An Architectural Framework for Accelerating Dynamic Parallel Algorithms on Reconfigurable Hardware

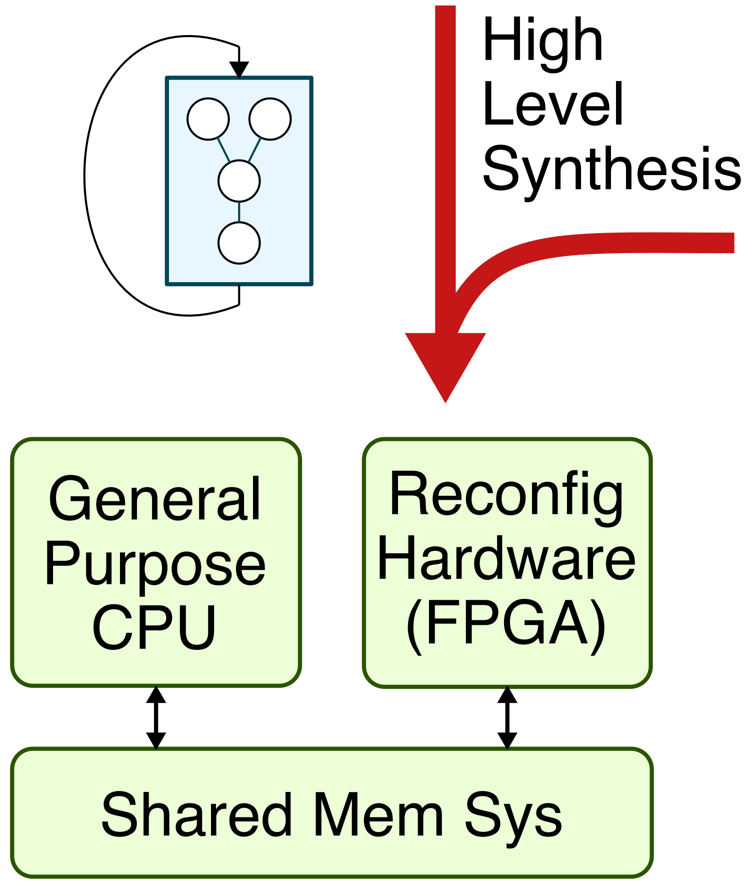
Tao Chen, Shreesha Srinath
Christopher Batten, G. Edward Suh

Computer Systems Laboratory
School of Electrical and Computer Engineering
Cornell University

51st Int'l Symp. on Microarchitecture
Fall 2018

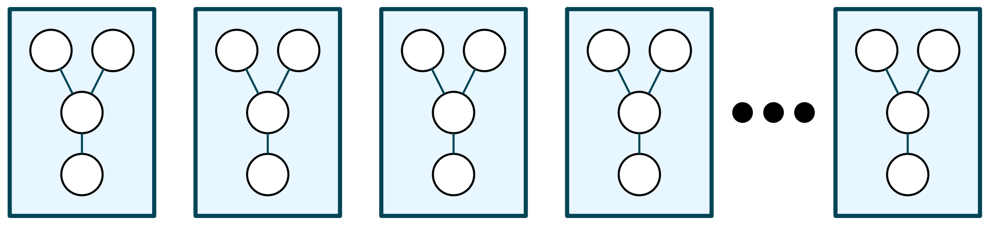
Accelerating *Static* Parallel Algorithms on Reconfigurable Hardware

```
for (int i=0; i<n; i++)
  c[i] = a[i] + b[i];
```



