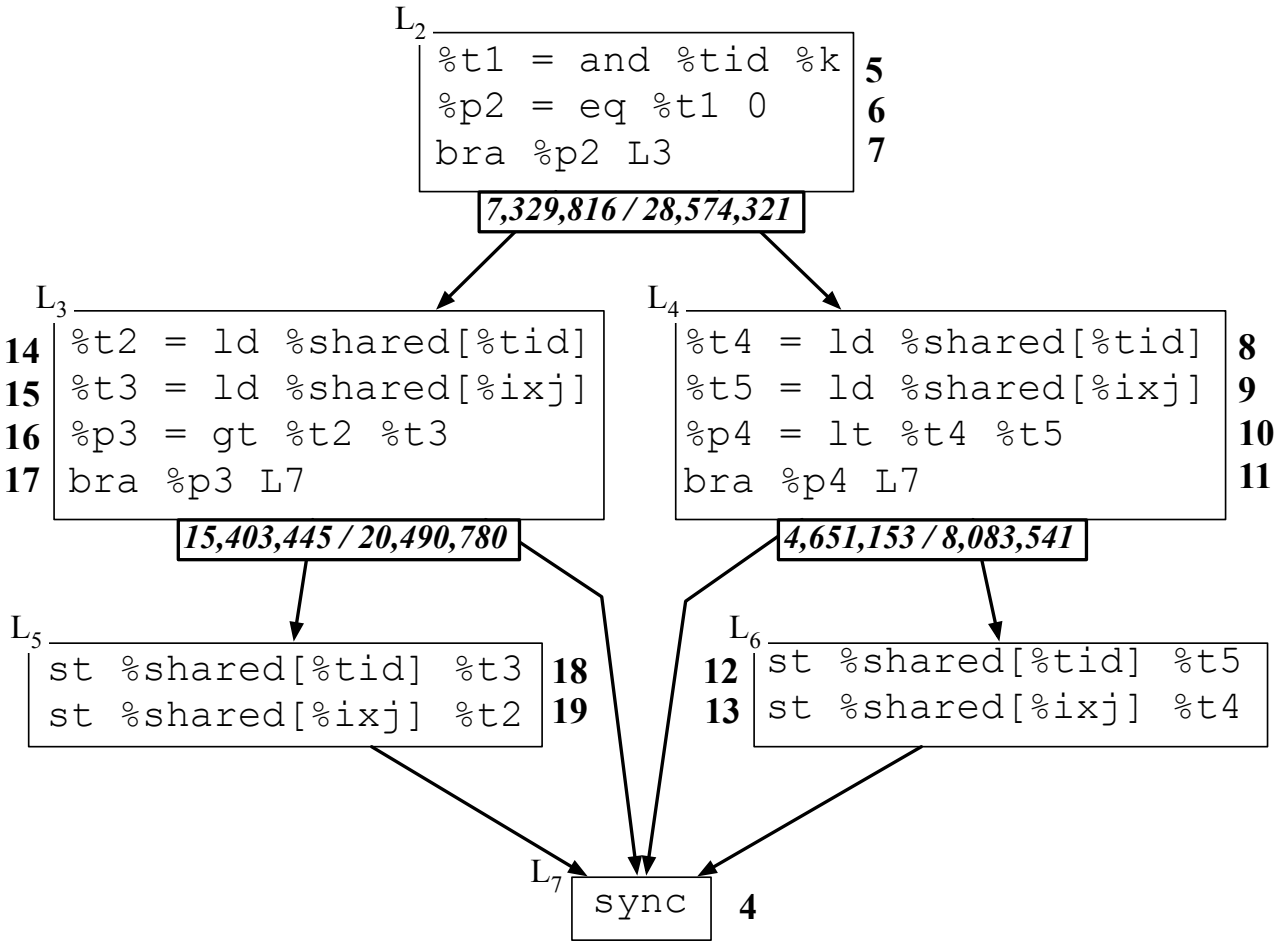
Can you improve this code?



fernando@dcc.ufmg.br

```
if ((tid & k) == 0) {
  if (shared[tid] > shared[ixj]) {
    swap(shared[tid], shared[ixj]);
  }
} else {
  if (shared[tid] < shared[ixj]) {
    swap(shared[tid], shared[ixj]);
  }
}
```

- How can we improve this program to mitigate the problem of divergences?



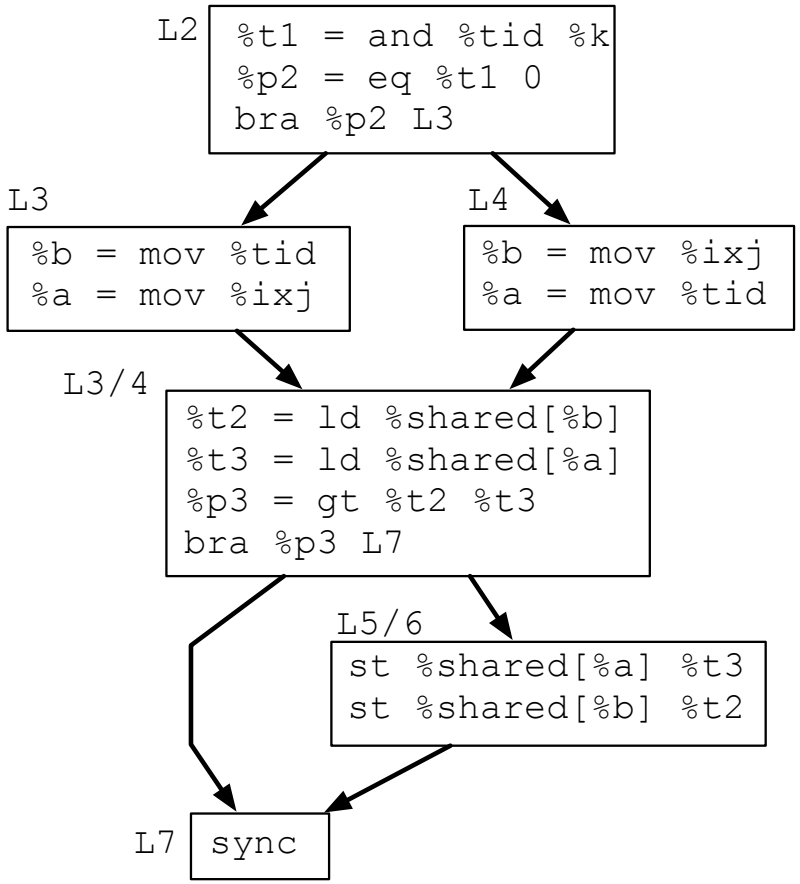
fernando@dcc.ufmg.br

First Optimization: 6.7% speed up*

(a)

```
unsigned int a, b;
if ((tid & k) == 0){
  b = tid;
  a = ixj;
} else {
  b = ixj;
  a = tid;
}
if (sh[b] > sh[a]){
  swap(sh[b],sh[a]);
}
```

(b)



*** Numbers refer to GTX 260**

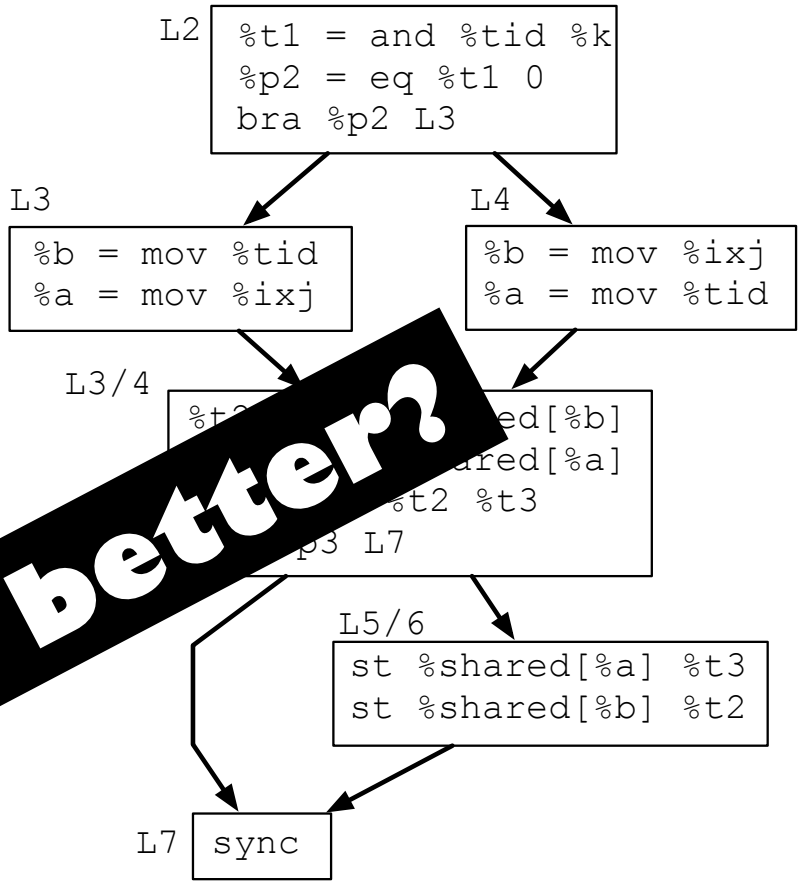
fernando@dcc.ufmg.br

First Optimization: 6.7% speed up*

(a)

```
unsigned int a, b;
if ((tid & k) == 0){
  b = tid;
  a = ixj;
} else {
  b = ixj;
  a = tid;
}
if (sh[b] > sh[a]){
  swap(sh[b],sh[a]);
}
```

(b)



Can you do better?

* Numbers

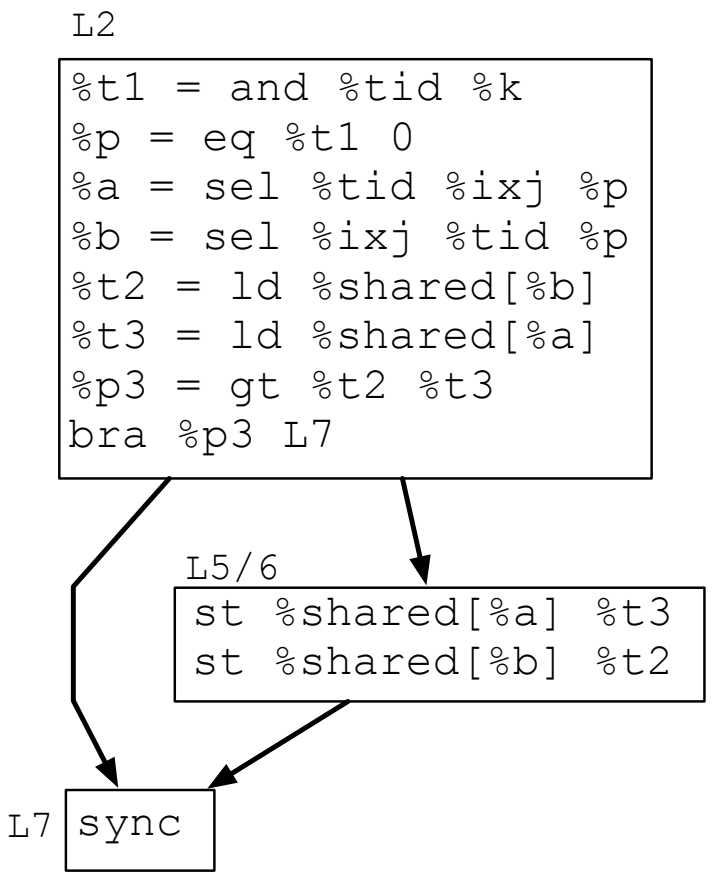
fernando@dcc.ufmg.br

Second Optimization: 9.2% speed up

(a)

```
int p = (tid & k) == 0;
unsigned b = p?tid:ixj;
unsigned a = p?ixj:tid;

if (sh[b] > sh[a]) {
  swap(sh[b], sh[a]);
}
```

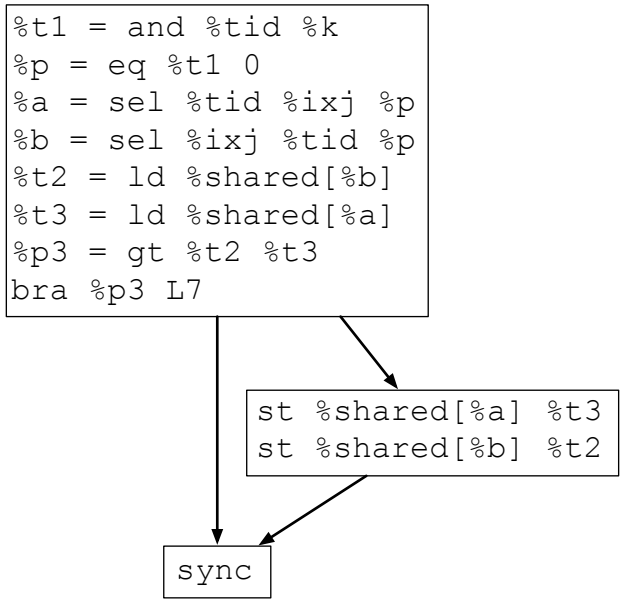
(b)

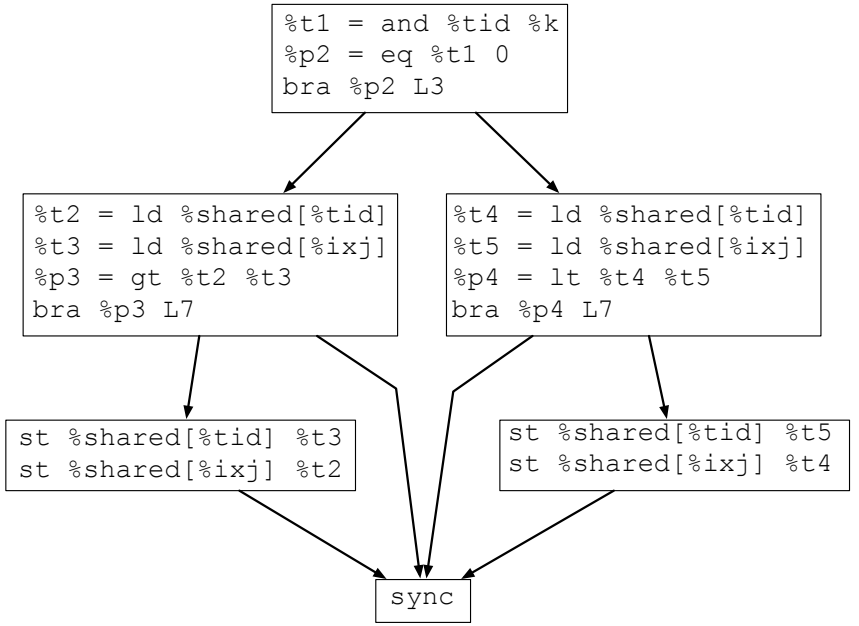


fernando@dcc.ufmg.br

The final result

```
if ((tid & k) == 0) {
  if (shared[tid] > shared[ixj]) {
    swap(shared[tid], shared[ixj]);
  }
} else {
  if (shared[tid] < shared[ixj]) {
    swap(shared[tid], shared[ixj]);
  }
}
```






```
int p = (tid & k) == 0;
unsigned b = p ? tid : ixj;
unsigned a = p ? ixj : tid;

if (shared[b] > shared [a]) {
  swap(shared[b], shared[a]);
}
```

fernando@dcc.ufmg.br

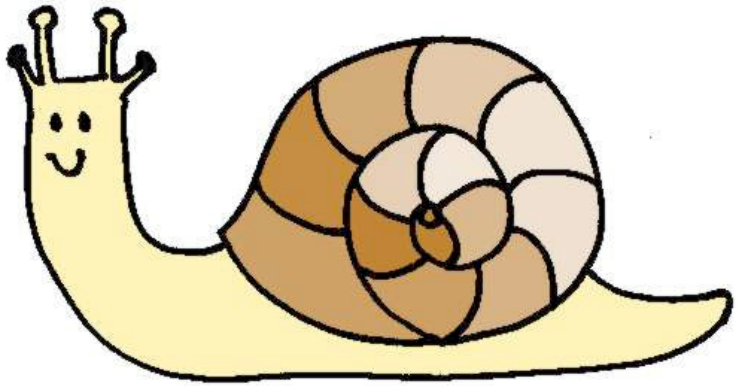
Divergence Analysis

Given a program, which branches might cause divergences, and which branches will never do it?

- This analysis helps us to find which branches the compiler must optimize.

fernando@dcc.ufmg.br

Why not to use profiling?



- Input dependent.
- Relatively hard to use.
- Huge slow-down: 140 times!
- The analysis can be easily embedded into the compiler.
- The analysis is definitive: if it says a branch is not divergent, then this branch will **never** diverge.

fernando@dcc.ufmg.br

Our Solution: Divergence Analysis

A local variable is divergent if different threads see it with different values.

- Which variables are divergent?

fernando@dcc.ufmg.br

Our Solution: Divergence Analysis

A local variable is divergent if different threads see it with different values.

- Which variables are divergent?
  - *v* = **`tid`**
  - **`atomic`** { v = f(...) }
  - *v* is **data dependent** on divergent variable *u*.
  - *v* is **control dependent** on divergent variable *u*.

