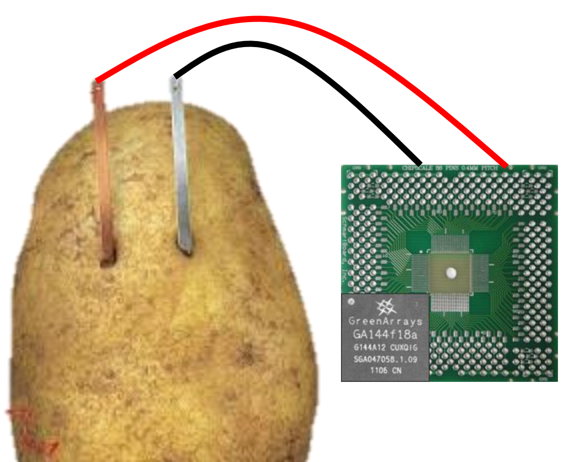
Compiling a Gesture Recognition Application for an Ultra Low-Power Architecture

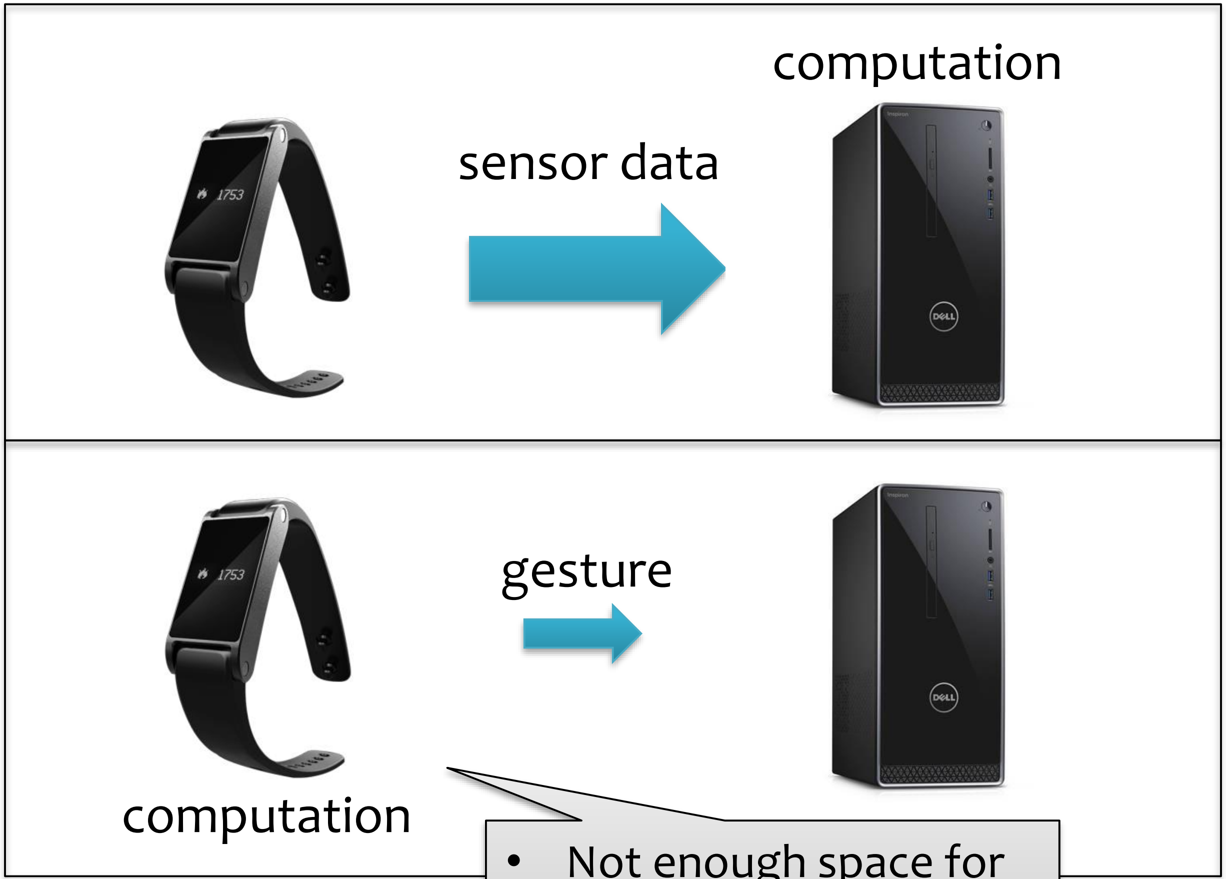
Phitchaya Mangpo Phothilimthana
Michael Schuldt
Berkeley
UNIVERSITY OF CALIFORNIA