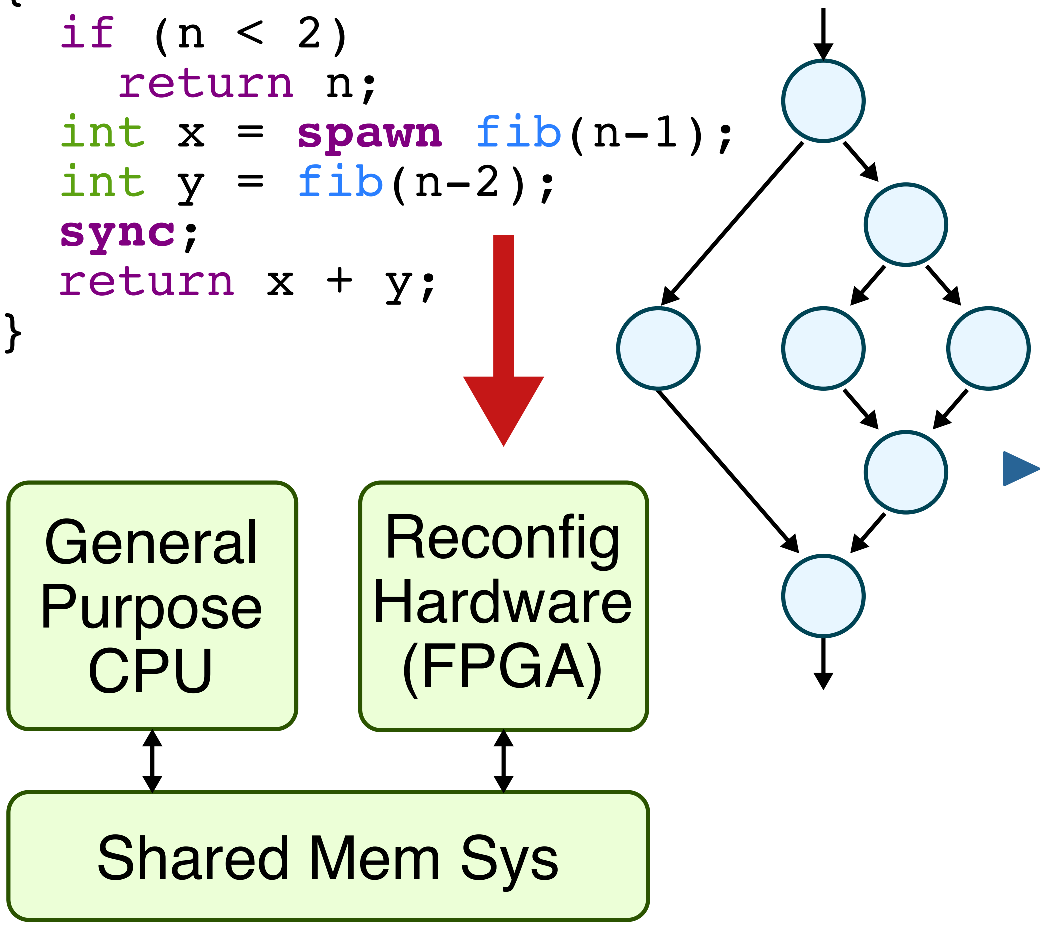
- Emerging CPU+FPGA platforms (Xilinx Zynq, Altera Cyclone SoC)
- HLS maps parallelism statically to highly pipelined *and parallel* PEs

```
__kernel
void vvadd( __global int* c,
            __global int* a,
            __global int* b, int n )
{
  int id = get_global_id(0);
  if ( id < n )
    c[id] = a[id] + b[id];
}
```



Programmers are increasingly moving from thread- to task-centric programming

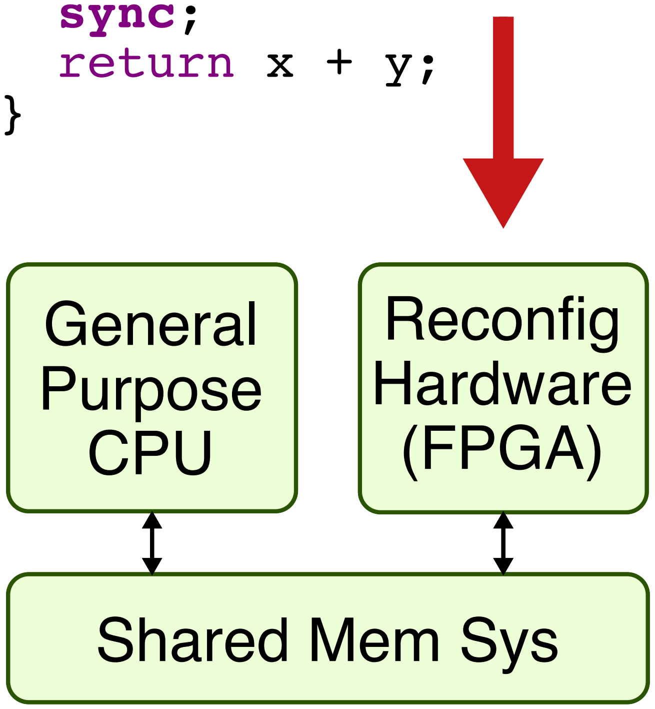
```
int fib( int n )
{
  if (n < 2)
    return n;
  int x = spawn fib(n-1);
  int y = fib(n-2);
  sync;
  return x + y;
}
```