The thread id is always divergent

```
__global__
void saxpy (int n, float alpha, float *x, float *y) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < n) y[i] = alpha * x[i] + y[i];
}
```

Each thread sees a different thread id, so…

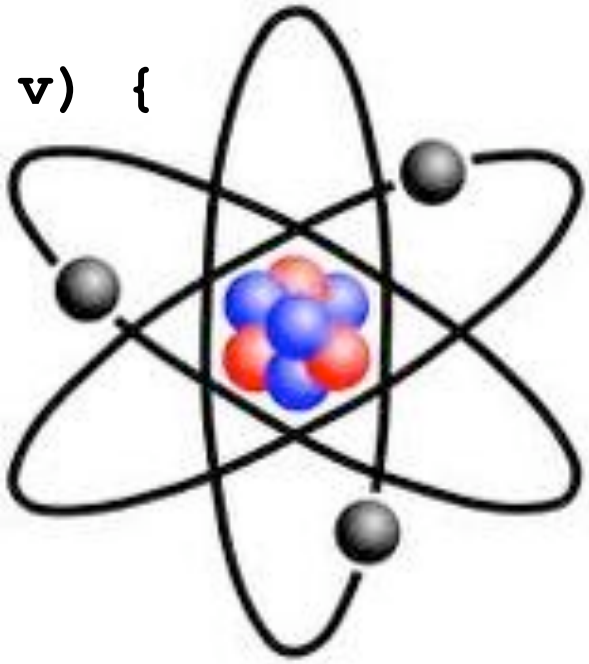
- But threads in different blocks may have the same **`threadIdx.x`**. Is this a problem?

fernando@dcc.ufmg.br

Variables defined by atomic ops

- What is the program below doing?
  - The macro ATOMINC increments a global memory position, and returns the value of the result.

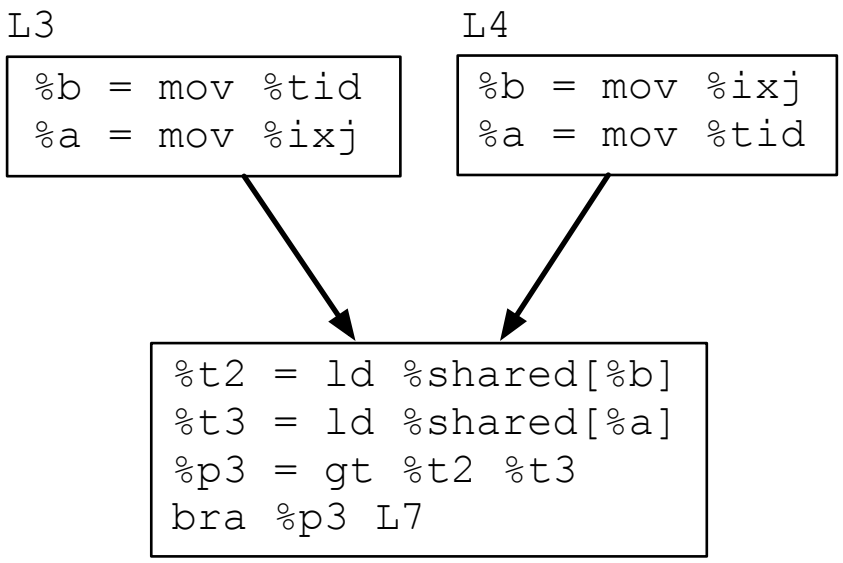
```
__global__
void ex_atomic (int index, float* v) {
  int i = 0;
  i = ATOMINC( v[index] );
}
```



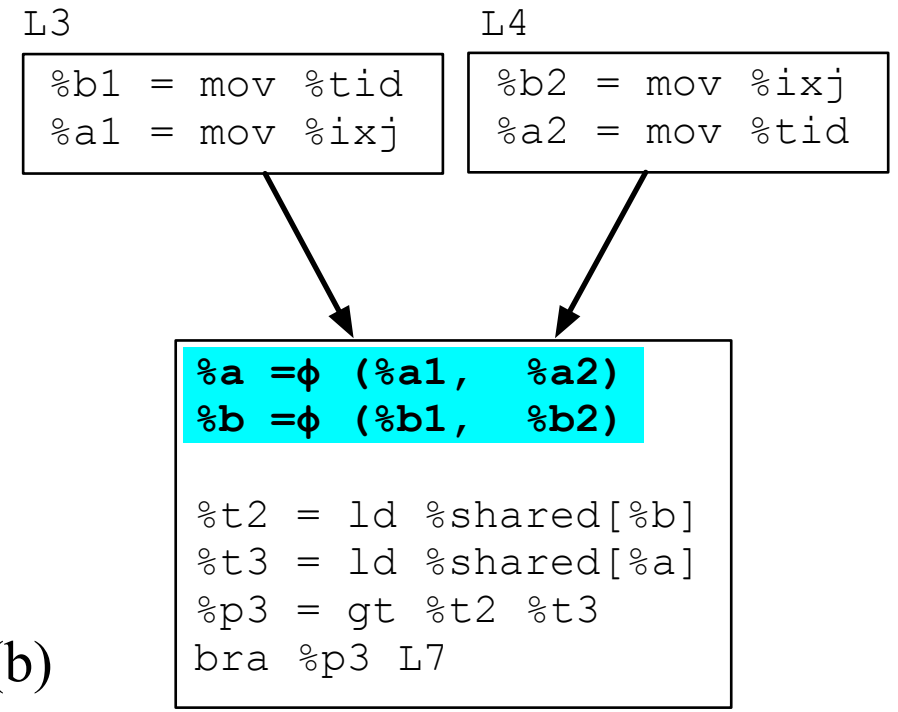
fernando@dcc.ufmg.br

Static Single Assignment Form

- An intermediate representation in which each variable is defined only once.
  - We need to understand phi-functions.



(a)

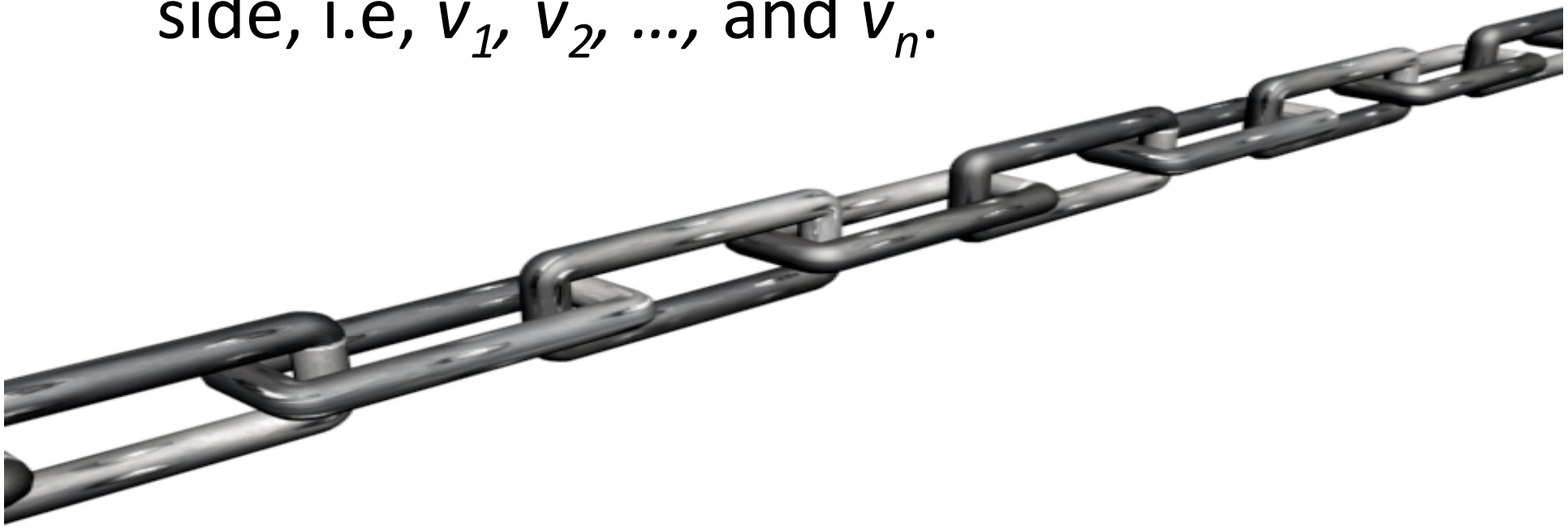


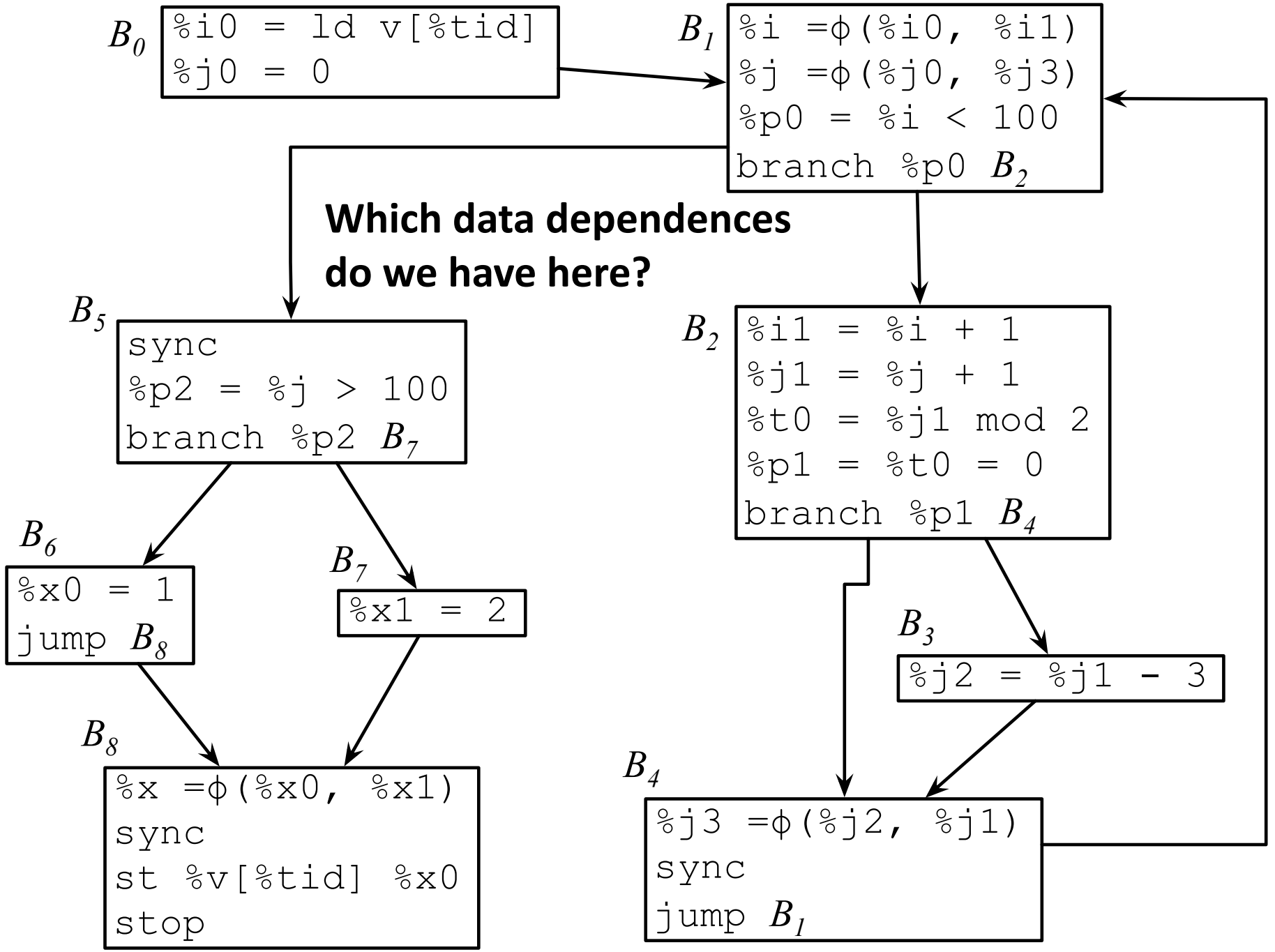
(b)

fernando@dcc.ufmg.br

What is Data Dependence?

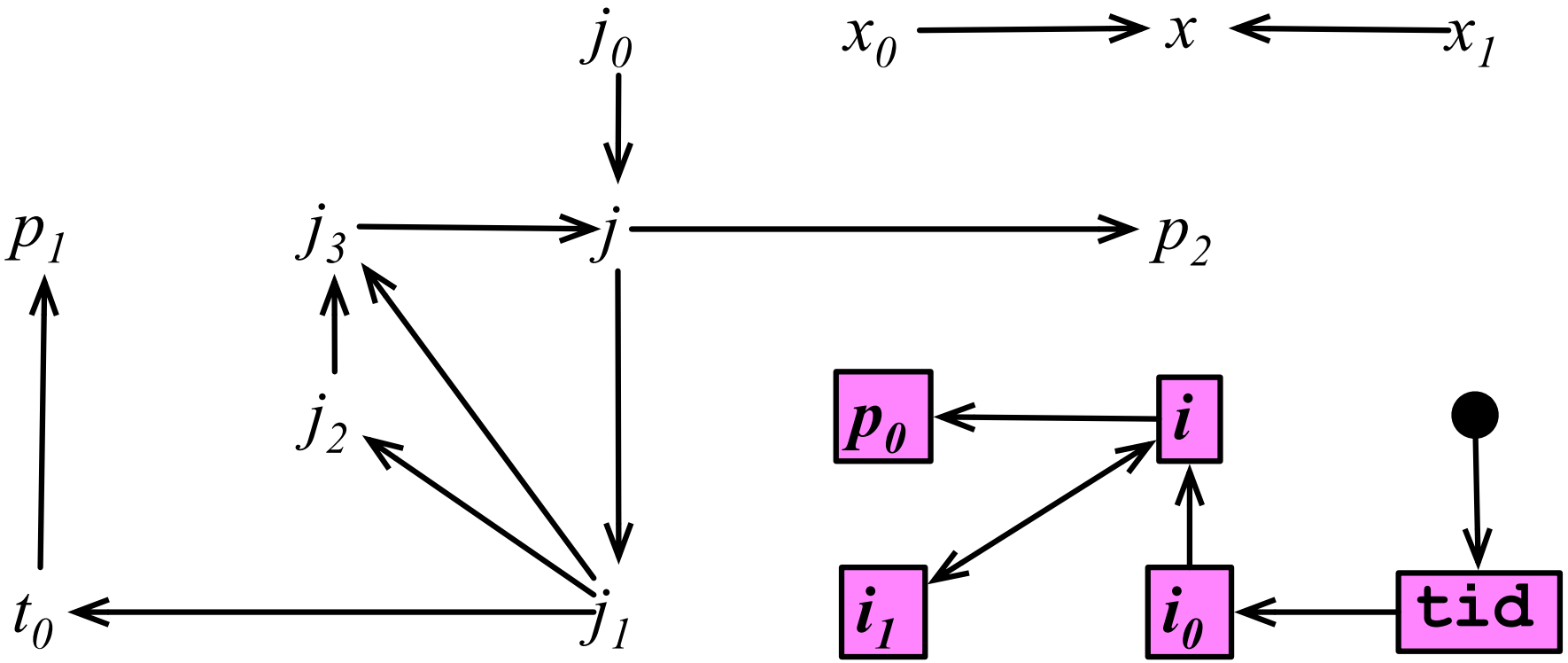
- If the program contains an assignment such as $v = f(v_1, v_2, \ldots, v_n)$, then v is data dependent on every variable in the right side, i.e, v_1, v_2, …, and v_n.