Rastislav Bodik
UNIVERSITY of WASHINGTON

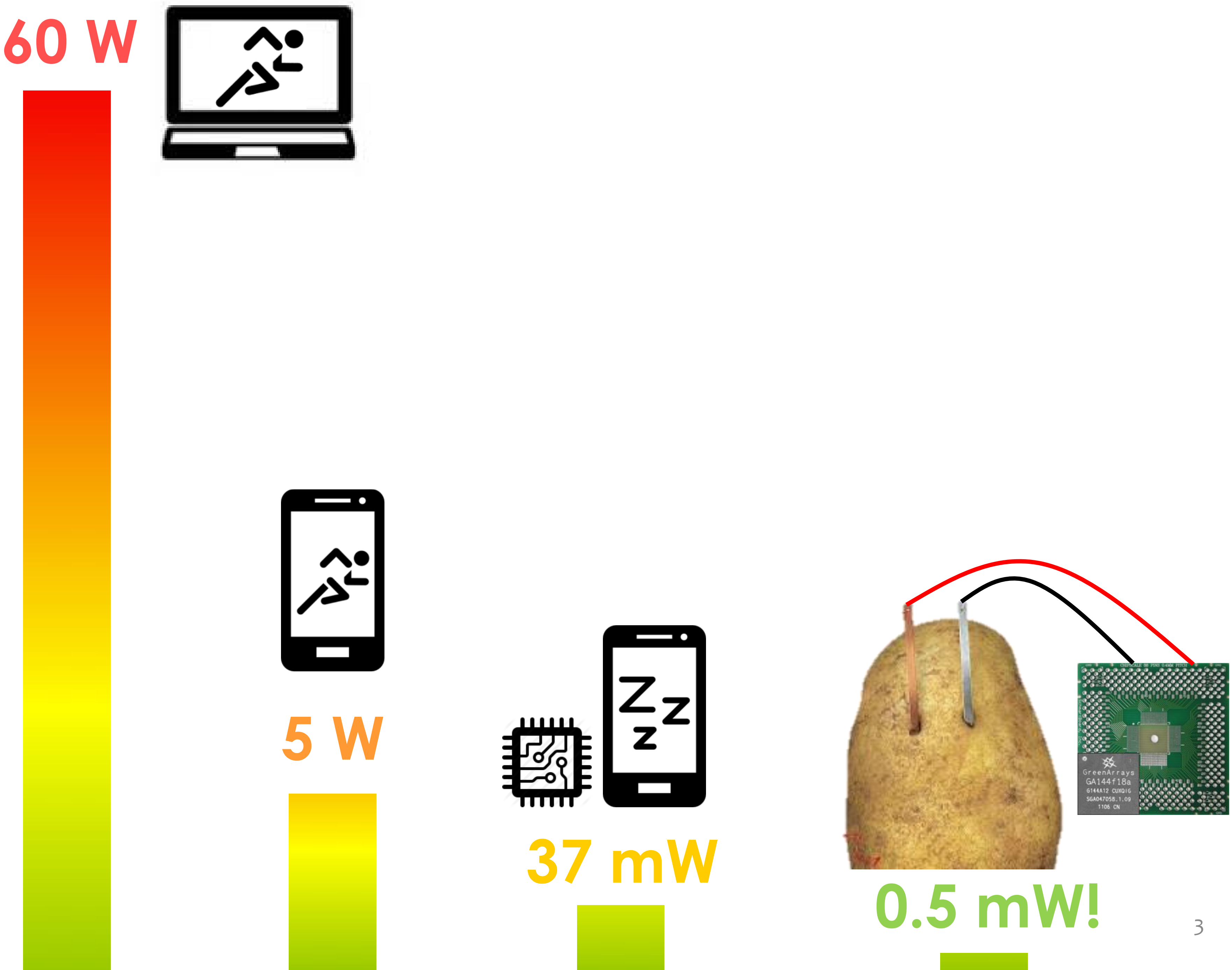
Classification Applications

- Intensive computation
- Large data storage for gesture models

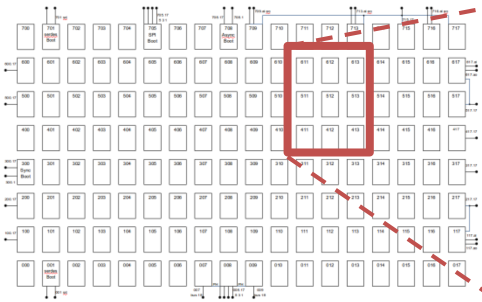




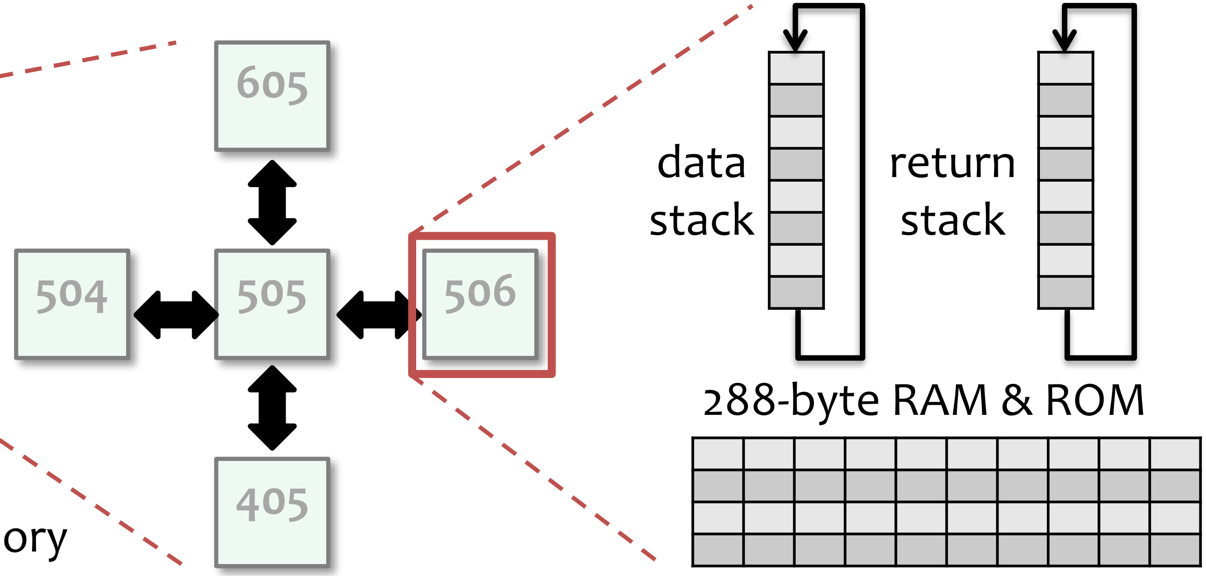
- Not enough space for data and program
- Not enough energy
- Too slow


60 W
5 W
37 mW
0.5 mW!
GreenArrays
GA144f18a

GreenArrays: Ultra Low Power Processor



144 cores



- No cache, no shared memory
- Stack-based 18-bit architecture
- 32 instructions

Programming challenges
- Manually partition a program across 144 cores
- Explicitly manage communication between cores
- Program in assembly-like stack-based language

How to program and compile for highly-constrained multicore processors with very small distributed memory?

Key idea: partition a program smartly to fit the code in tiny cores and achieve fast execution time by balancing **communication** and **code replication**.

In the context of partitioning both:
1. data & computation
2. control flow statements

Existing Solution

Chlorophyll

[Phothilimthana et al. PLDI'14]

Input: imperative sequential program

Output: per-core assembly programs

Compilation strategy: similar to SPMD

- Output per-core programs have the same structure (control flow).
- Data and computation are distributed across cores.

Extra input:
partial constraints on data and computation partitioning

Spatial programming model

```
int x[12];
for(i from 1 to 12)
  x[i] += x[i-1];
```

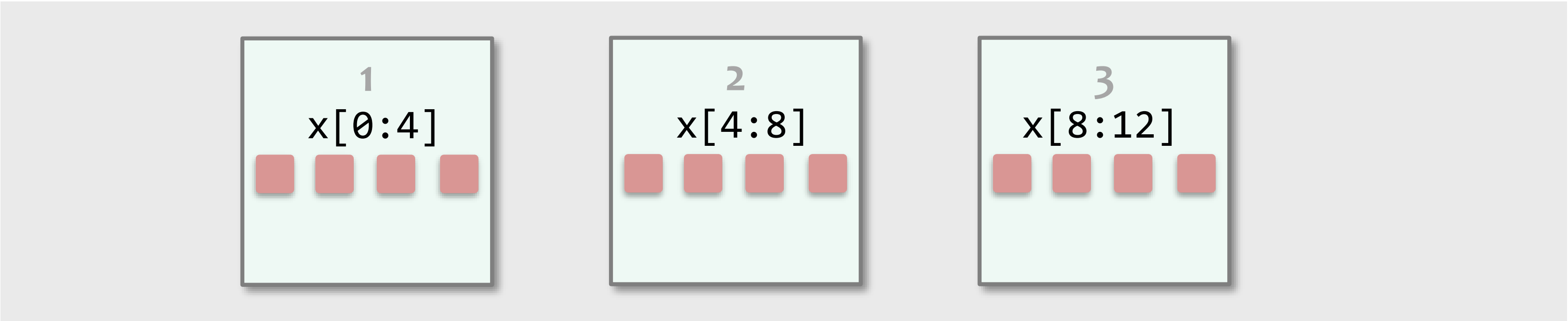
Spatial programming model

```
int[4]@{1,2,3} x[12];
for(i from 1 to 12)
  x[i] += x[i-1];
```

Partition Type

pins data and operators to specific partitions (logical cores)

Similar to [Chandra *et al.* PPoPP'08]



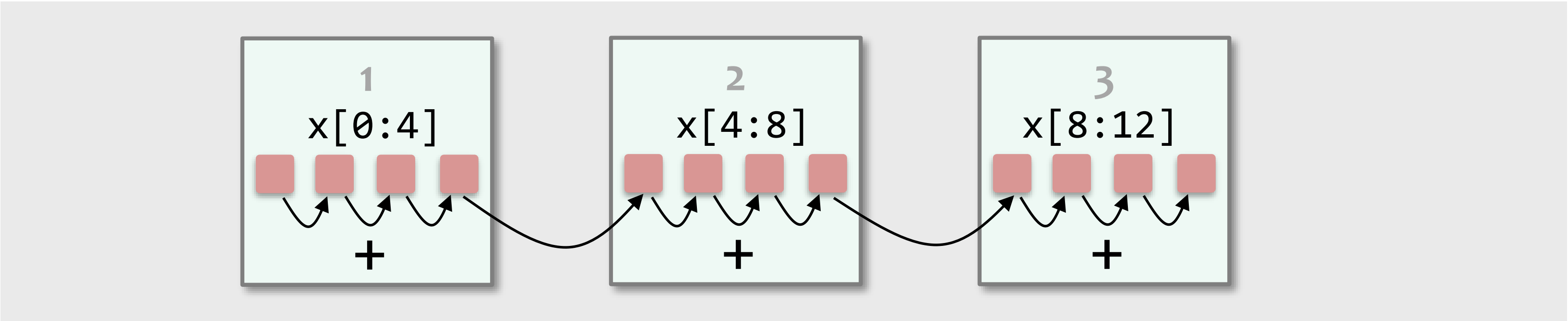
Spatial programming model

```
int[4]@{1,2,3} x[12];
for(i from 1 to 12)
  x[i] +=@loc(x[i]) x[i-1];
```

Partition Type

pins data and operators to specific partitions (logical cores)

Similar to [Chandra *et al.* PPoPP'08]



Incomplete Annotations

```
int[4] x[12];
for(i from 1 to 12)
  x[i] += x[i-1];
```

Incomplete Annotations

```
int[4]@?? x[12];
for(i from 1 to 12)
  x[i] +=@?? x[i-1];
```

Program + some partition annotations

Partition Type Inference

Complete partition annotations

Hard constraint:

Code fits in each logical core (partition).

Objective:

Minimize number of messages sent between partitions.

Problems

Problem I	Generated code is too large.
Cause	**Control flow statements partitioning** strategy exploits **code replication** but **not enough communication**.

Problem II	Slow execution time (no parallelism)
Cause	**Data & computations partitioning** strategy does **not exploit code replication**.

Problems

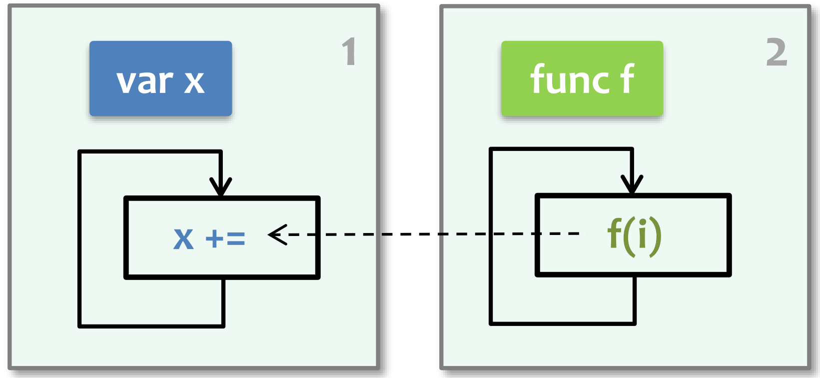
Problem I	Generated code is too large.
Cause	**Control flow statements partitioning** strategy exploits **code replication** but not enough **communication**.