- **Task-parallel programming frameworks** enable creating tasks dynamically as the program executes
  - Intel Cilk Plus, Intel C++ TBB, Microsoft's .NET TPL, Java's Fork/Join, OpenMP
- Benefits of this approach:
  - hierarchical data structures
  - divide-and-conquer algos
  - adaptive algorithms
  - arbitrary nesting, composition
  - automatic load balancing
  - efficient in theory and practice

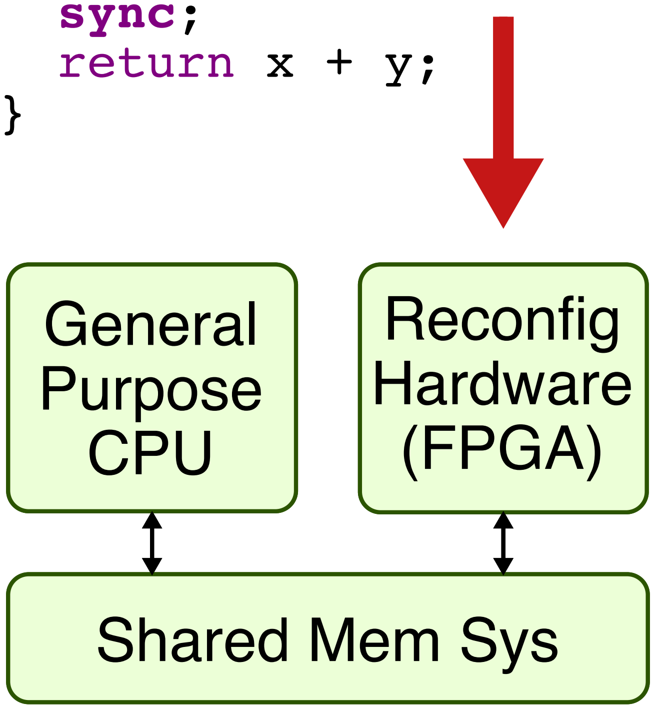
Accelerating *Dynamic* Parallel Algorithms on Reconfigurable Hardware

```
int fib( int n )
{
  if (n < 2)
    return n;
  int x = spawn fib(n-1);
  int y = fib(n-2);
  sync;
  return x + y;
}
```