B_0
%i0 = ld v[%tid]
%j0 = 0
B_1
%i =ϕ(%i0, %i1)
%j =ϕ(%j0, %j3)
%p0 = %i < 100
branch %p0 B_2
Which data dependences do we have here?
B_5
sync
%p2 = %j > 100
branch %p2 B_7
B_2
%i1 = %i + 1
%j1 = %j + 1
%t0 = %j1 mod 2
%p1 = %t0 = 0
branch %p1 B_4
B_6
%x0 = 1
jump B_8
B_7
%x1 = 2
B_3
%j2 = %j1 - 3
B_8
%x =ϕ(%x0, %x1)
sync
st %v[%tid] %x0
stop
B_4
%j3 =ϕ(%j2, %j1)
sync
jump B_1

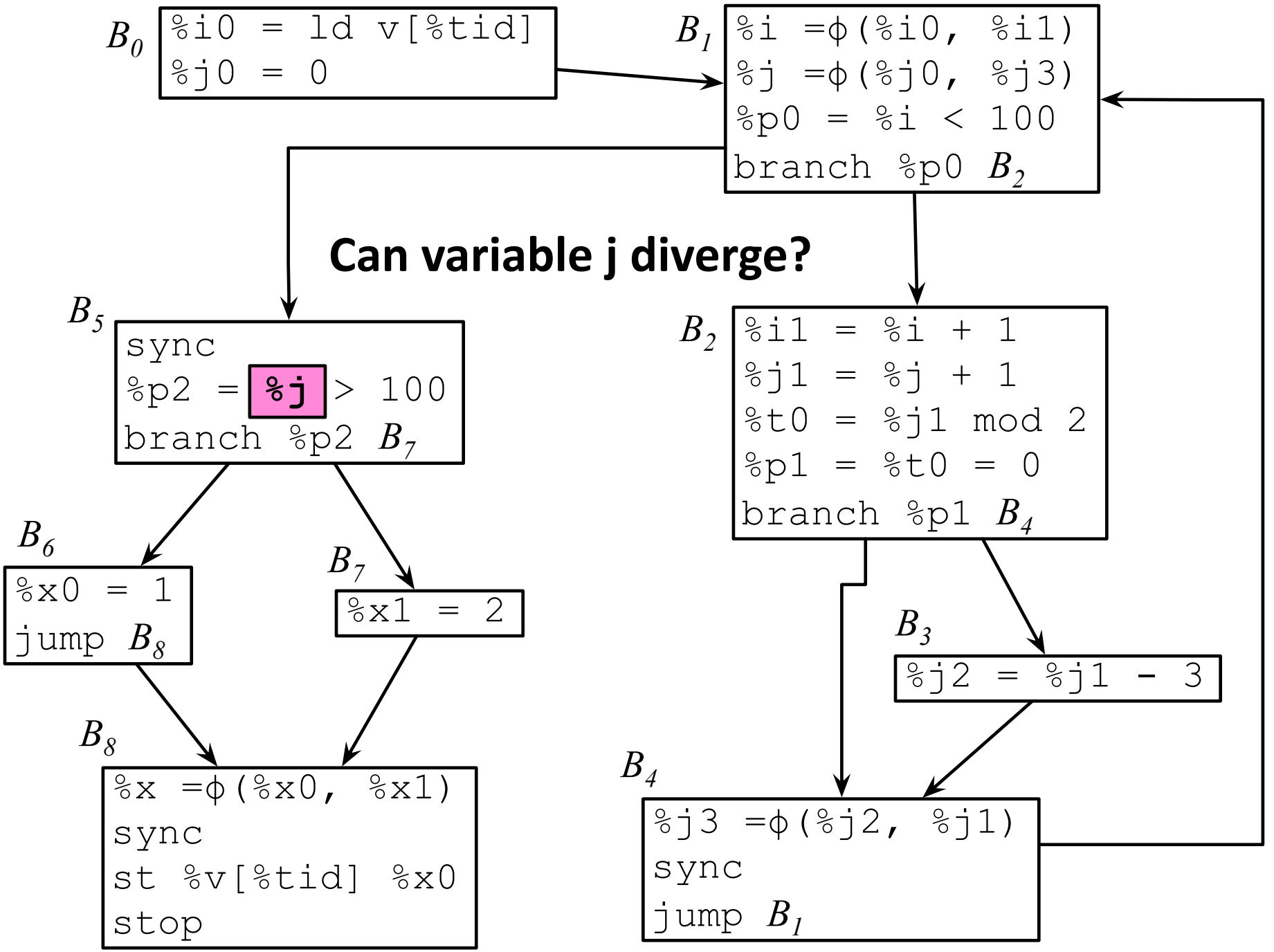
fernando@dcc.ufmg.br

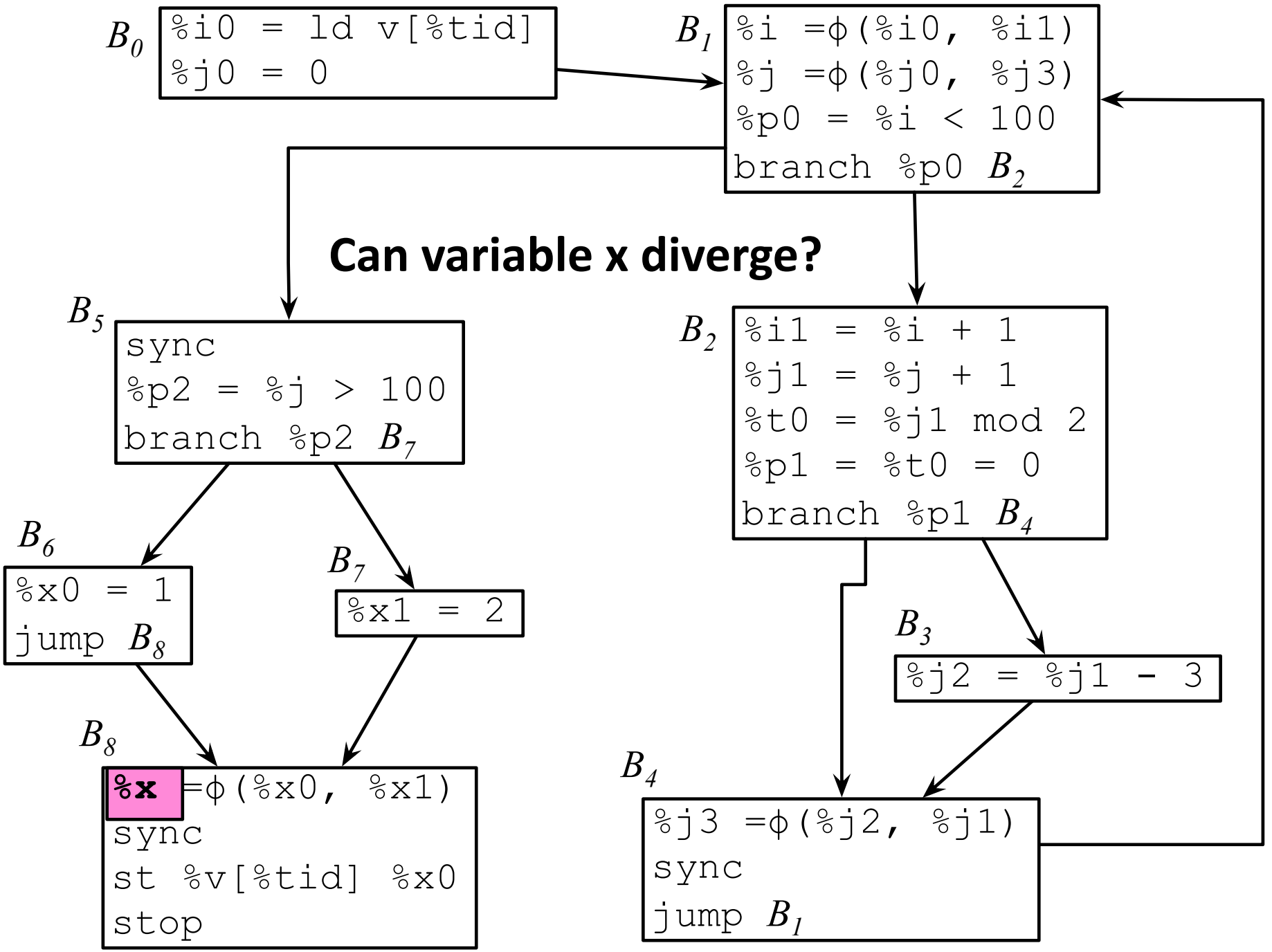
Divergences due to data dependences

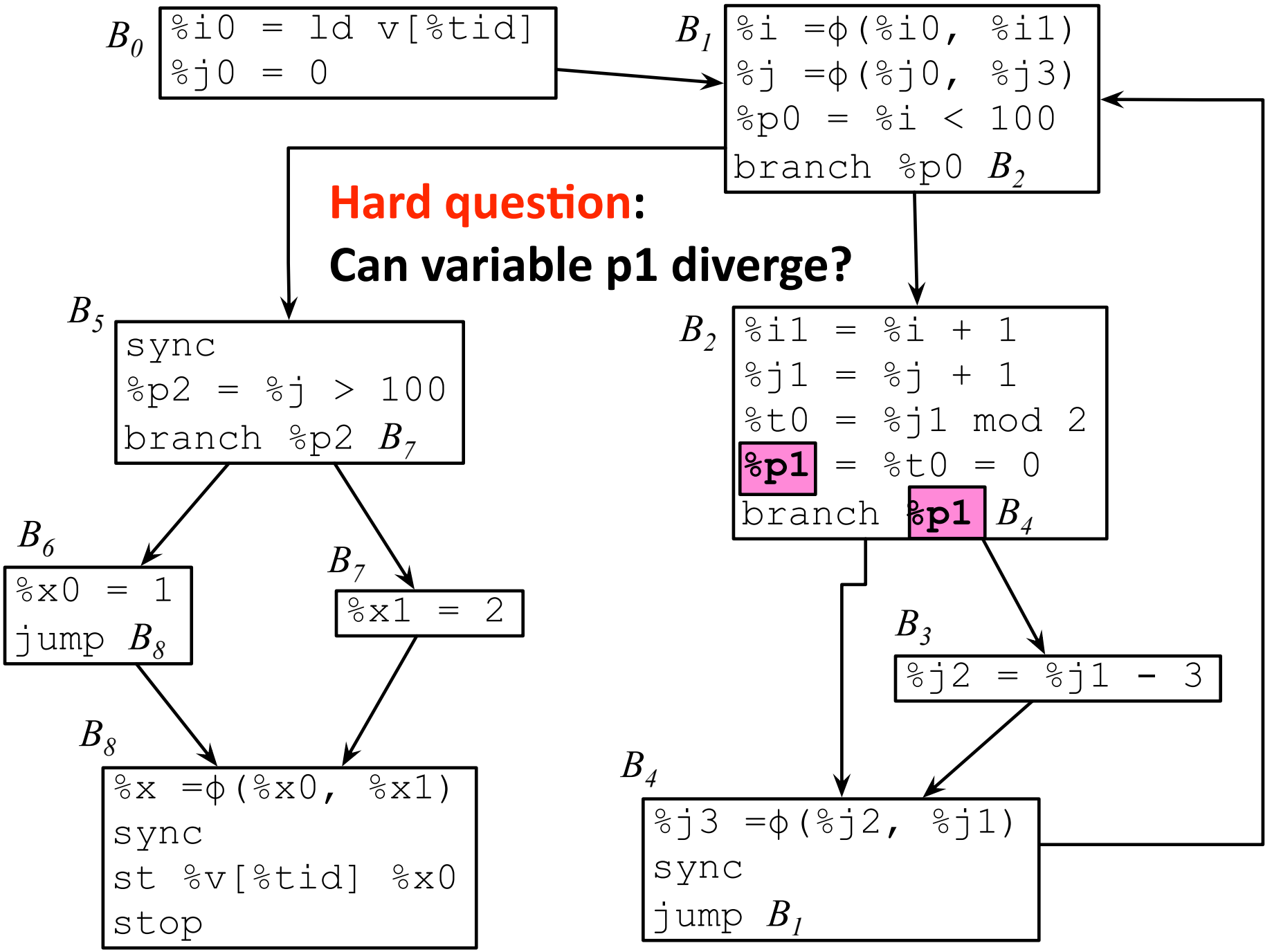


Why is SSA so good here?

Are we missing anything?


B_0
%i0 = ld v[%tid]
%j0 = 0
B_1
%i =ϕ(%i0, %i1)
%j =ϕ(%j0, %j3)
%p0 = %i < 100
branch %p0 B_2
Can variable j diverge?
B_5
sync
%p2 = %j > 100
branch %p2 B_7
B_2
%i1 = %i + 1
%j1 = %j + 1
%t0 = %j1 mod 2
%p1 = %t0 = 0
branch %p1 B_4
B_6
%x0 = 1
jump B_8
B_7
%x1 = 2
B_3
%j2 = %j1 - 3
B_8
%x =ϕ(%x0, %x1)
sync
st %v[%tid] %x0
stop
B_4
%j3 =ϕ(%j2, %j1)
sync
jump B_1

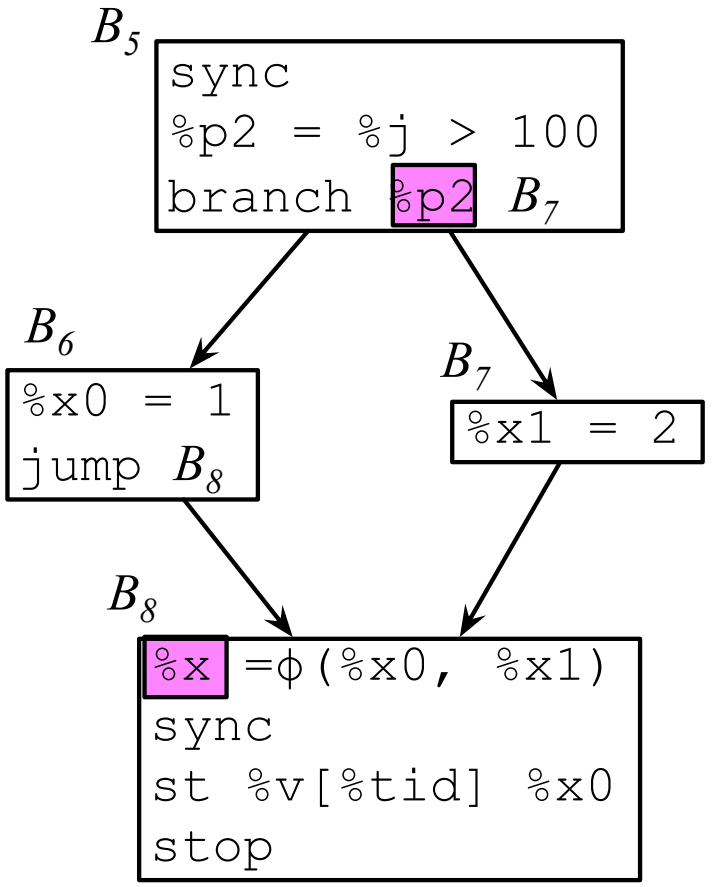

B_0
%i0 = ld v[%tid]
%j0 = 0
B_1
%i =ϕ(%i0, %i1)
%j =ϕ(%j0, %j3)
%p0 = %i < 100
branch %p0 B_2
Can variable x diverge?
B_5
sync
%p2 = %j > 100
branch %p2 B_7
B_2
%i1 = %i + 1
%j1 = %j + 1
%t0 = %j1 mod 2
%p1 = %t0 = 0
branch %p1 B_4
B_6
%x0 = 1
jump B_8
B_7
%x1 = 2
B_3
%j2 = %j1 - 3
B_8
%x =ϕ(%x0, %x1)
sync
st %v[%tid] %x0
stop
B_4
%j3 =ϕ(%j2, %j1)
sync
jump B_1


B_0
%i0 = ld v[%tid]
%j0 = 0
B_1
%i =ϕ(%i0, %i1)
%j =ϕ(%j0, %j3)
%p0 = %i < 100
branch %p0 B_2
Hard question:
Can variable p1 diverge?
B_5
sync
%p2 = %j > 100
branch %p2 B_7
B_2
%i1 = %i + 1
%j1 = %j + 1
%t0 = %j1 mod 2
%p1 = %t0 = 0
branch %p1 B_4
B_6
%x0 = 1
jump B_8
B_7
%x1 = 2
B_3
%j2 = %j1 - 3
B_8
%x =ϕ(%x0, %x1)
sync
st %v[%tid] %x0
stop
B_4
%j3 =ϕ(%j2, %j1)
sync
jump B_1

fernando@dcc.ufmg.br

Sync Dependences

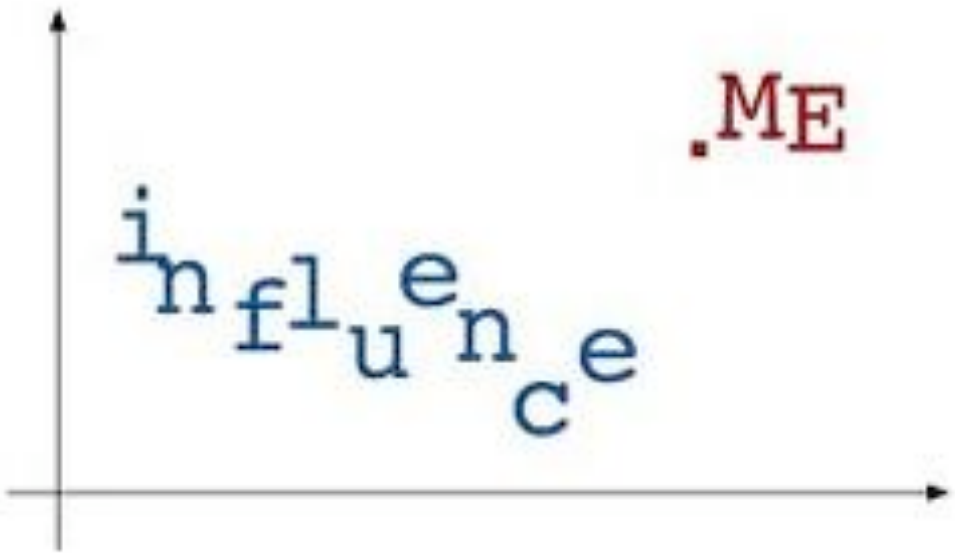
- A variable *v* is sync dependent on a predicate variable *p* if *v* may reach a synchronization point with a different value for different threads, depending on how threads branch on *p*.
- How to find the set of sync dependent variables?



fernando@dcc.ufmg.br

The influence region

- The influence region of a predicate is the set of basic blocks that may (or may not) be reached depending on the value of the predicate, up to the synchronization barrier.