Problem II	Slow execution time (no parallelism)
Cause	**Data & computations partitioning** strategy does not exploit **code replication.**

Control Flow Partitioning

SPMD strategy (original)

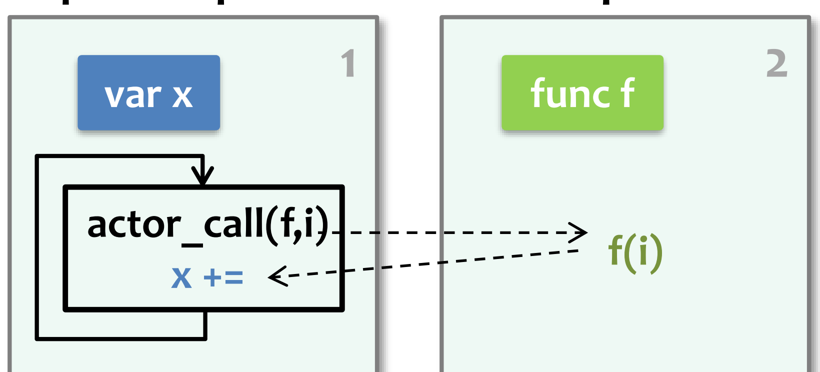
```
// source
int@2 f(int@2 i)
{ ... }

int@1 x;
for(i from 0 to 100)
  x += f(i);
```



More code
Less communication

Replicates relevant control flow constructs onto every partition.

“Requestor partition” **“Actor partition”**



Less code
More communication

Actor strategy

```
// annotate with
actor f;
```

Sends a request to execute code to avoid control flow duplication.

Compiling with Mixed Strategy

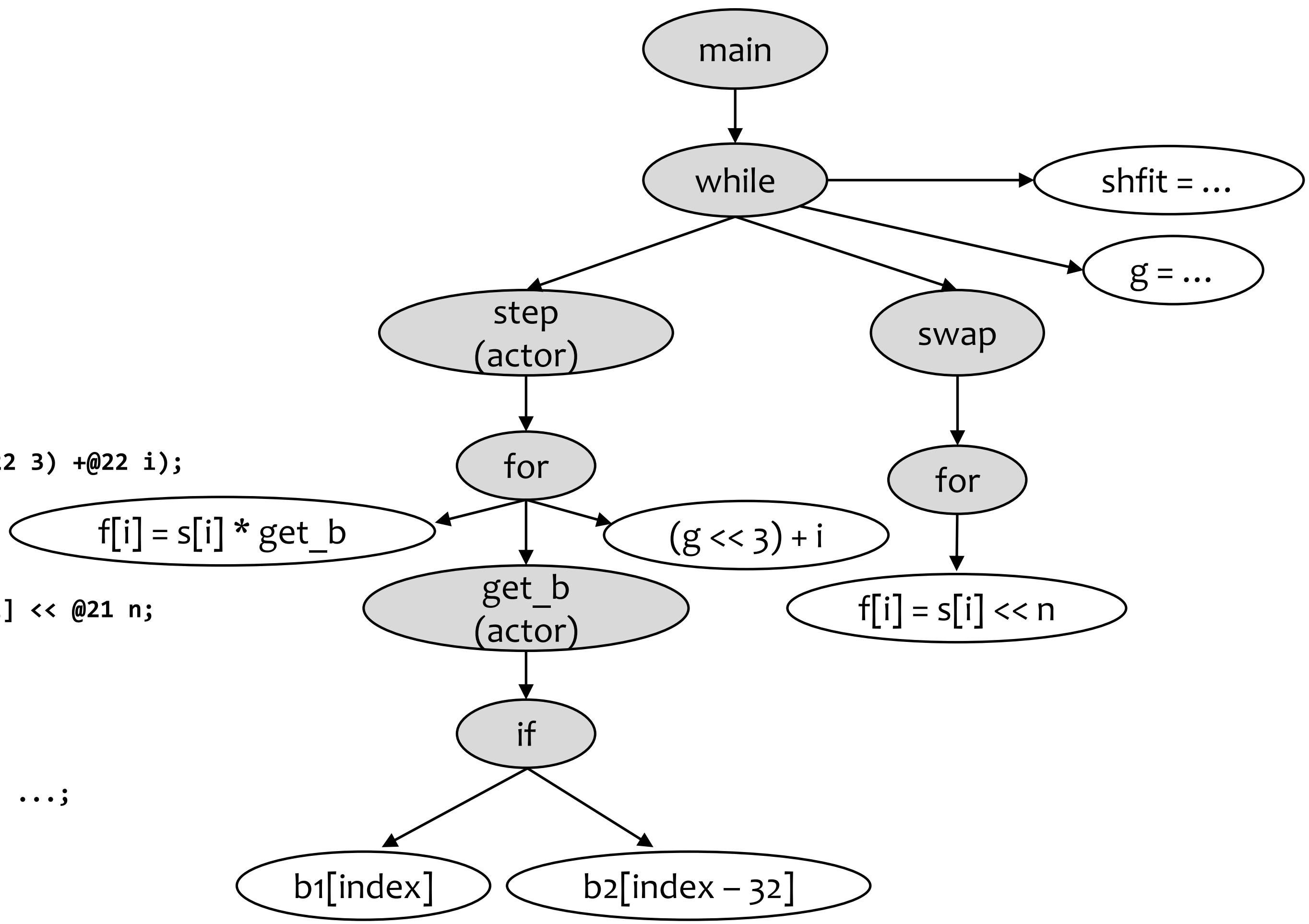
Given a program with a complete partitioning assignment, how to generate code for each partition?

- Which control flow constructs need to be replicated in each partition?
- Which partition is a “requestor” of a function?
- Which partitions are “actors” of a function?

Compiling with Mixed Strategy

```
fix1_t@21 f[8];      fix1_t@11 s[8];
fix1 t@23 b1[32];    fix1 t@13 b2[32];

actor get_b;
fix1 t@23 get_b(int@23 index) {
  if (index <@23 32)
    return b1[index] ;
  else
    return b2[index -@13 32];
}

actor step;
void step(int@22 g) {
  for (i from 0 to 8)
    f[i] = s[i] *@22 get_b((g <<@22 3) +@22 i);
}

void swap(int@21 n) {
    for (i from 0 to 8) s[i] = f[i] << @21 n;
}

void main() {
  while(1) {
    int@32 g = ...; int@32 shift = ...;
    step(g);
    swap(shift);
  }
}
```

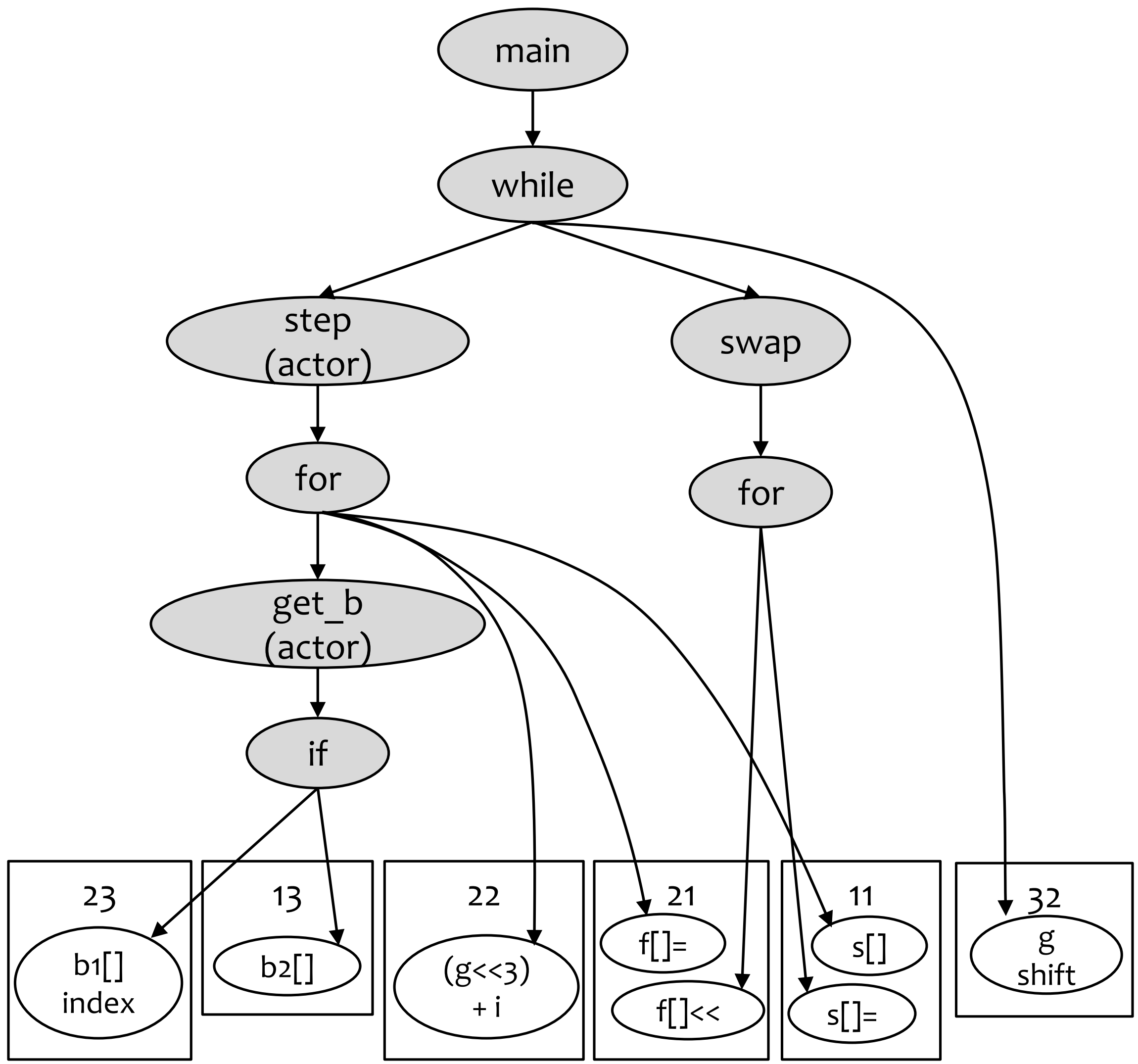
Control Dependence Graph (CDG)