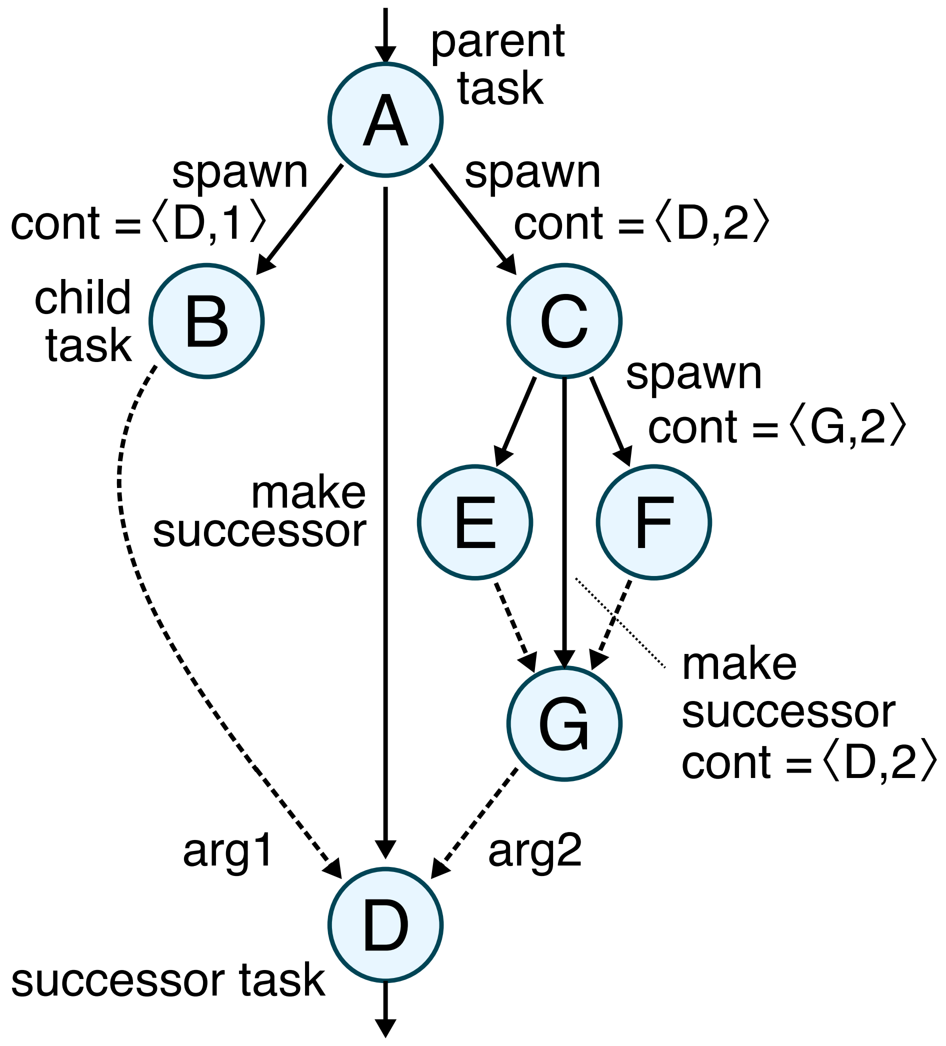


Accelerating *Dynamic* Parallel Algorithms on Reconfigurable Hardware

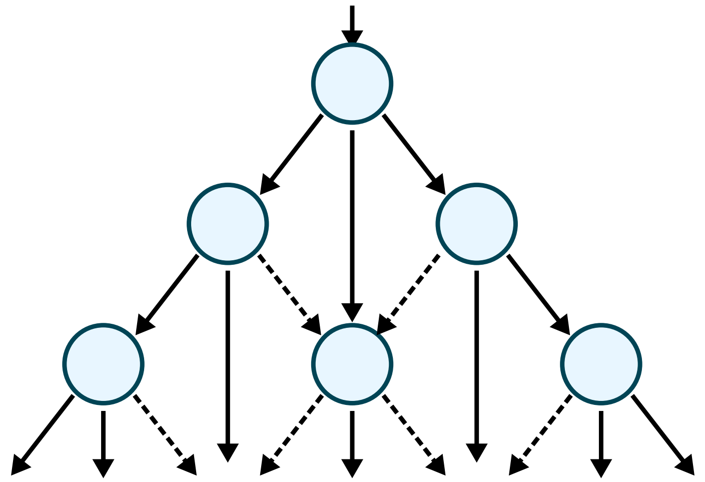
```
int fib( int n )
{
  if (n < 2)
    return n;
  int x = spawn fib(n-1);
  int y = fib(n-2);
  sync;
  return x + y;
}
```



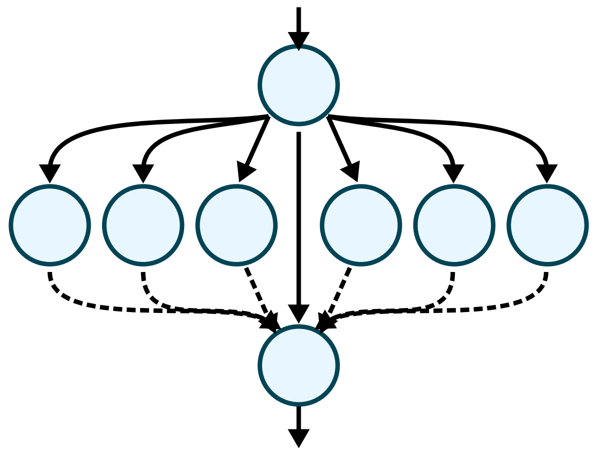
Explicit Continuation Passing



Fork/Join Pattern



Data-Flow Pattern



Data-Parallel Pattern

Example of Explicit Continuation Passing w/ Cilk

```
int fib( int n )
{
  if (n < 2)
    return n;
  int x = spawn fib(n-1);
  int y = fib(n-2);
  sync;
  return x + y;
}
```