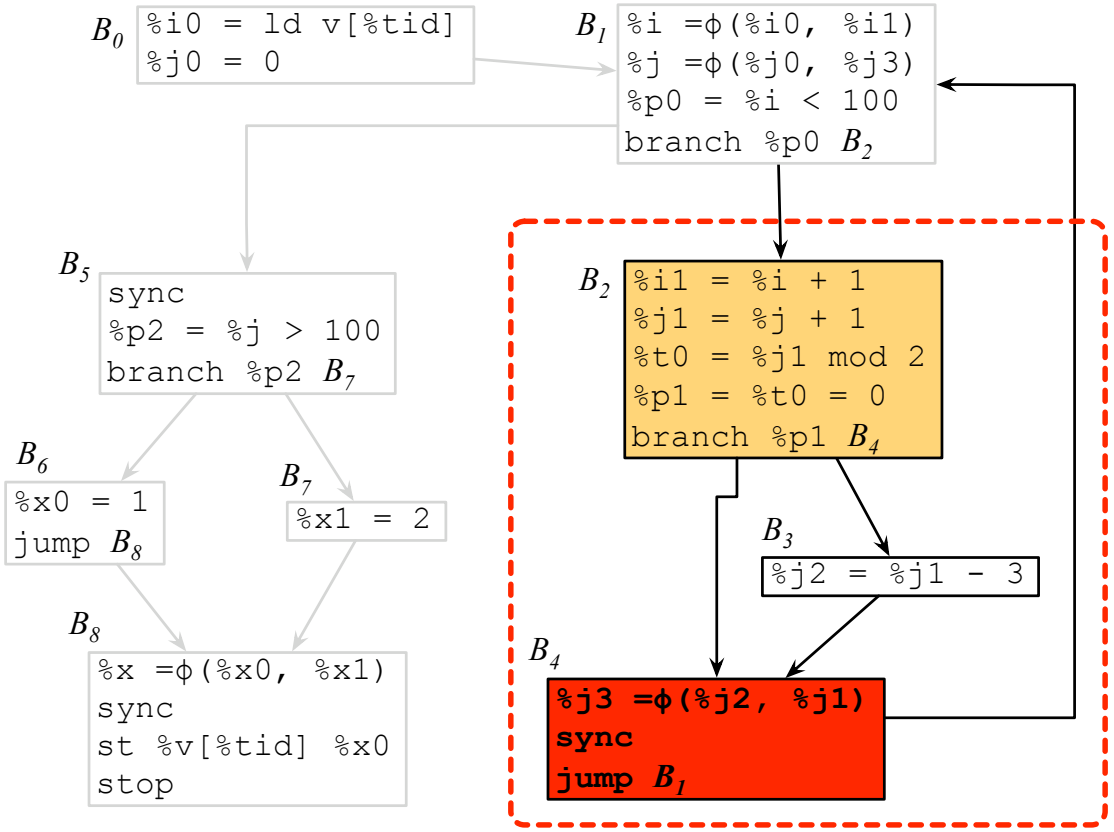
- The Synchronization barrier is placed at the *immediate* **post-dominator** of the branch.
- **Alert!** Head is spinning: what the heck is a post-dominator?

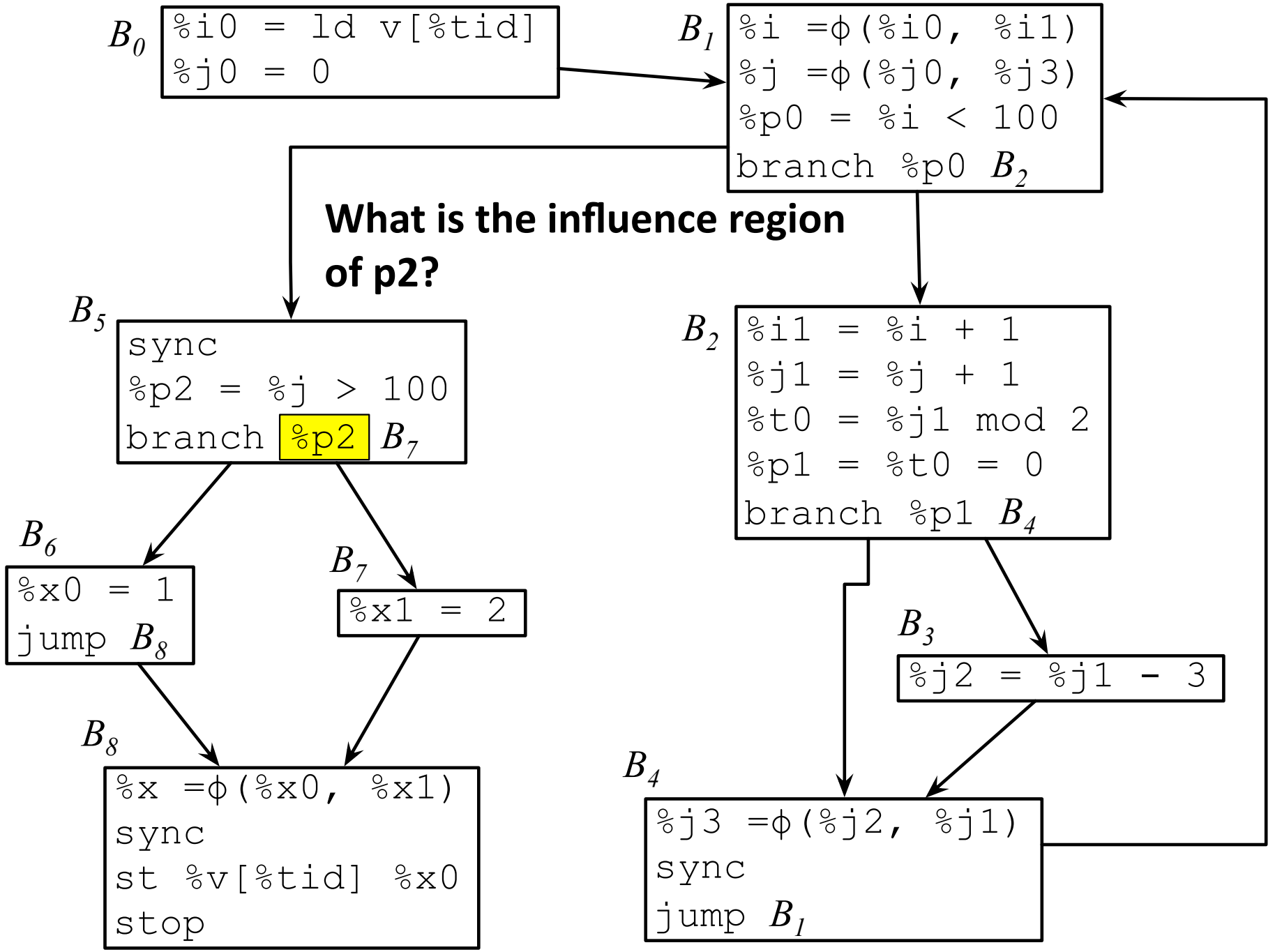
fernando@dcc.ufmg.br

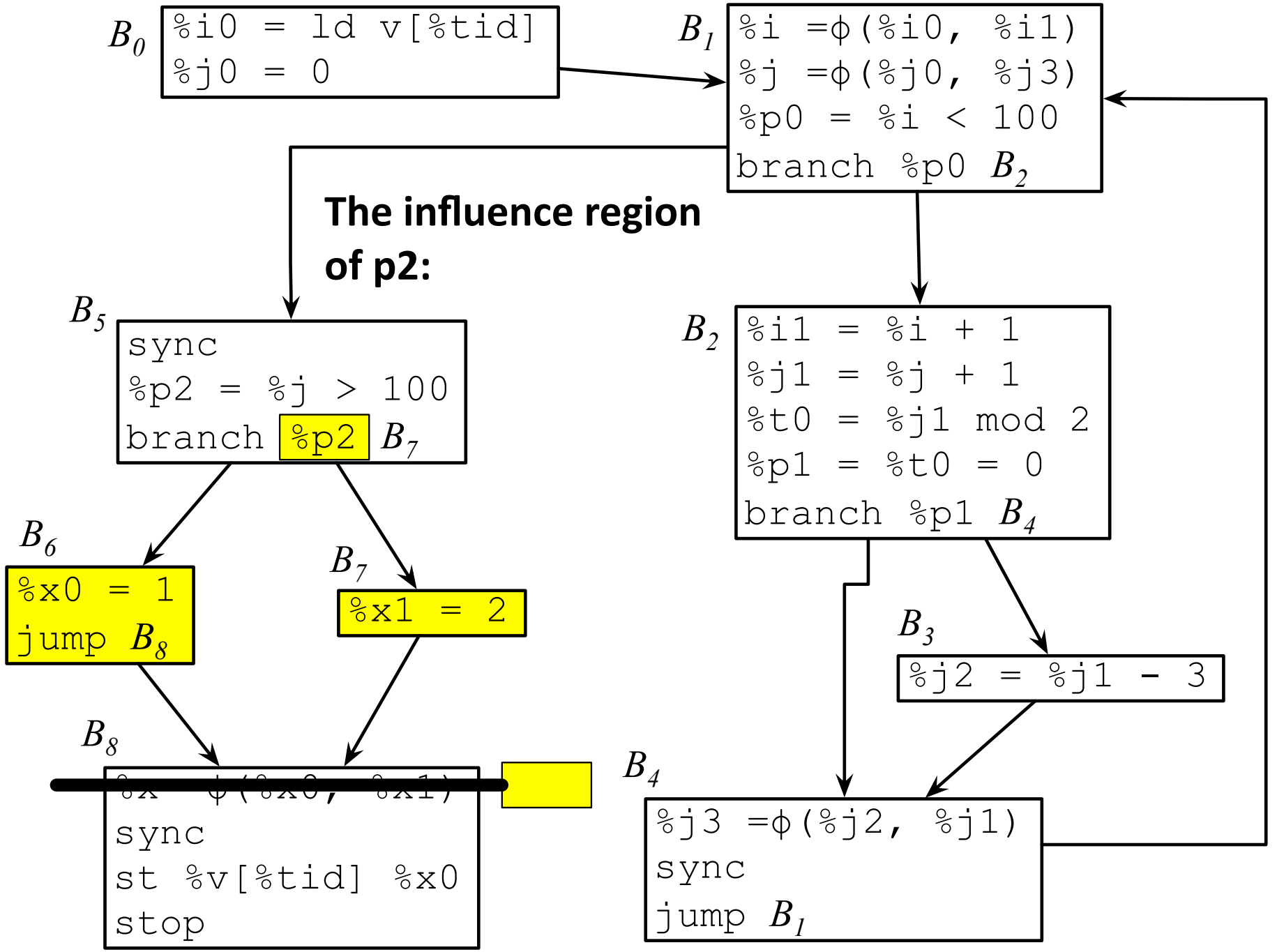
Post-Dominance

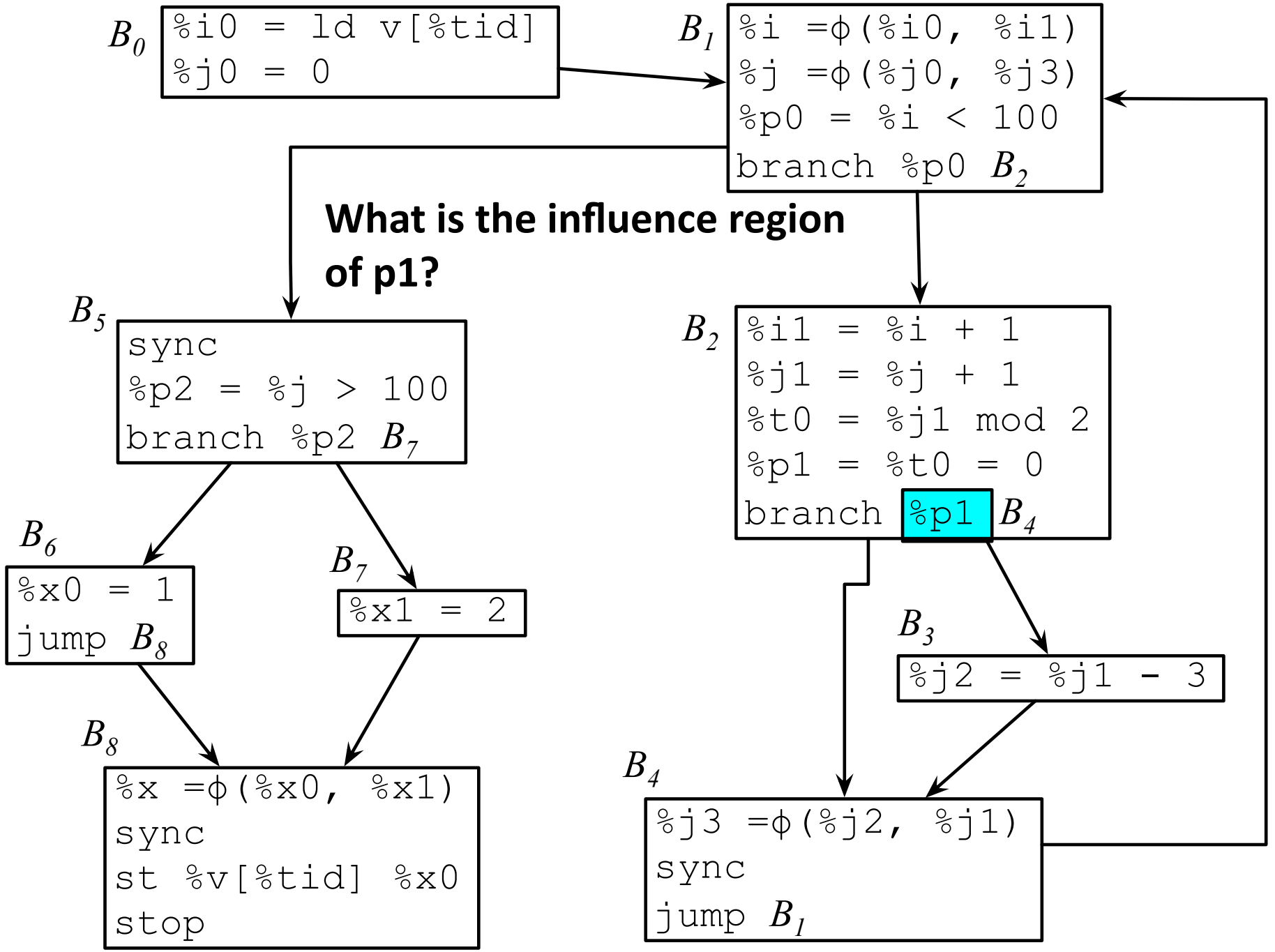
- A Basic block B_1 post-dominates another basic block B_2 if, and only if, every path from B_2 to the end of the program goes across B_1.

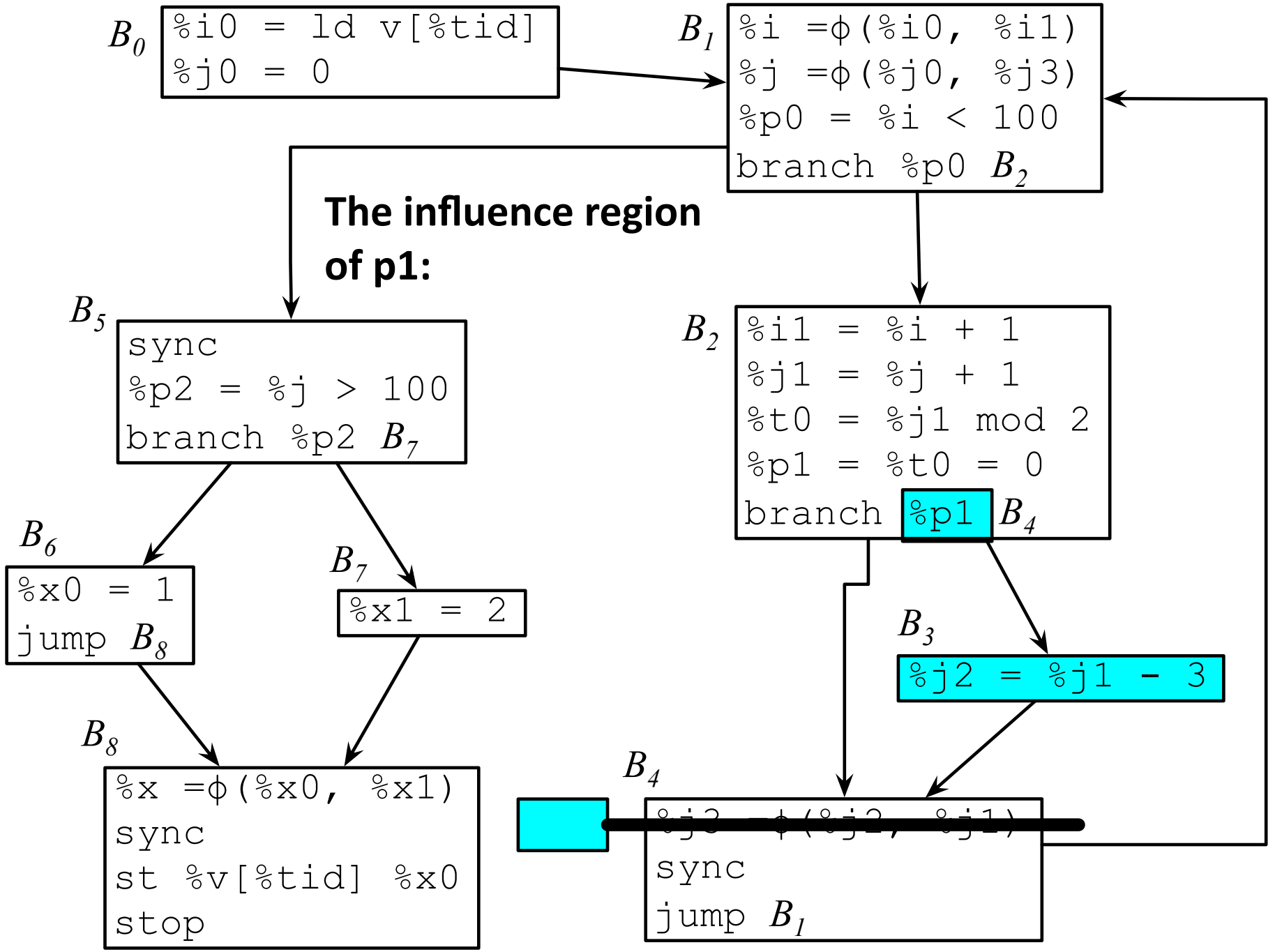
- For instance, in our running example, B_4 post-dominates B_2, thus, there is a sync barrier at B_4 to synchronize the threads that may diverge at B_2.