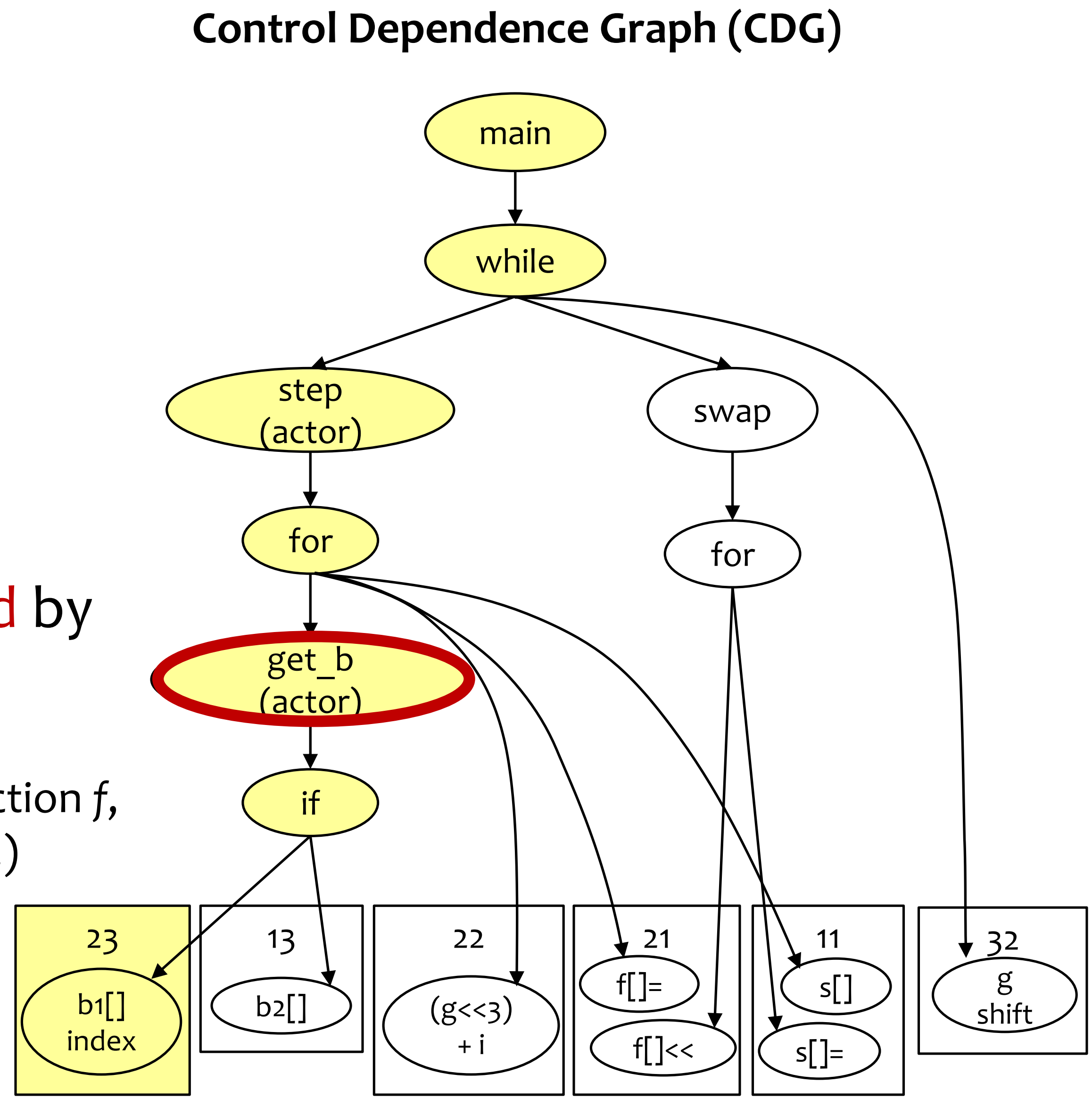
Compiling with Mixed Strategy

```
fix1_t@21 f[8];      fix1_t@11 s[8];
fix1 t@23 b1[32];    fix1 t@13 b2[32];

actor get_b;
fix1 t@23 get_b(int@23 index) {
  if (index <@23 32)
    return b1[index] ;
  else
    return b2[index -@13 32];
}

actor step;
void step(int@22 g) {
  for (i from 0 to 8)
    f[i] = s[i] *@22 get_b((g <<@22 3) +@22 i);
}

void swap(int@21 n) {
    for (i from 0 to 8) s[i] = f[i] << @21 n;
}

void main () {
  while(1) {
    int@32 g = ...; int@32 shift = ...;
    step(g);
    swap(shift);
  }
}
```

Control Dependence Graph (CDG)



Compiling with Mixed Strategy



Partition 23 is dominated by actor function get_b.

(Partition *p* is dominated by function *f*, if all paths from main to *p* pass *f*.)

Compiling with Mixed Strategy

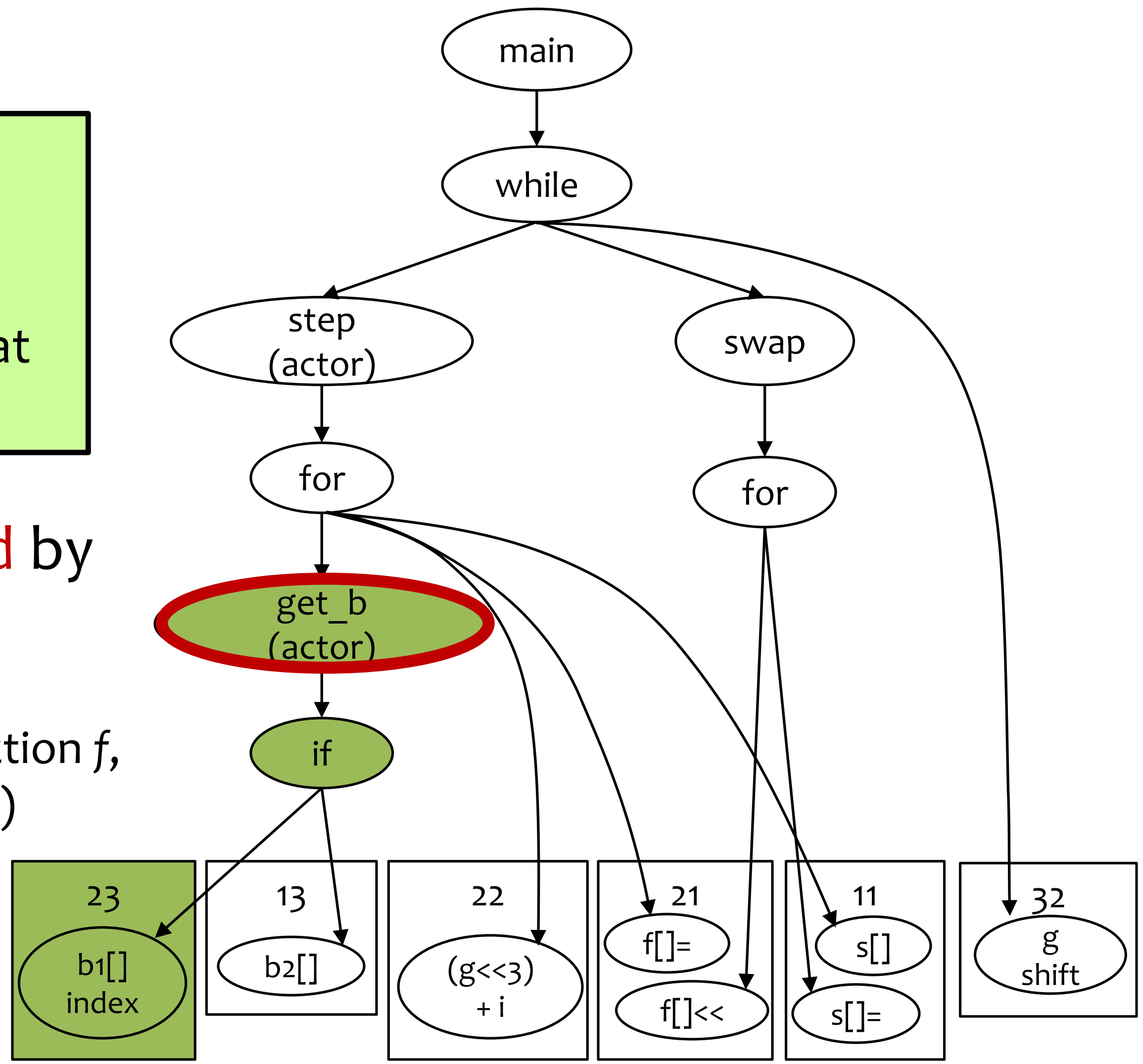
Therefore,

- Partition 23 is an actor of **get_b.**
- Control flow of 23 starts at **get_b.**

Partition 23 is dominated by actor function **get_b.**

(Partition *p* is dominated by function *f*, if all paths from main to *p* pass *f*.)