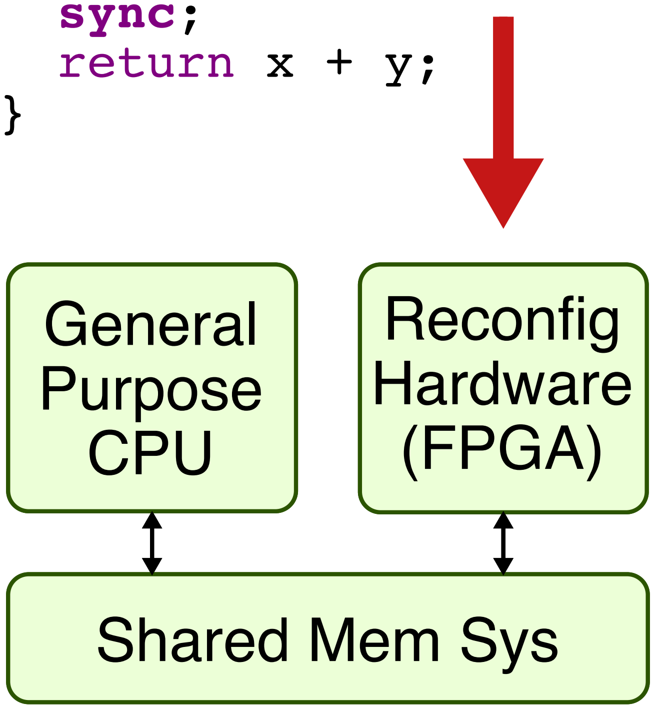
```
task fib( cont int k, int n )
{
  if ( n < 2 )
    send_argument( k, n );
  else {
    cont int x, y;
    spawn_next sum( k, ?x, ?y );
    spawn      fib( x, n-1 );
    spawn      fib( y, n-2 );
  }
}

task sum( cont int k, int x, int y )
{
  send_argument( k, x+y );
}
```

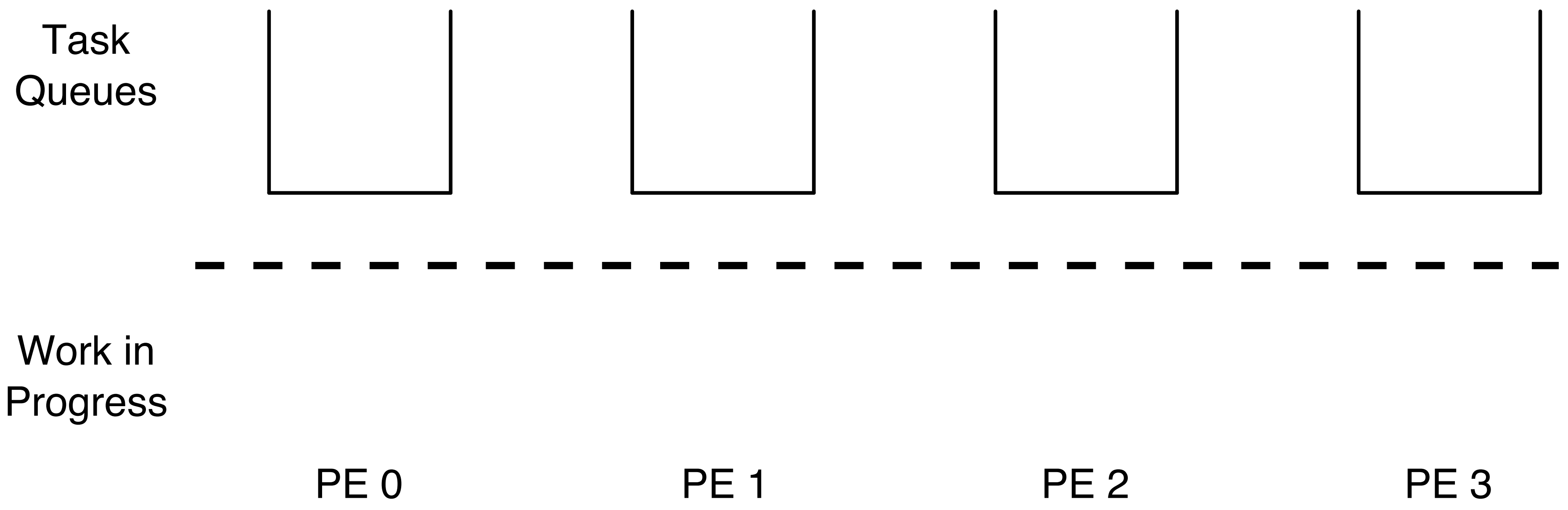
- ▶ Cilk-1 used explicit continuation passing (JPDC'96)
- ▶ Cilk-5 used call/return semantics for parallelism (PLDI'98)
- ▶ Explicit continuation passing is an elegant match for hardware