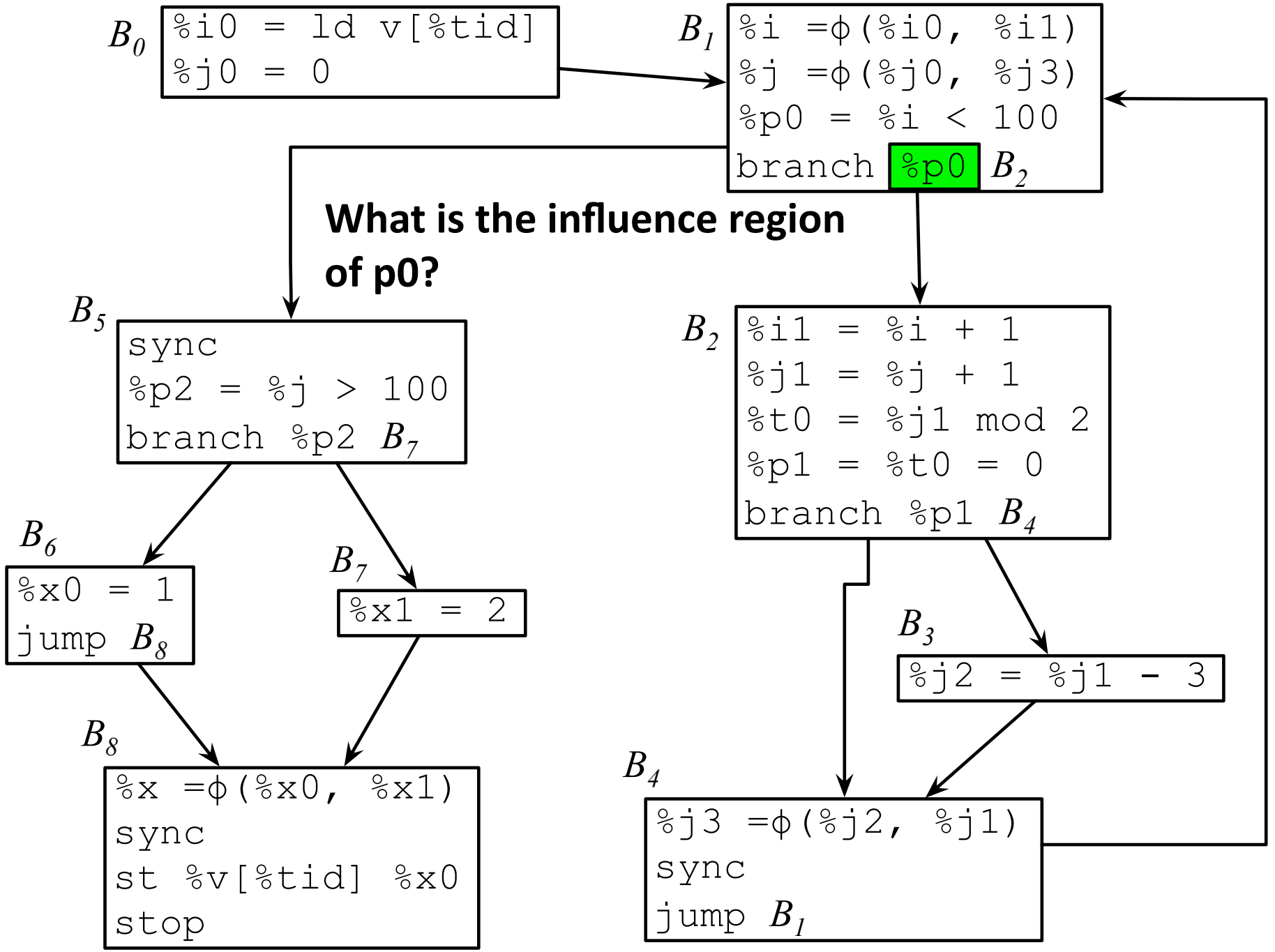


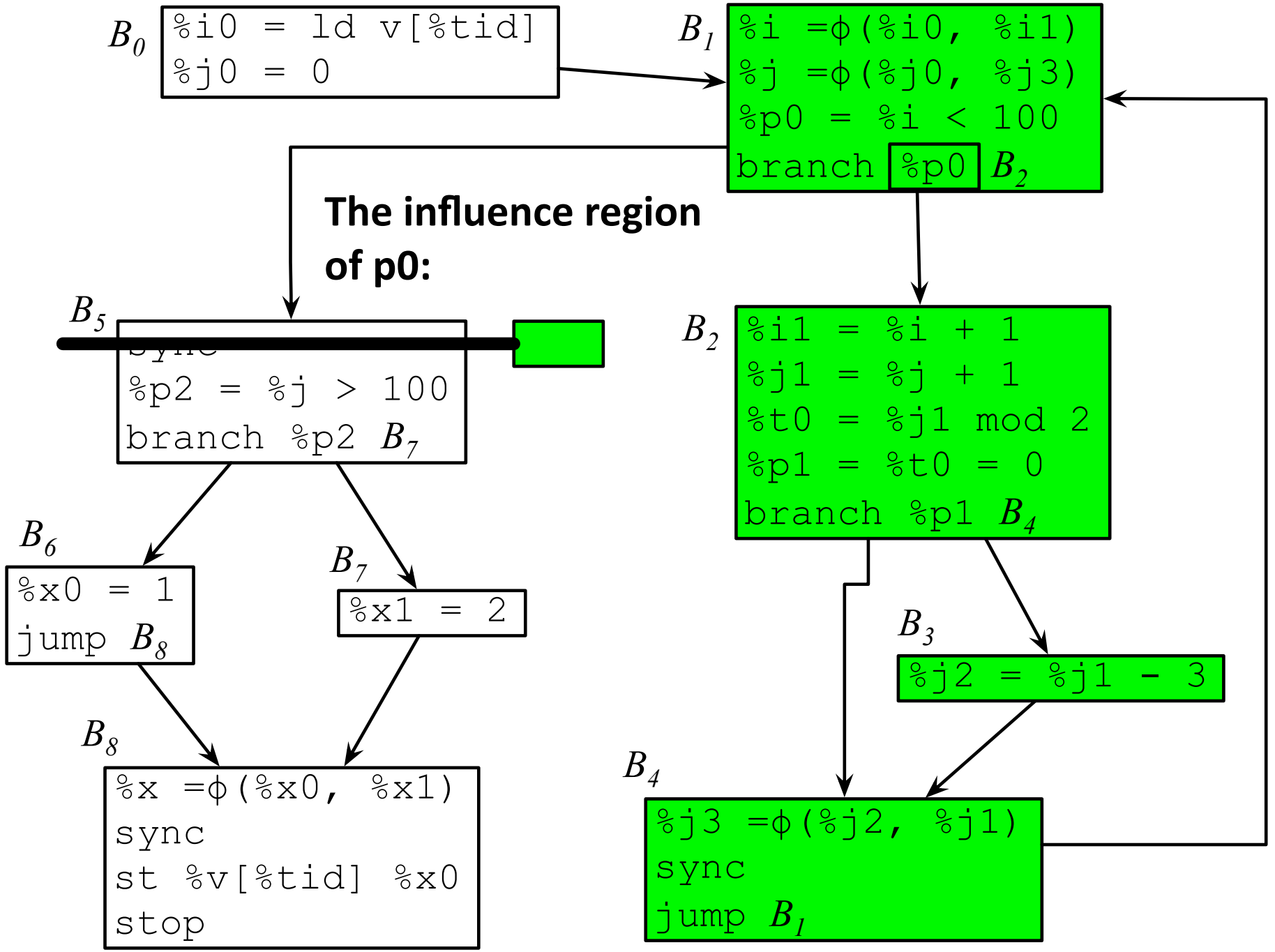

B_0
%i0 = ld v[%tid]
%j0 = 0
B_1
%i =ϕ(%i0, %i1)
%j =ϕ(%j0, %j3)
%p0 = %i < 100
branch %p0 B_2
What is the influence region of p2?
B_5
sync
%p2 = %j > 100
branch %p2 B_7
B_2
%i1 = %i + 1
%j1 = %j + 1
%t0 = %j1 mod 2
%p1 = %t0 = 0
branch %p1 B_4
B_6
%x0 = 1
jump B_8
B_7
%x1 = 2
B_3
%j2 = %j1 - 3
B_8
%x =ϕ(%x0, %x1)
sync
st %v[%tid] %x0
stop
B_4
%j3 =ϕ(%j2, %j1)
sync
jump B_1


B_0
%i0 = ld v[%tid]
%j0 = 0
B_1
%i =ϕ(%i0, %i1)
%j =ϕ(%j0, %j3)
%p0 = %i < 100
branch %p0 B_2
The influence region of p2:
B_5
sync
%p2 = %j > 100
branch %p2 B_7
B_2
%i1 = %i + 1
%j1 = %j + 1
%t0 = %j1 mod 2
%p1 = %t0 = 0
branch %p1 B_4
B_6
%x0 = 1
jump B_8
B_7
%x1 = 2
B_3
%j2 = %j1 - 3
B_8
%x = ϕ(%x0, %x1)
sync
st %v[%tid] %x0
stop
B_4
%j3 =ϕ(%j2, %j1)
sync
jump B_1


B_0
%i0 = ld v[%tid]
%j0 = 0
B_1
%i =ϕ(%i0, %i1)
%j =ϕ(%j0, %j3)
%p0 = %i < 100
branch %p0 B_2
What is the influence region of p1?
B_5
sync
%p2 = %j > 100
branch %p2 B_7
B_2
%i1 = %i + 1
%j1 = %j + 1
%t0 = %j1 mod 2
%p1 = %t0 = 0
branch %p1 B_4
B_6
%x0 = 1
jump B_8
B_7
%x1 = 2
B_3
%j2 = %j1 - 3
B_8
%x =ϕ(%x0, %x1)
sync
st %v[%tid] %x0
stop
B_4
%j3 =ϕ(%j2, %j1)
sync
jump B_1


B_0
%i0 = ld v[%tid]
%j0 = 0
B_1
%i =ϕ(%i0, %i1)
%j =ϕ(%j0, %j3)
%p0 = %i < 100
branch %p0 B_2
The influence region of p1:
B_5
sync
%p2 = %j > 100
branch %p2 B_7
B_2
%i1 = %i + 1
%j1 = %j + 1
%t0 = %j1 mod 2
%p1 = %t0 = 0
branch %p1 B_4
B_6
%x0 = 1
jump B_8
B_7
%x1 = 2
B_3
%j2 = %j1 - 3
B_8
%x =ϕ(%x0, %x1)
sync
st %v[%tid] %x0
stop
B_4
%j3 =ϕ(%j2, %j1)
sync
jump B_1

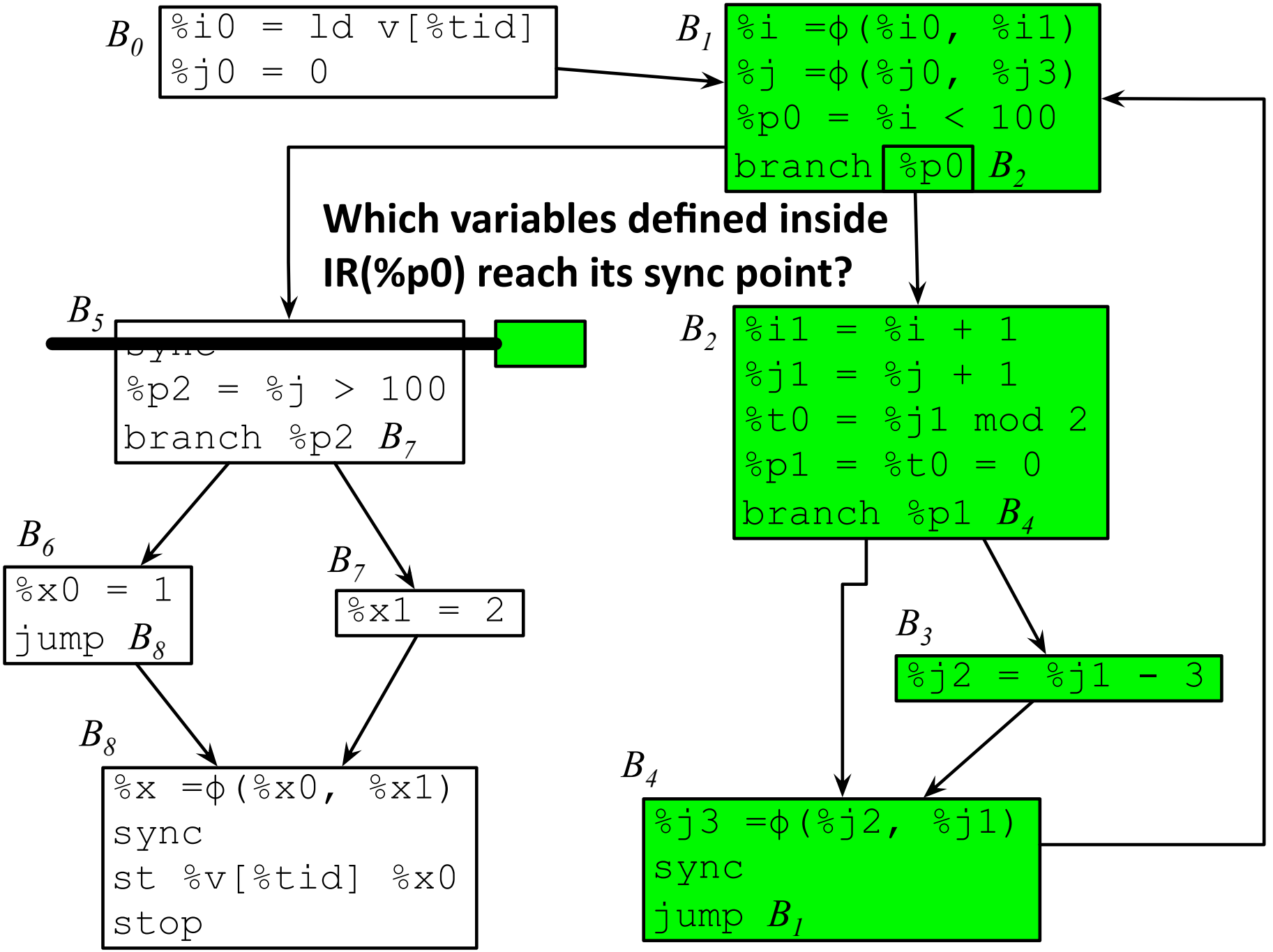

B_0
%i0 = ld v[%tid]
%j0 = 0
B_1
%i =ϕ(%i0, %i1)
%j =ϕ(%j0, %j3)
%p0 = %i < 100
branch %p0 B_2
What is the influence region of p0?
B_5
sync
%p2 = %j > 100
branch %p2 B_7
B_2
%i1 = %i + 1
%j1 = %j + 1
%t0 = %j1 mod 2
%p1 = %t0 = 0
branch %p1 B_4
B_6
%x0 = 1
jump B_8
B_7
%x1 = 2
B_3
%j2 = %j1 - 3
B_8
%x =ϕ(%x0, %x1)
sync
st %v[%tid] %x0
stop
B_4
%j3 =ϕ(%j2, %j1)
sync
jump B_1