Control Dependence Graph (CDG)

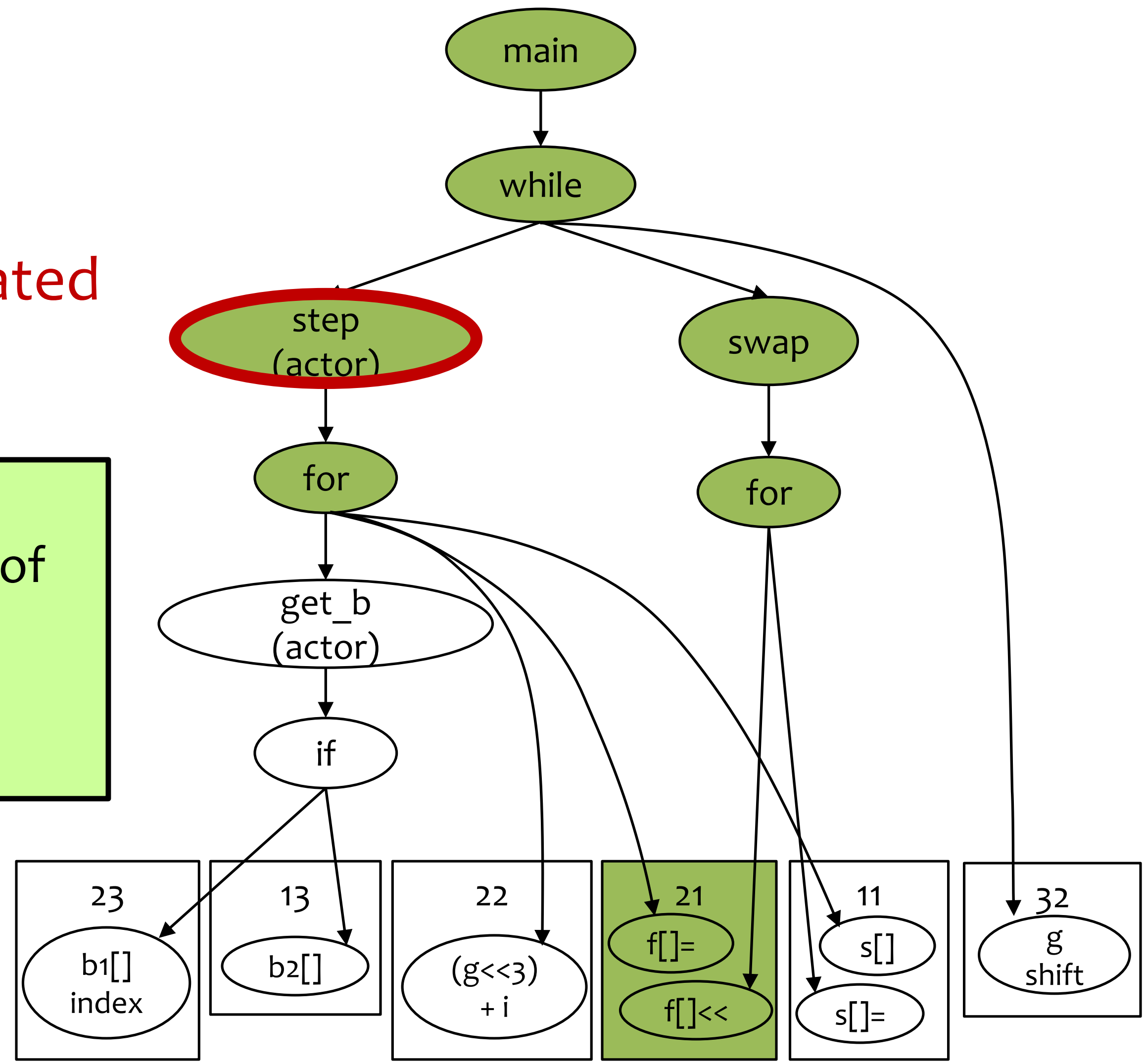


Compiling with Mixed Strategy

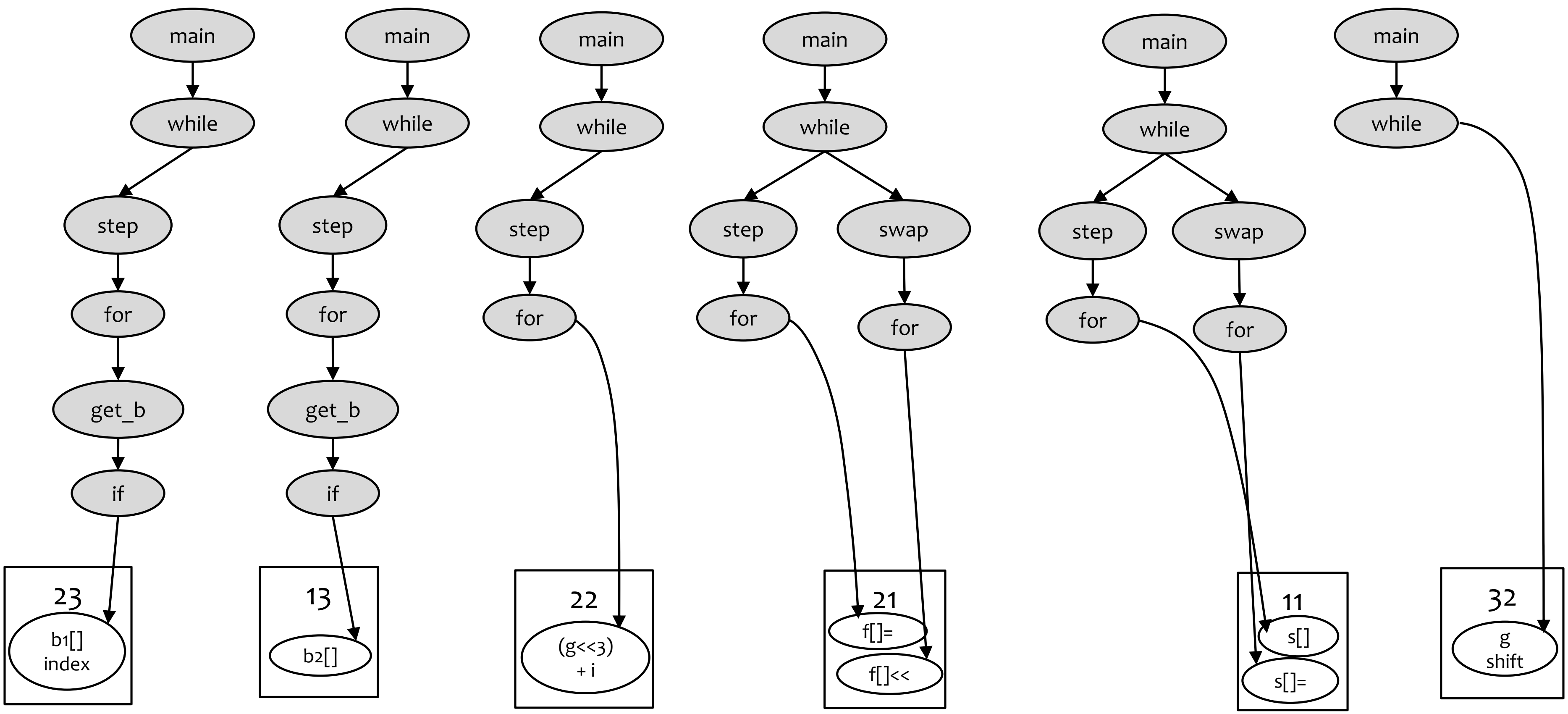
Partition 21 is not dominated by actor function **`step`**.

Therefore,
- Partition 21 is not an actor of **step**.
- Control flow of 21 starts at **main**.