Accelerating *Dynamic* Parallel Algorithms on Reconfigurable Hardware

```
int fib( int n )
{
  if (n < 2)
    return n;
  int x = spawn fib(n-1);
  int y = fib(n-2);
  sync;
  return x + y;
}
```

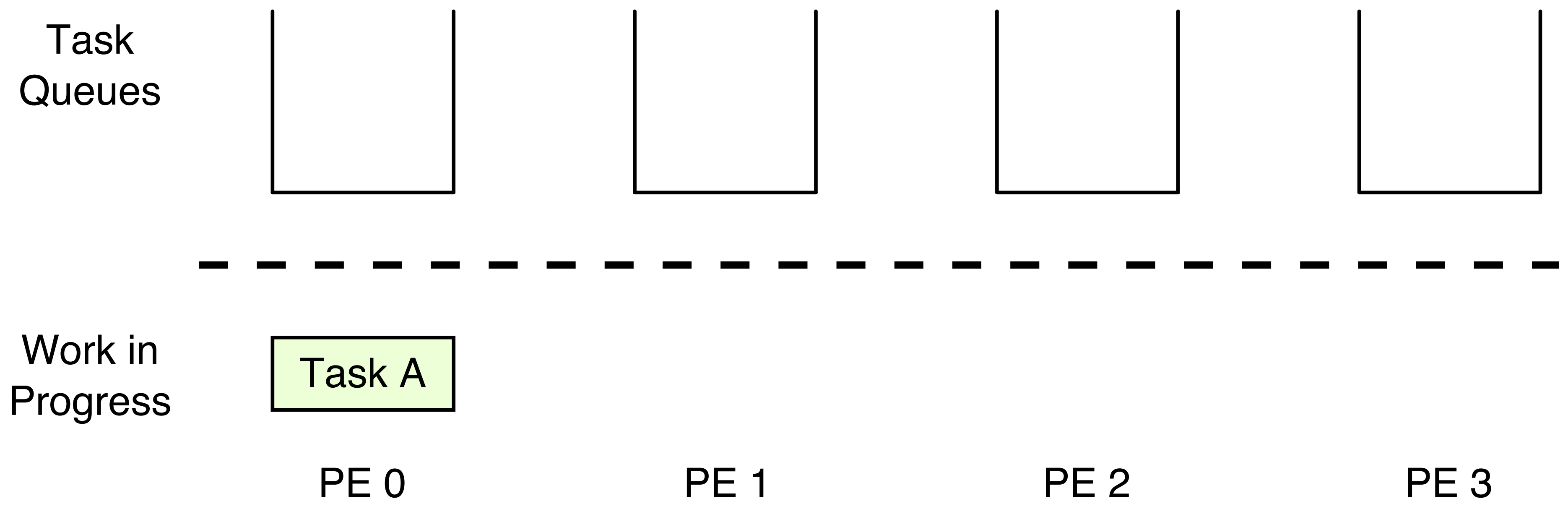


Scheduling Tasks with Work Stealing



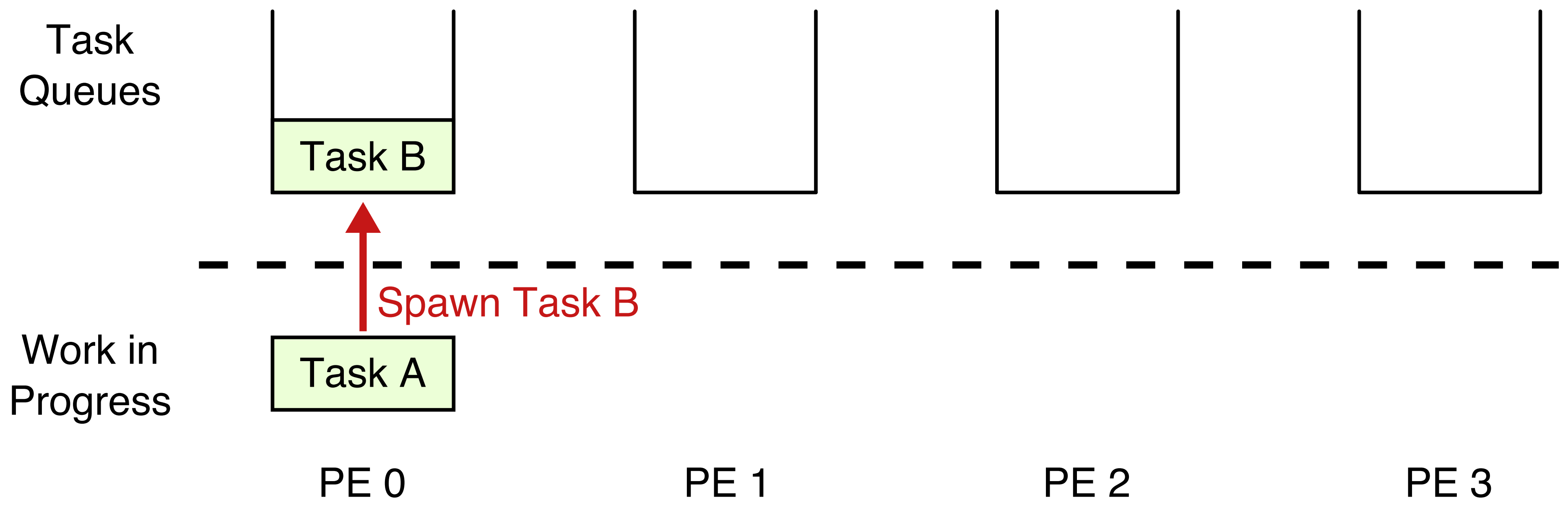