B_0
%i0 = ld v[%tid]
%j0 = 0
B_1
%i =ϕ(%i0, %i1)
%j =ϕ(%j0, %j3)
%p0 = %i < 100
branch %p0 B_2
The influence region of p0:
B_5
%p2 = %j > 100
branch %p2 B_7
B_2
%i1 = %i + 1
%j1 = %j + 1
%t0 = %j1 mod 2
%p1 = %t0 = 0
branch %p1 B_4
B_6
%x0 = 1
jump B_8
B_7
%x1 = 2
B_3
%j2 = %j1 - 3
B_8
%x =ϕ(%x0, %x1)
sync
st %v[%tid] %x0
stop
B_4
%j3 =ϕ(%j2, %j1)
sync
jump B_1

fernando@dcc.ufmg.br

The law of sync dependences

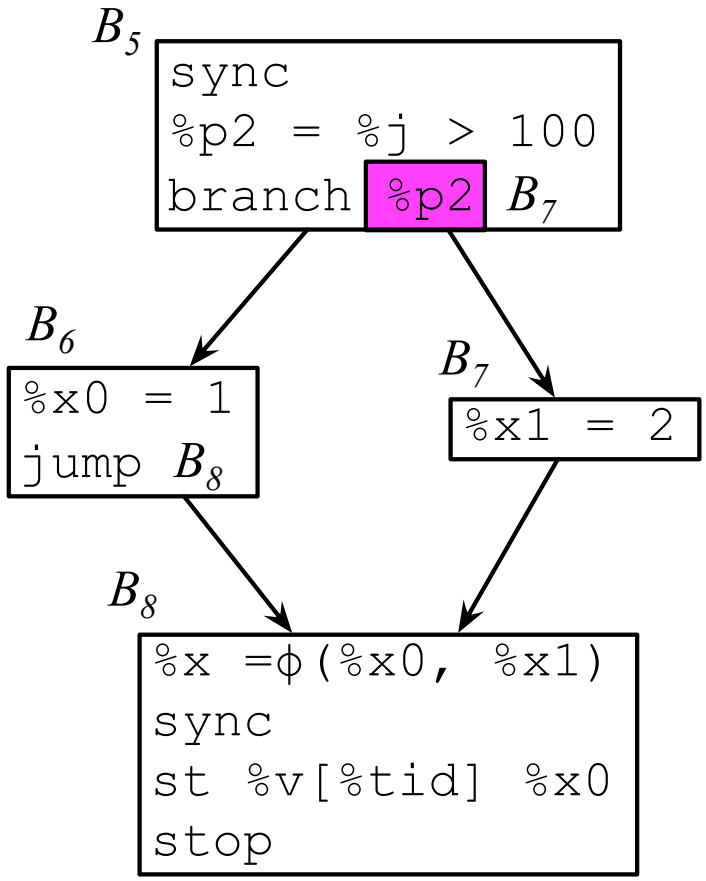
- **Theorem:** Let **`branch %p B`** be a branch instruction, and let *lp* be its synchronization point. A variable *v* is sync dependent on **`%p`** if, and only if, *v* is defined inside the influence region of the branch and *v* ***reaches*** *lp*.
  - Again: *lp* is the post-dominator of the place where the branch is defined.
  - **And alert again**: what does it mean for a variable to **reach** a certain program point?

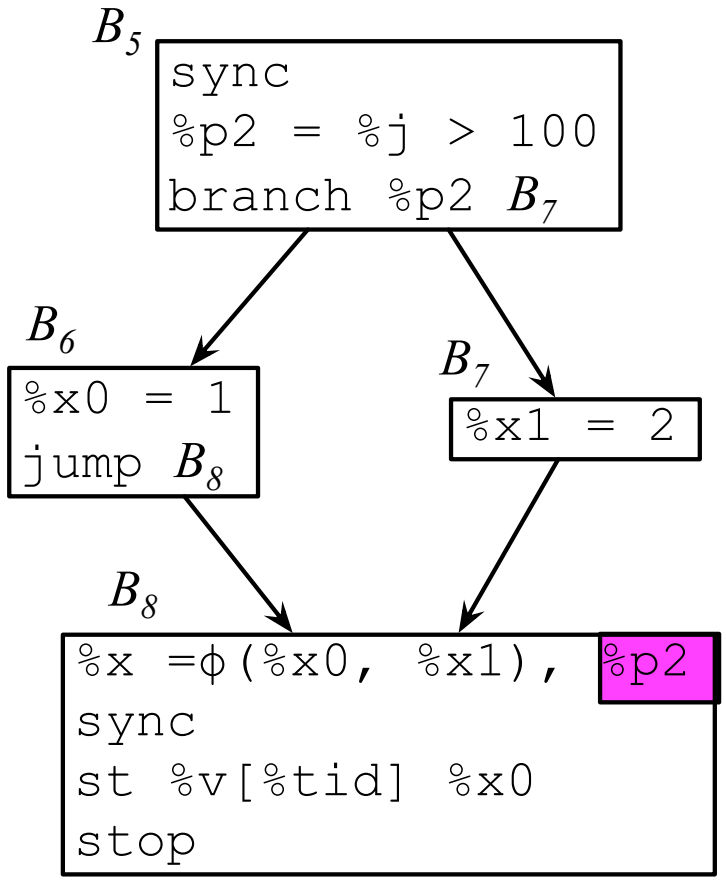

B_0
%i0 = ld v[%tid]
%j0 = 0
B_1
%i =ϕ(%i0, %i1)
%j =ϕ(%j0, %j3)
%p0 = %i < 100
branch %p0 B_2
Which variables defined inside
IR(%p0) reach its sync point?
B_5
%p2 = %j > 100
branch %p2 B_7
B_2
%i1 = %i + 1
%j1 = %j + 1
%t0 = %j1 mod 2
%p1 = %t0 = 0
branch %p1 B_4
B_6
%x0 = 1
jump B_8
B_7
%x1 = 2
B_3
%j2 = %j1 - 3
B_8
%x =ϕ(%x0, %x1)
sync
st %v[%tid] %x0
stop
B_4
%j3 =ϕ(%j2, %j1)
sync
jump B_1

fernando@dcc.ufmg.br

How to transform sync dependences into data dependences?

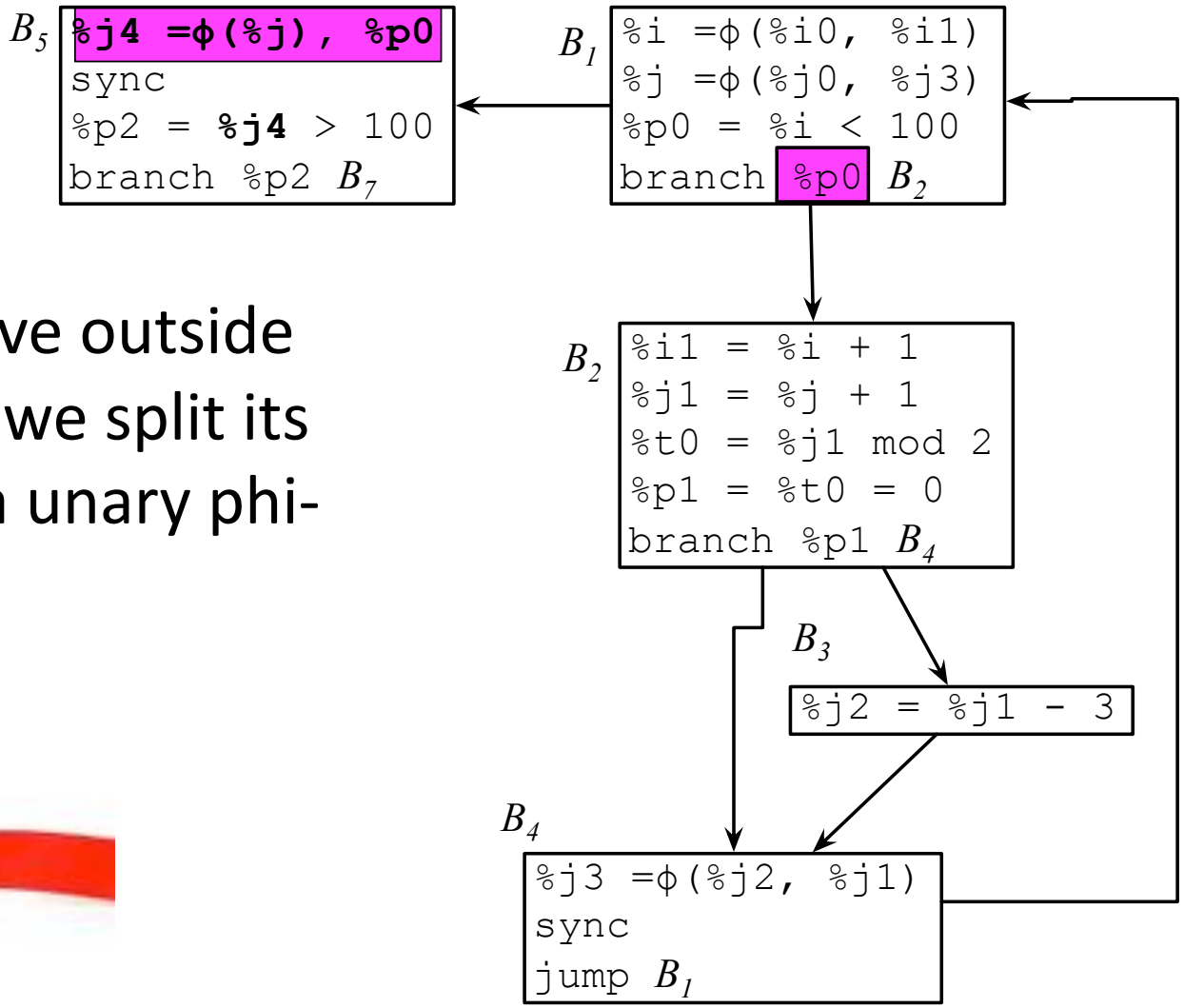
- We augment phi-functions with predicates. For instance, to create a data dependence between %x and %p2:



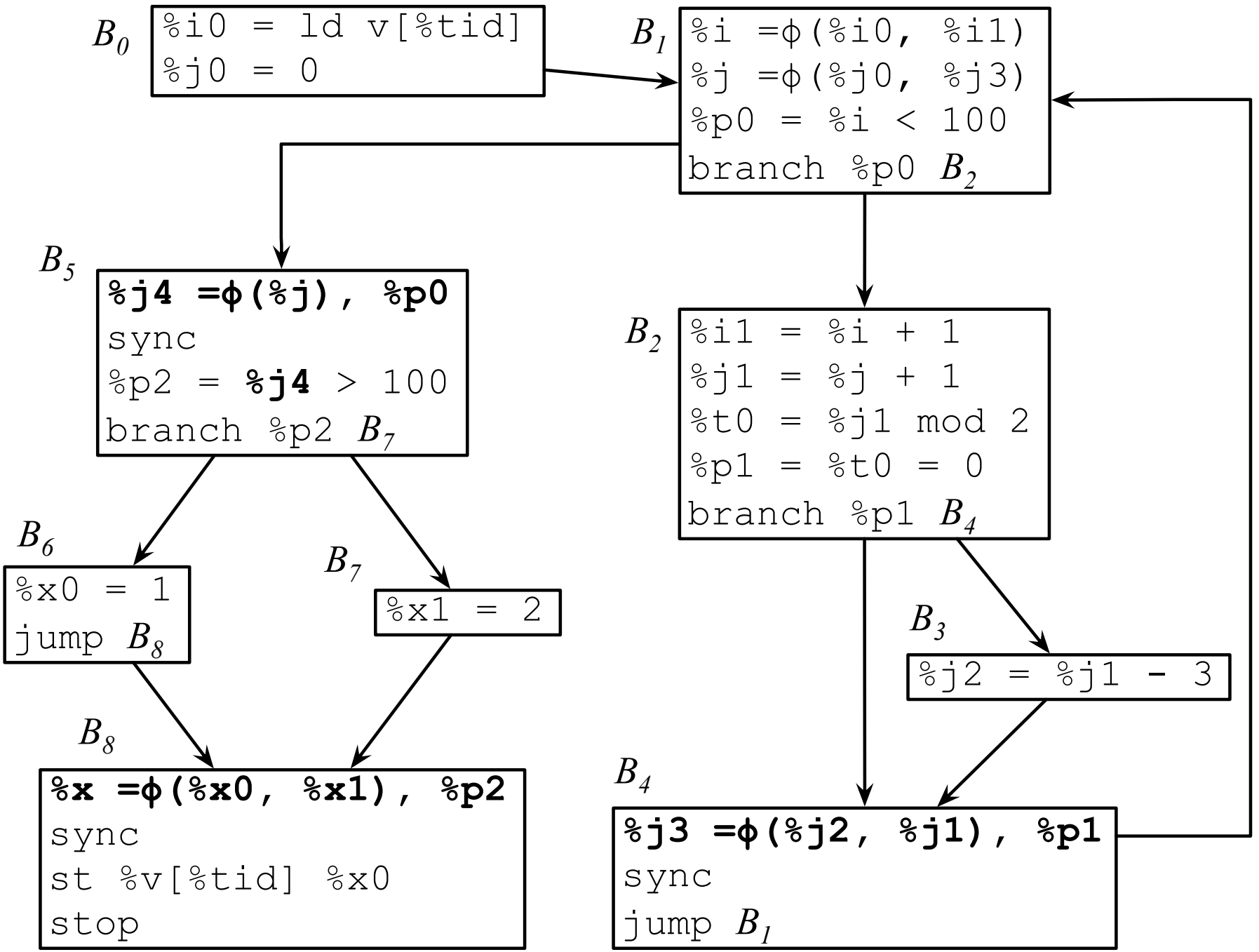


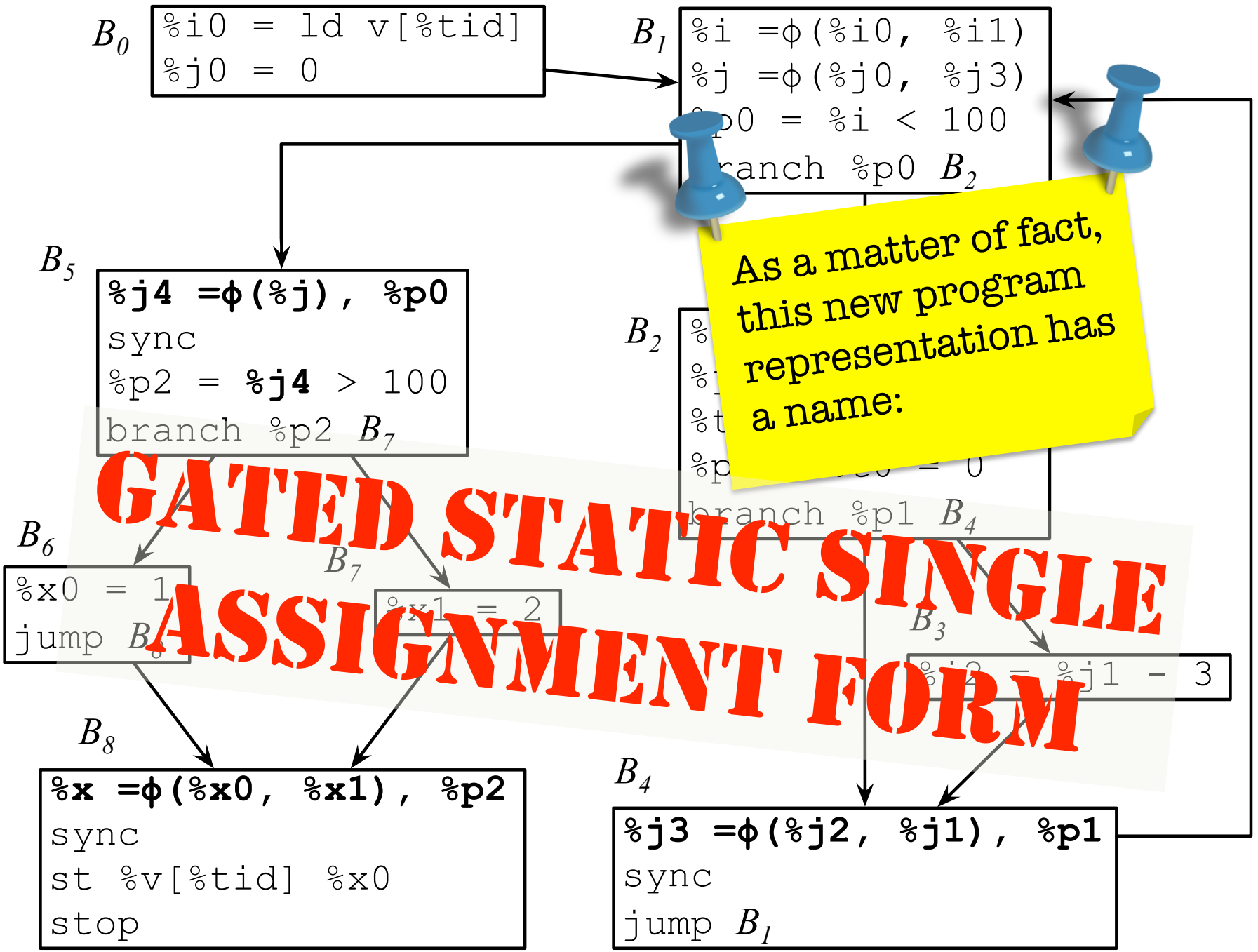
fernando@dcc.ufmg.br

How to transform sync dependences into data dependences?