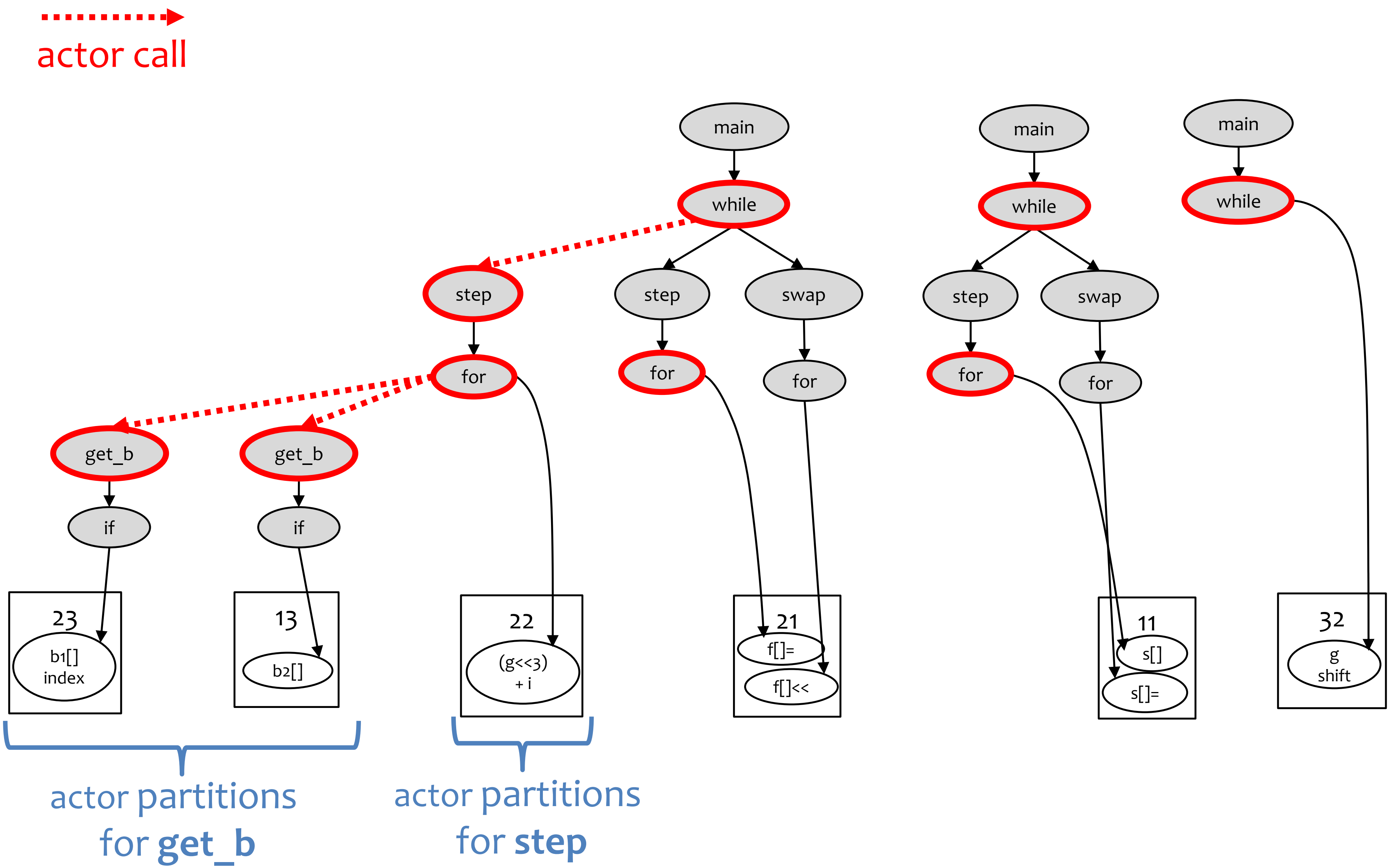
Relevant control flow slice of partition 21



SPMD Strategy



SPMD + Actor Strategy

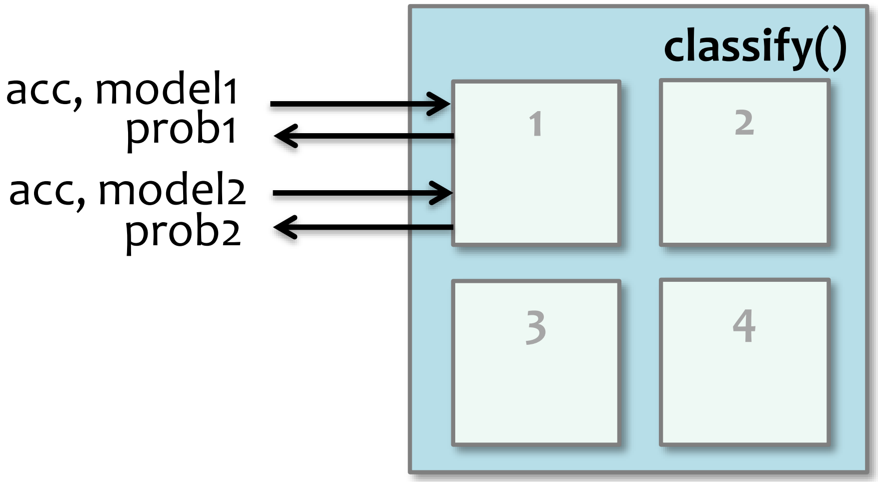


Problems

Problem I	Generated code is too large.
Cause	**Control flow statements partitioning** strategy exploits **code replication** but not enough **communication.**
Problem II	Slow execution time (no parallelism)
Cause	**Data & computations partitioning** strategy does not exploit **code replication**.

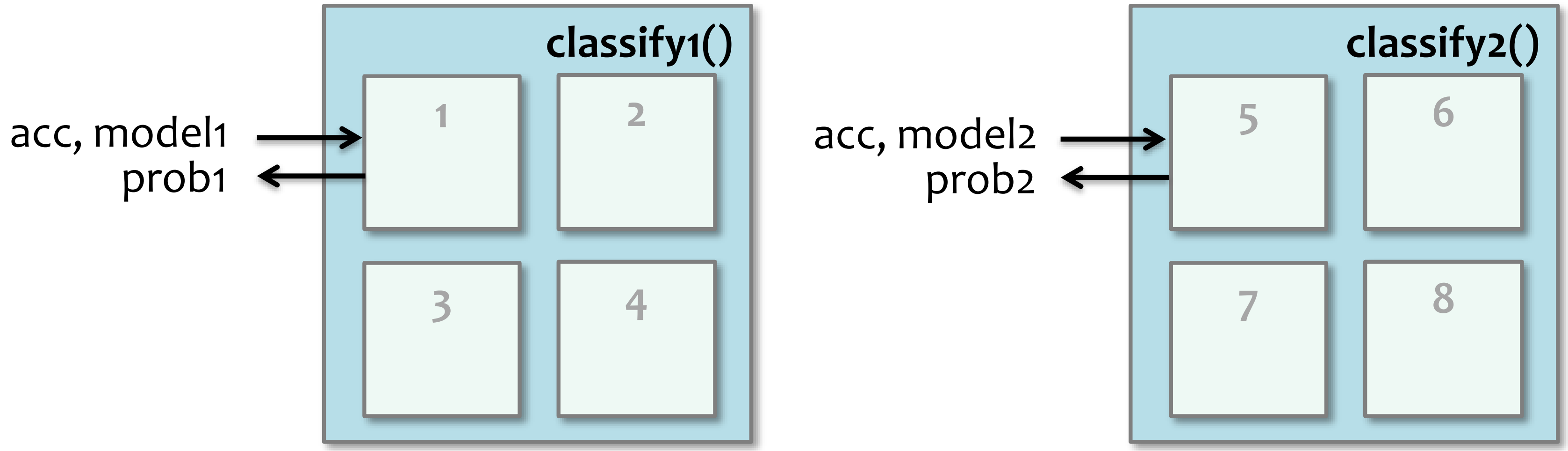
Original: No Task Parallelism

```
fix1_t@1 classify(fix1_t@1 acc[3], fix1_t@2 model[N]) {...}

prob1 = classify(acc, model1);
prob2 = classify(acc, model2);
```



Work Around: Manual Replication

```
fix1_t@1 classify1(fix1_t@1 acc[3], fix1_t@2 model[N]) {...}
fix1_t@5 classify2(fix1_t@5 acc[3], fix1_t@6 model[N]) {...}

prob1 = classify1(acc, model1);
prob2 = classify2(acc, model2);
```



Solution: Automatic Replication

```
// Define module
module Classifier(model_init) {
  fix1_t@1 model[N] = model_init ;
  fix1_t@2 classify(fit1_t@2 acc[3]) {
    ... }
}

// Create module instances
C1 = new Classifier(model1);
C2 = new Classifier(model2);

// Call two different functions
C1.classify(acc);
C2.classify(acc);
```