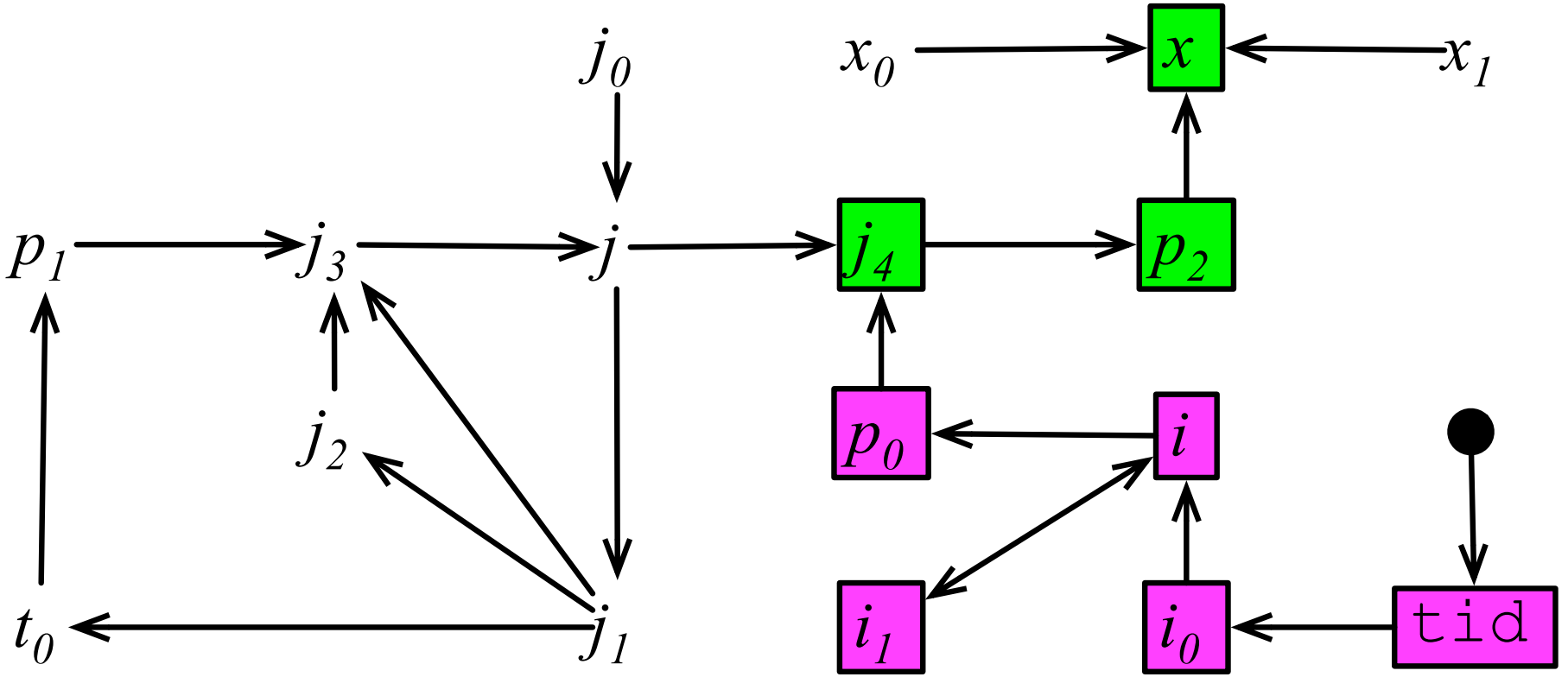
- If a variable is alive outside the IR(%p), then we split its live range, with a unary phi-function.


B_0
%i0 = ld v[%tid]
%j0 = 0
B_1
%i =ϕ(%i0, %i1)
%j =ϕ(%j0, %j3)
%p0 = %i < 100
branch %p0 B_2
B_2
%i1 = %i + 1
%j1 = %j + 1
%t0 = %j1 mod 2
%p1 = %t0 = 0
branch %p1 B_4
B_3
%j2 = %j1 - 3
B_4
%j3 =ϕ(%j2, %j1), %p1
sync
jump B_1
B_5
%j4 =ϕ(%j), %p0
sync
%p2 = %j4 > 100
branch %p2 B_7
B_6
%x0 = 1
jump B_8
B_7
%x1 = 2
B_8
%x =ϕ(%x0, %x1), %p2
sync
st %v[%tid] %x0
stop


B_0
%i0 = ld v[%tid]
%j0 = 0
B_1
%i =ϕ(%i0, %i1)
%j =ϕ(%j0, %j3)
%p0 = %i < 100
branch %p0 B_2
B_5
%j4 =ϕ(%j), %p0
sync
%p2 = %j4 > 100
branch %p2 B_7
B_2
branch %p1 B_4
B_6
%x0 = 1
jump B_8
B_7
%x1 = 2
B_3
%i2 = %j1 - 3
B_8
%x =ϕ(%x0, %x1), %p2
sync
st %v[%tid] %x0
stop
B_4
%j3 =ϕ(%j2, %j1), %p1
sync
jump B_1
As a matter of fact, this new program representation has a name:
GATED STATIC SINGLE ASSIGNMENT FORM

fernando@dcc.ufmg.br

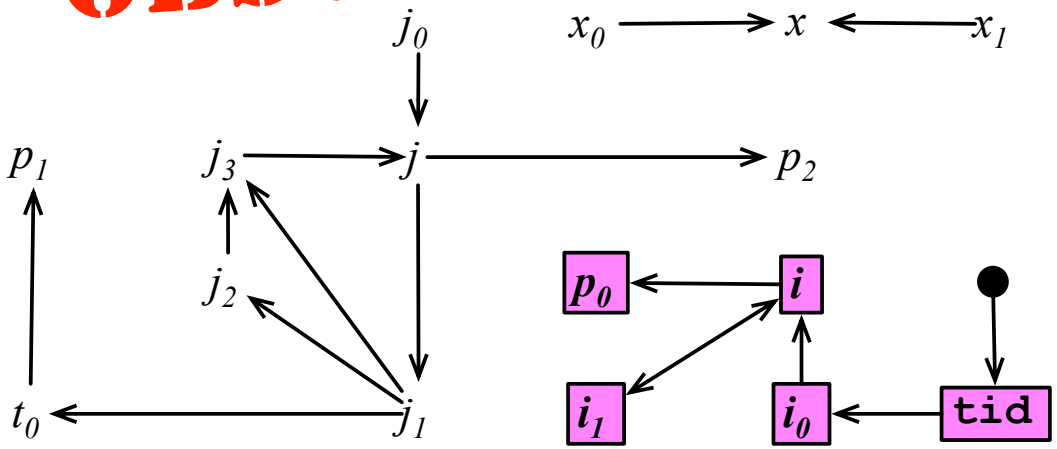
Sync dependence = data dependence

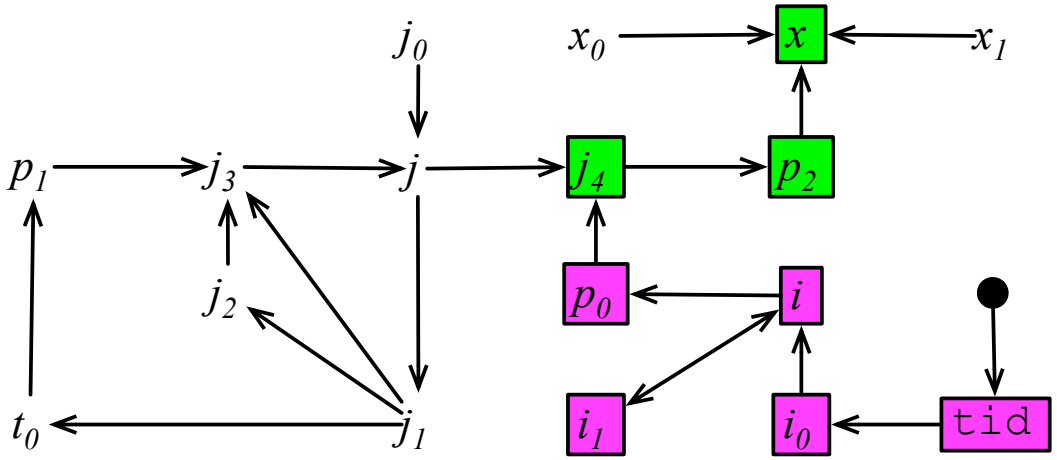


fernando@dcc.ufmg.br

Sync Dependences Change Everything

OBSOLETE

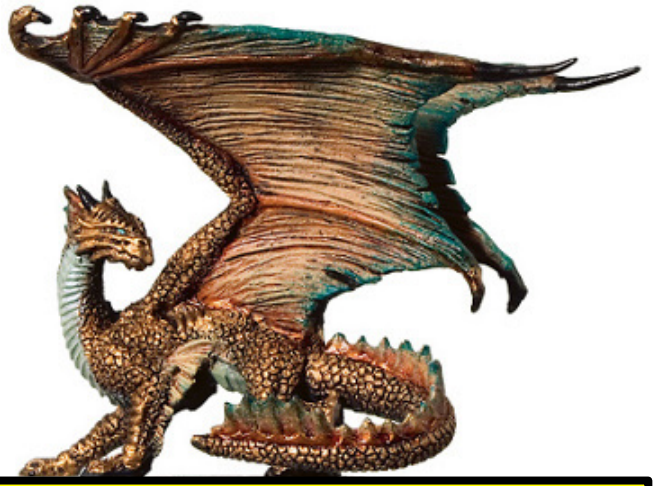




fernando@dcc.ufmg.br

Great! We know which branches are divergent. **And so what?**

- Thread re-location.
- Variable sharing.
- Barrier elimination.
- Branch fusion.



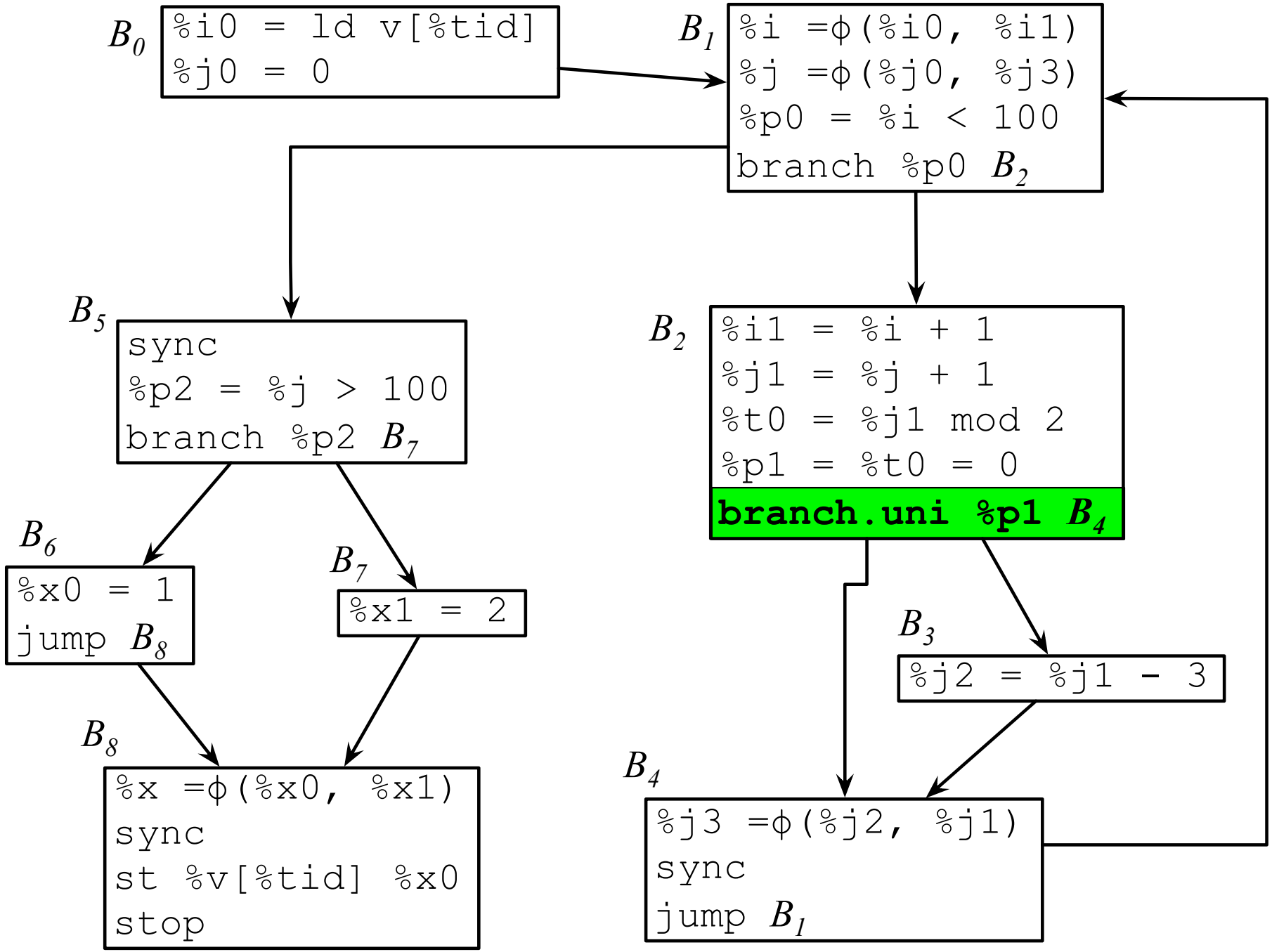
Let developers worry about their *algorithms*, while the **compiler** takes care of *divergences*.

fernando@dcc.ufmg.br

I dream about a world without barriers

"All control constructs are assumed to be divergent points unless the control-flow instruction is marked as uniform, using the **`uni`** suffix. Therefore, a compiler or code author targeting PTX can ignore the issue of divergent threads, but has the opportunity to improve performance by marking branch points as uniform when the compiler or author can guarantee that the branch point is non-divergent."

PTX programmer's manual


B_0
%i0 = ld v[%tid]
%j0 = 0
B_1
%i =ϕ(%i0, %i1)
%j =ϕ(%j0, %j3)
%p0 = %i < 100
branch %p0 B_2
B_5
sync
%p2 = %j > 100
branch %p2 B_7
B_2
%i1 = %i + 1
%j1 = %j + 1
%t0 = %j1 mod 2
%p1 = %t0 = 0
branch.uni %p1 B_4
B_6
%x0 = 1
jump B_8
B_7
%x1 = 2
B_3
%j2 = %j1 - 3
B_8
%x =ϕ(%x0, %x1)
sync
st %v[%tid] %x0
stop
B_4
%j3 =ϕ(%j2, %j1)
sync
jump B_1

fernando@dcc.ufmg.br

Variable Sharing

- The number of threads running on the GPU is bounded by the number of registers. E.g:
  - The GTX 8800 has 8,192 registers, and can run up to 768 threads. How many registers can we assign to each thread to get maximum thread usage?

If necessary, send non-divergent variables to the shared memory to make room for registers.

- Have you seen this kernel before?

From `regPress`, available in the course webpage.

```
__global__ void regPress1(float* In, float* Out, int Width) {
  int tid = blockIdx.x * blockDim.x + threadIdx.x;
  if (tid < Width) {
    Out[tid] = 0.0;
    float a = 2.0F, b = 3.0F, c = 5.0F, d = 7.0F;
    for (int k = tid; k < Width * Width; k += Width) {
      Out[tid] += In[k] / (a - b);
      Out[tid] -= In[k] / (c - d);
      float aux = a;
      a = b;
      b = c;
      c = d;
      d = aux;
    }
  }
}
```

- Which variables are non-divergent?

- Do you remember what we can do with these variables?

```
__global__ void regPress1(float* In, float* Out, int Width) {
  int tid = blockIdx.x * blockDim.x + threadIdx.x;
  if (tid < Width) {
    Out[tid] = 0.0;
    float a = 2.0F, b = 3.0F, c = 5.0F, d = 7.0F;
    for (int k = tid; k < Width * Width; k += Width) {
      Out[tid] += In[k] / (a - b);
      Out[tid] -= In[k] / (c - d);
      float aux = a;
      a = b;
      b = c;
      c = d;
      d = aux;
    }
  }
}
```

From `regPress`, available in the course webpage.

- This is not always worth it. What are the costs of sending data to shared memory?

```
__global__ void regPress2(float* In, float* Out, int Width) {
  int tid = blockIdx.x * blockDim.x + threadIdx.x;
  if (tid < Width) {
    __shared__ float common0, common1;
    float c = 5.0F; float d = 7.0F;
    if (threadIdx.x == 0) { common0 = 2.0F; common1 = 3.0F; }
    __syncthreads();
    Out[tid] = 0.0;
    for (int k = tid; k < Width * Width; k += Width) {
      Out[tid] += In[k] / (common0 - common1);
      Out[tid] -= In[k] / (c - d);
      float aux = common0;
      if (threadIdx.x == 0) { common0 = common1; common1 = c; }
      __syncthreads();
      c = d; d = aux;
    }
  }
}
```

From `regPress`, available in the course webpage.

fernando@dcc.ufmg.br

What the heck is Branch Fusion?