Desugared to

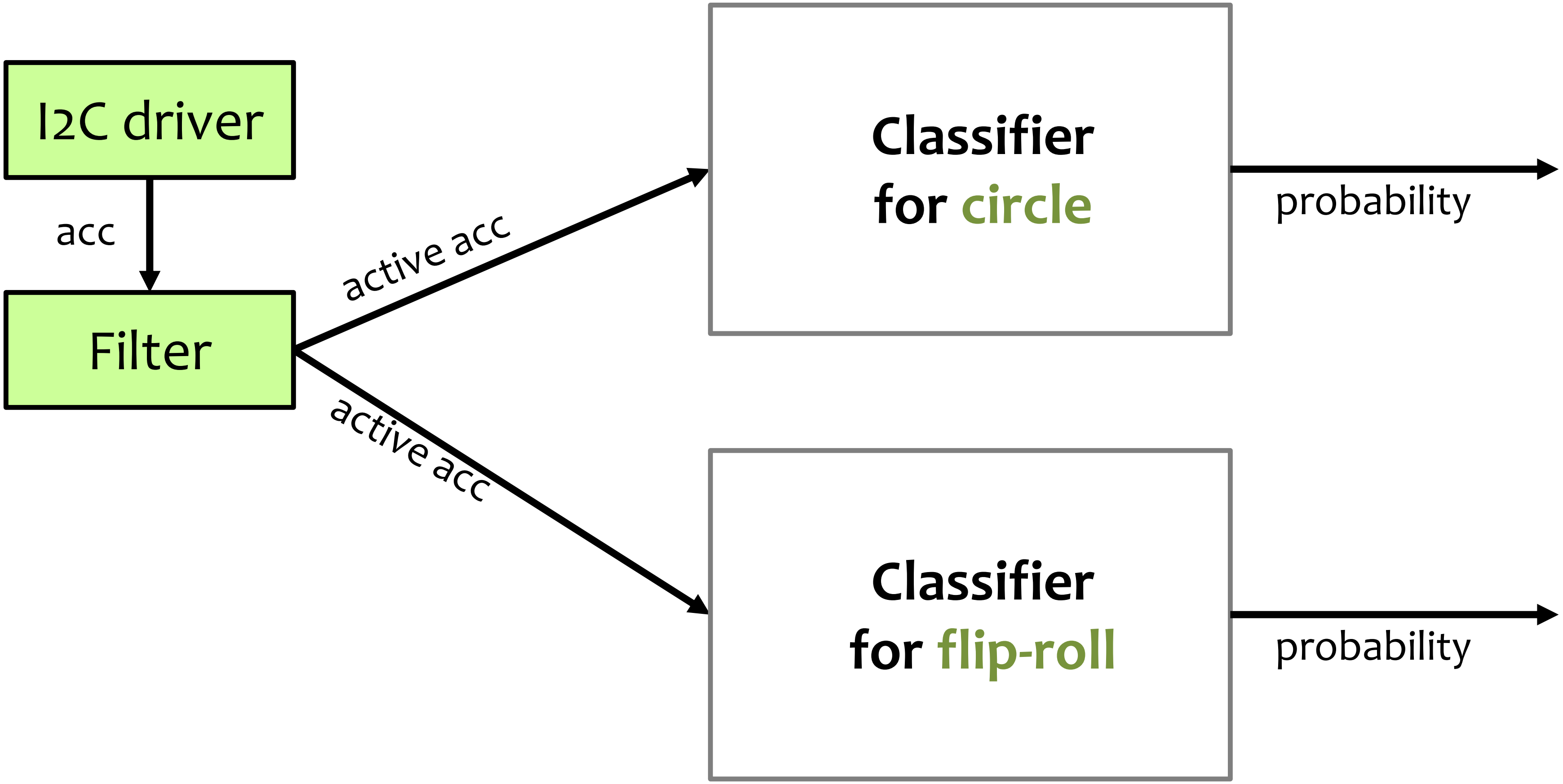
```
// Expanded from module instance 1
fix1_t@1 C1_model[N] = model1 ;
fix1_t@2 C1_classify(fit1_t@2 acc[3]) {
  ... }

// Expanded from module instance 2.
fix1_t@3 C2_model[N] = model2;
fix1_t@4 C2_classify(fit1_t@4 acc[3]) {
  ... }

C1_classify(acc);
C2_classify(acc);
```

Solution Summary

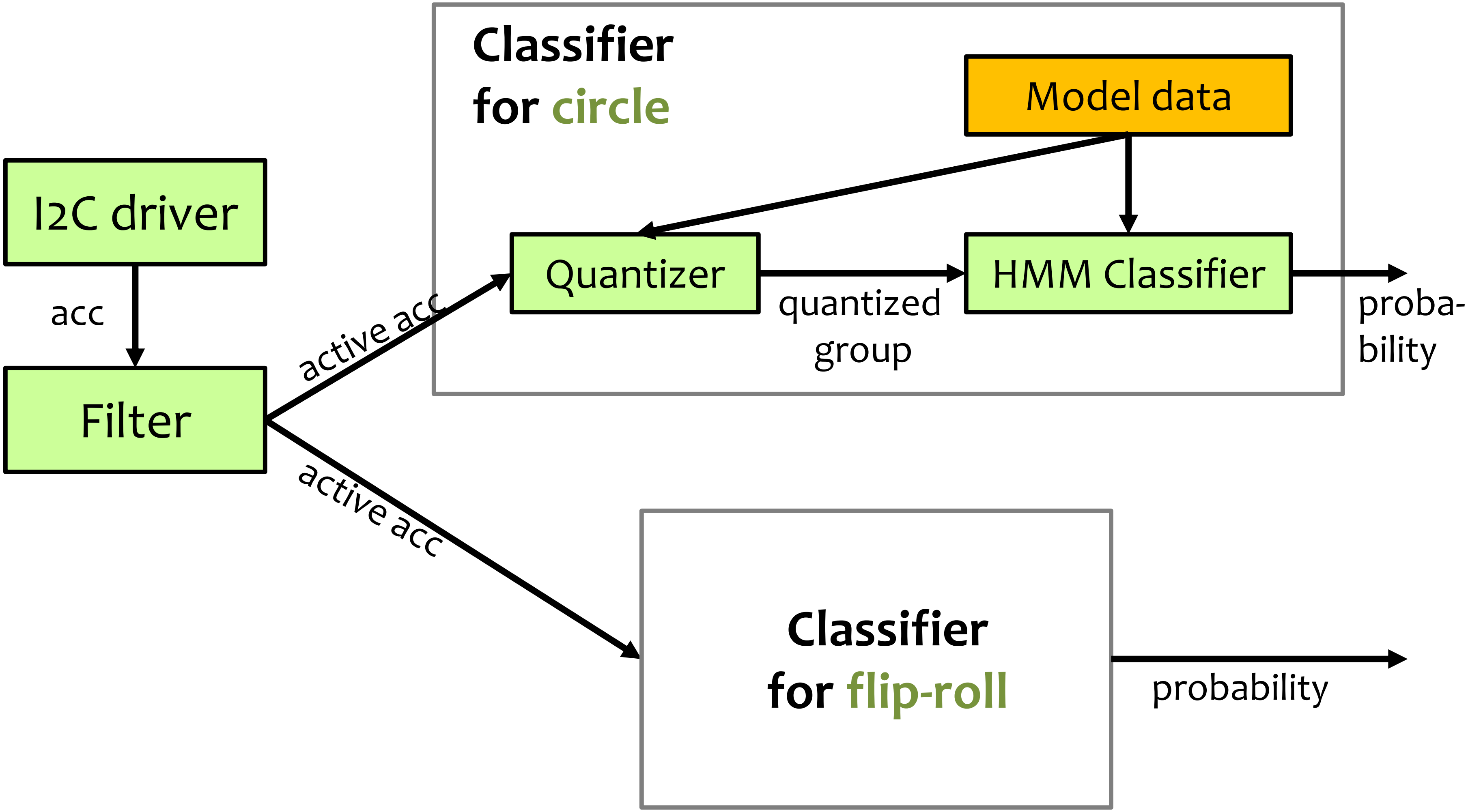
Balance the use of **communication** and **code replication** to partition

- program control flow statements
- data and computation

Hand Gesture Recognition

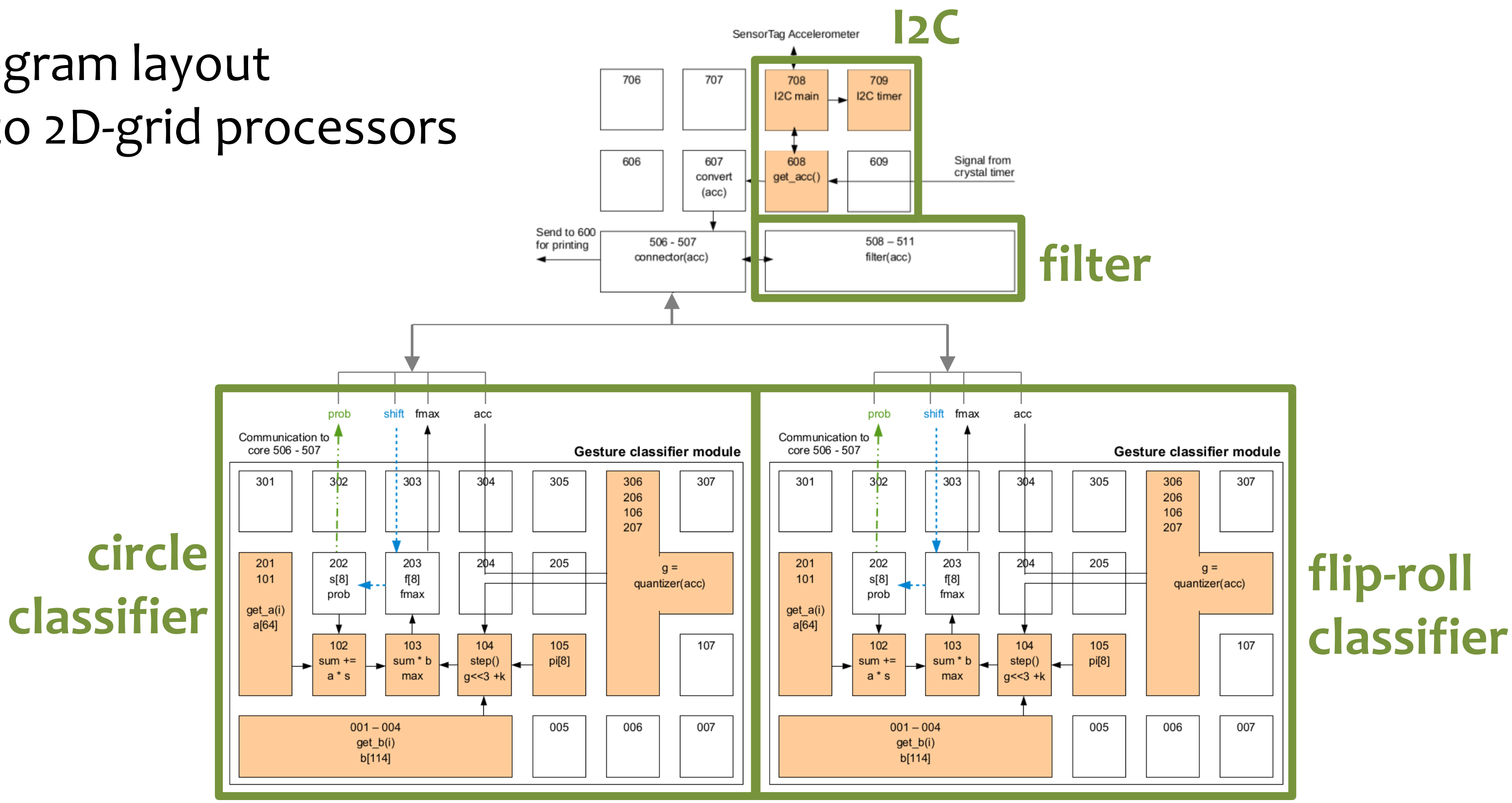


Hand Gesture Recognition



Implementation

Program layout
onto 2D-grid processors



1. Use **mixed partitioning strategy** to make the application fit on GA144.
   orange = actor cores
2. Use **parallel module** to classify circle and flip-roll gestures in parallel.

Result

Can we use Chlorophyll with our extensions to generate code for the gesture recognition application for GA144?

Partitioning strategy	Number of cores	Overflowed cores	Size of largest core (words)
SPMD	90	12	87
SPMD + Actor	82	0	64

Note: each core can contain up to 64 words.

Code occupies 82 out of 144 cores.

Prediction accuracy = 80-91% (similar to Wigee [Schlomer et al. 08])

GA144 vs. MSP430

How much energy consumption can we reduce by being able to compile for GA144?

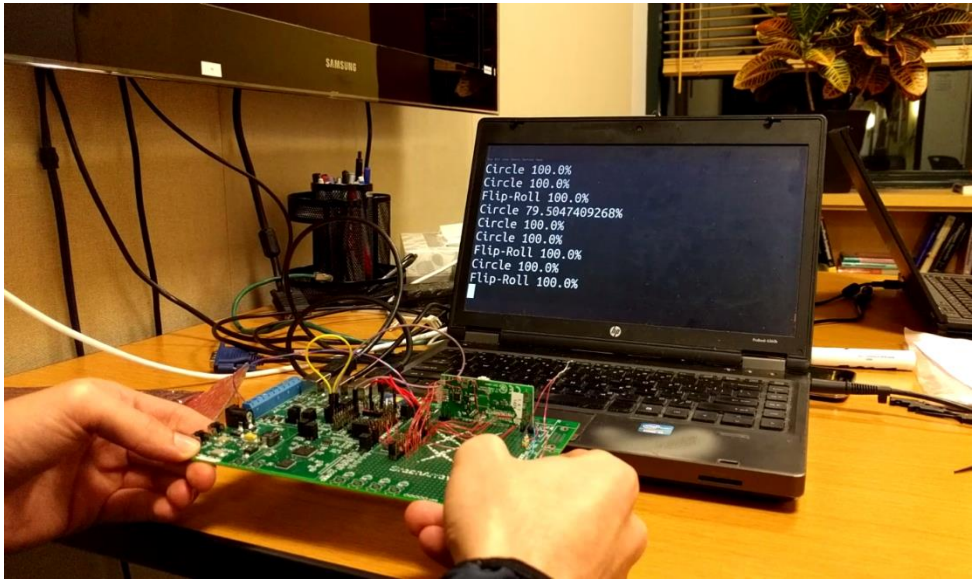
*per one round of accelerometer reading

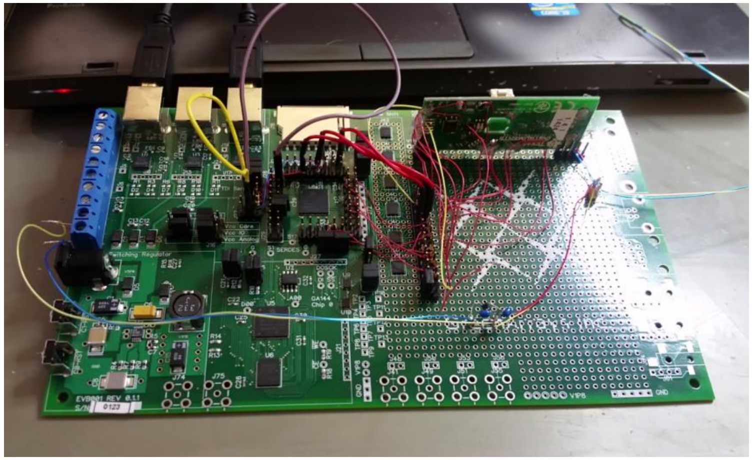
Processor	Execution time (ms)	Energy consumption (uJ)		
		Accelerometer	Computation	Total
GA144	2.6	1.7	0.6	2.2
MSP430	61.3	0.8	41.2	41.9

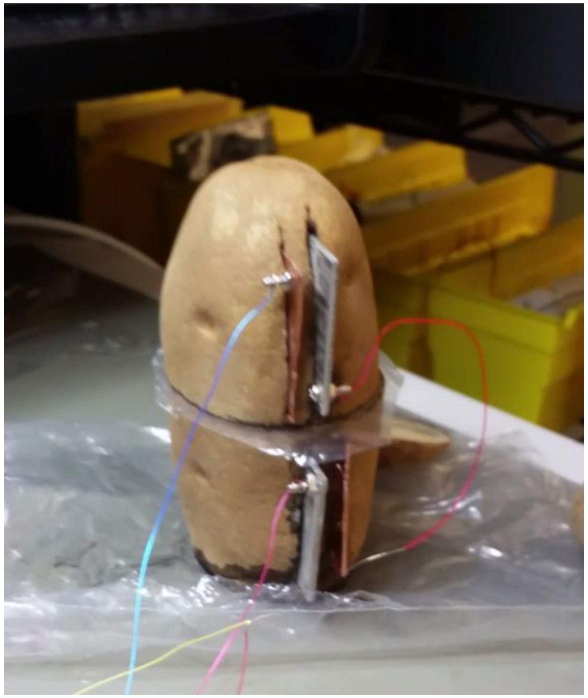
23x faster

19x more energy-efficient

Demo

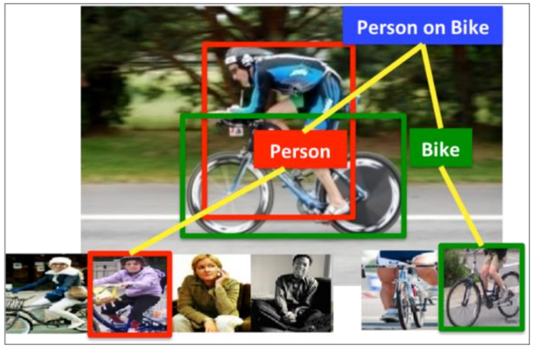

Circle 100.0%
Circle 100.0%
Flip-Roll 100.0%
Circle 79.5047409268%
Circle 100.0%
Circle 100.0%
Flip-Roll 100.0%
Circle 100.0%
Flip-Roll 100.0%

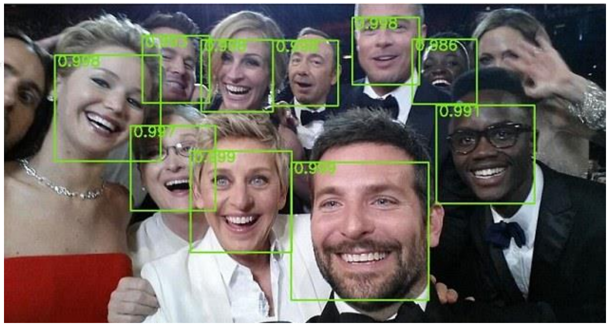




Summary

Partition a program smartly to fit the code in tiny cores and achieve fast execution time by balancing communication and code replication.





0.5 mW!