Find the longest sequences of common instructions in the two paths of a branch, and merge them.

- Branch fusion finds common work for **lefties** and **righties**, bringing more *harmony* to paralland.

fernando@dcc.ufmg.br

Let's consider an example:

```
__global__ void exampleKernel
(float* u, float* v, float c1, float c2) {
  y = u[tid];
  if (y != 0.0) {
    float x = v[tid];
    v[tid] = c1*x*x*x/y*y + c2*x;
  } else {
    float x = v[tid];
    v[tid] = c1*x*x/2.0 + c2*x*x;
  }
}
```

What is the longest chain of common instructions?

```
1  %t0 = %u[%tidx]
2  %p2 = ne %t1 0.0
3  bra %p2 (12)
```

```
12 %t1 = %v[%tidx]
13 %t2 = %t1 * %t1
14 %t3 = %t2 * %t1
15 %t4 = %t3 * %c1
16 %t5 = %t4 / %t0
17 %t6 = %t5 / %t0
18 %t7 = %t1 * %c2
19 %t8 = %t6 + %t7
20 %v[%tidx] = %t8
```

```
4  %t9 = %v[%tidx]
5  %t10 = %t9 * %t9
6  %t11 = %t10 * %c1
7  %t12 = %t11 / 2.0
8  %t13 = %t9 * %c2
9  %t14 = %t13 * %t9
10 %t15 = %t12 + %t14
11 %v[%tidx] = %t8
```

fernando@dcc.ufmg.br

How do we do branch fusion?

- The answer comes from *computational biology*:
  - **Smith-Waterman** sequence alignment.
  - But, instead of **genes**, we match **instructions**.

fernando@dcc.ufmg.br

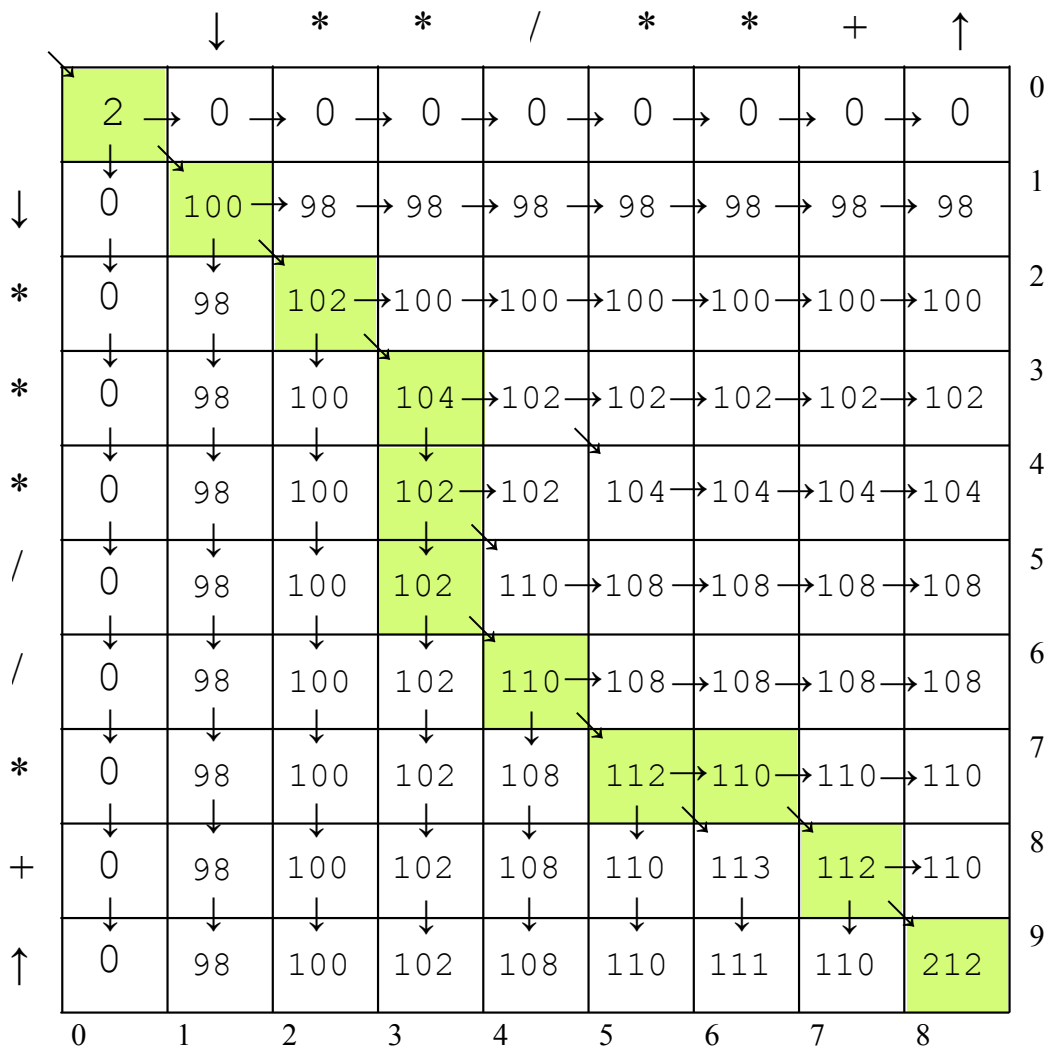
Let's try to align the instructions:

	✓	✓	✓	✗	✗	✓	✗	✓	✓	✓
$T =$	$\downarrow$	*	*	_	_	/	*	*	+	$\uparrow$
$F =$	$\downarrow$	*	*	*	/	/	_	*	+	$\uparrow$

How to find these chains automatically?

fernando@dcc.ufmg.br

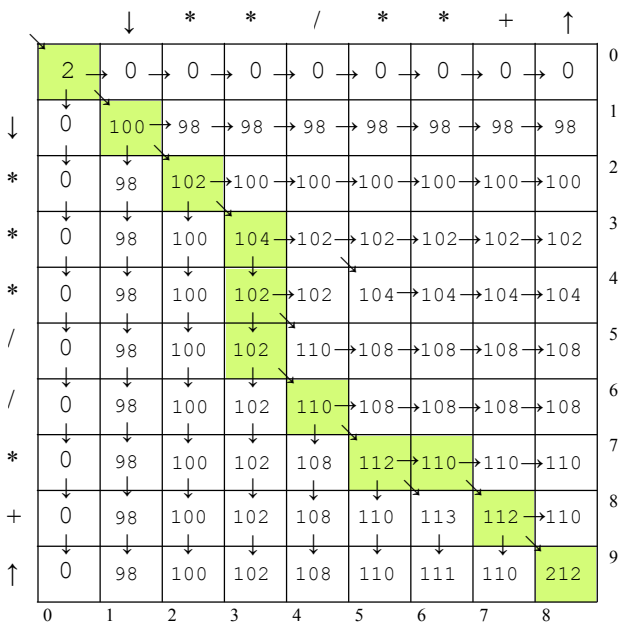
Smith-Waterman



fernando@dcc.ufmg.br

The Profitability Matrix

- Each cell is the profit of merging two instructions.
- What is the profit of merging two divisions?
- What represents the most profitable path inside the matrix?
- What are the diagonals in this path?
- What are the vertical and horizontal paths?
  - Is there a cost in leaving a diagonal?
  - Is there a cost in entering a diagonal?



fernando@dcc.ufmg.br

Touché!

```
1  %t0 = %u[%tidx]
2  %p2 = ne %t1 0.0
3  bra %p2 (12)
```

```
12 %t1 = %v[%tidx]
13 %t2 = %t1 * %t1
14 %t3 = %t2 * %t1
15 %t4 = %t3 * %c1
16 %t5 = %t4 / %t0
17 %t6 = %t5 / %t0
18 %t7 = %t1 * %c2
19 %t8 = %t6 + %t7
20 %v[%tidx] = %t8
```

```
4  %t9 = %v[%tidx]
5  %t10 = %t9 * %t9
6  %t11 = %t10 * %c1
7  %t12 = %t11 / 2.0
8  %t13 = %t9 * %c2
9  %t14 = %t13 * %t9
10 %t15 = %t12 + %t14
11 %v[%tidx] = %t8
```

```
1  %t0 = %u[%tidx]
2  %p2 = ne %t1 0.0
3  %t1_9 = %v[%tidx]
4  %t2_10 = %t1_9 * %t1_9
5  %s1 = %p2 ? %t2_10 : %c1
6  %t3_11 = %t2_10 * %s1
7  bra %p2 (13)
```

```
13 %t4 = %t3 * %c1
14 %t5 = %t4 / %t0
```

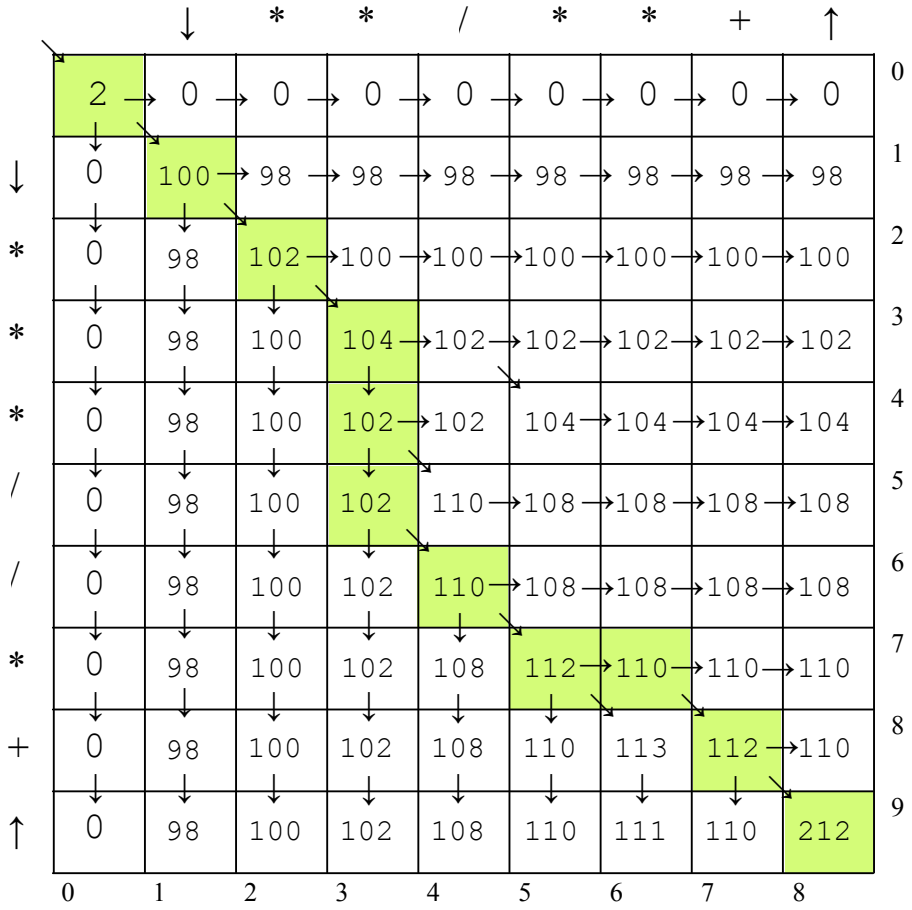
```
8  %s2 = %p2 ? %t5 : %t3_11
9  %s3 = %p2 ? %t0 : 2.0
10 %t6_12 = %s2 / %s3
11 %t7_13 = %t1_9 * %c2
12 bra %p2 (16)
```

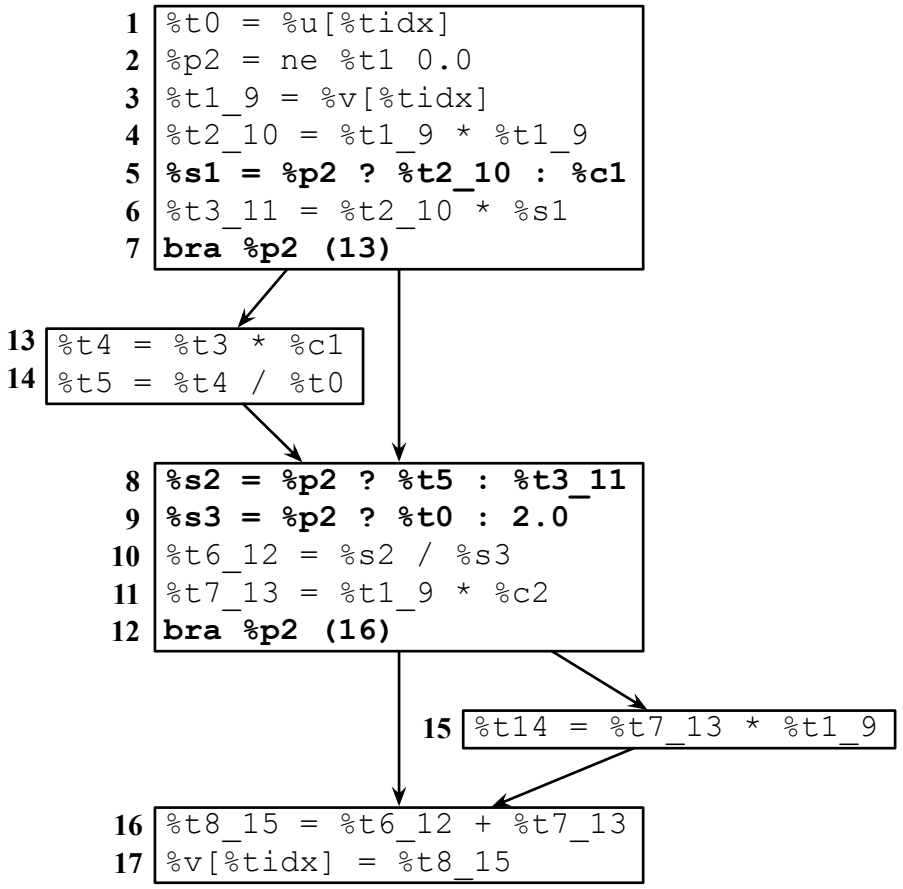
```
15 %t14 = %t7_13 * %t1_9
```

```
16 %t8_15 = %t6_12 + %t7_13
17 %v[%tidx] = %t8_15
```

fernando@dcc.ufmg.br

The matrix and the program





Data Re-location

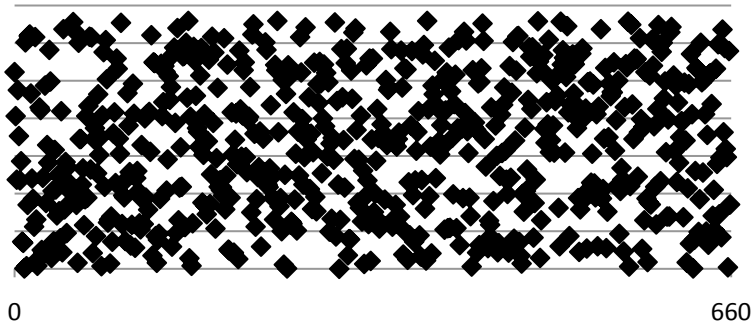
- Sometimes it pays off spending some time to reorder the data before processing it.
  - Or then we can try to "quasi-sort" these data.
- What could we do to improve the execution of this old acquaintance of ours?

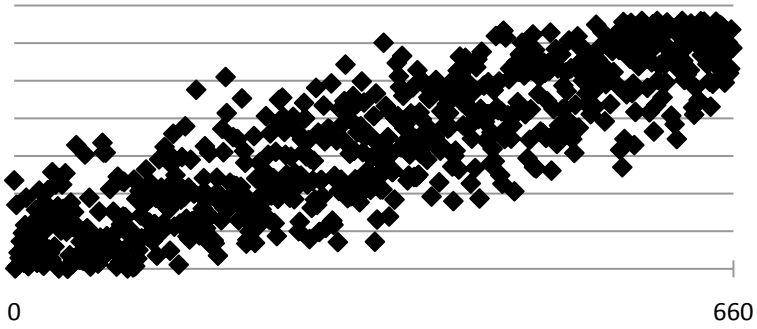
```
__global__ void dec2zero(int* v, int N) {
  int xIndex = blockIdx.x*blockDim.x+threadIdx.x;
  if (xIndex < N) {
    while (v[xIndex] > 0) {
      v[xIndex]--;
    }
  }
}
```

fernando@dcc.ufmg.br

Quasi-sorting

- Copy data to shared memory
- Each thread sorts small chunks of data, say, four cells
- Scatter the data around the shared memory:
  - Each thread puts its smallest element in the first ¼ of the array, the second smallest in the second ¼ of the array, and so forth.





fernando@dcc.ufmg.br

The Rainbow's End

- Many conferences take papers on optimization of GPU programs:
  - PLDI, POPL, PPoPP, PACT, MICRO, ASPLOS
- There are many discussion forums
  - Nvidia, Ocelot
- Books
  - Computer Organization and Design
  - Lots of Nvidia on-line books