- Work stealing has good performance, space requirements, and communication overheads in both theory and practice

Scheduling Tasks with Work Stealing



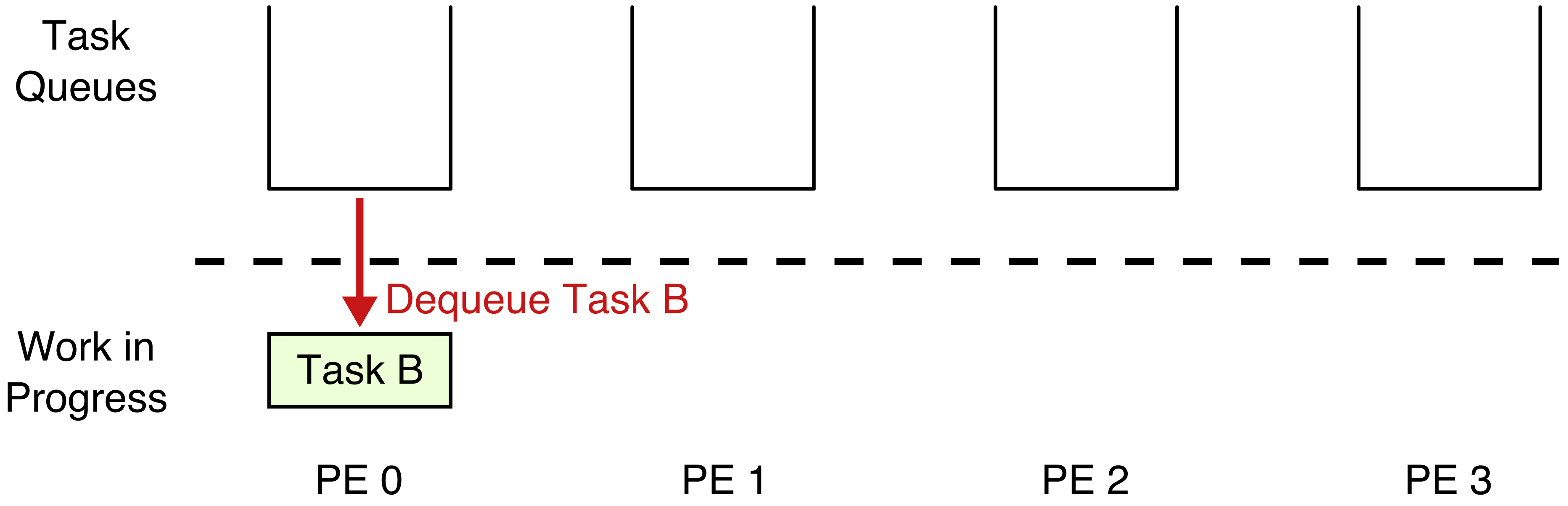
- Work stealing has good performance, space requirements, and communication overheads in both theory and practice

Scheduling Tasks with Work Stealing



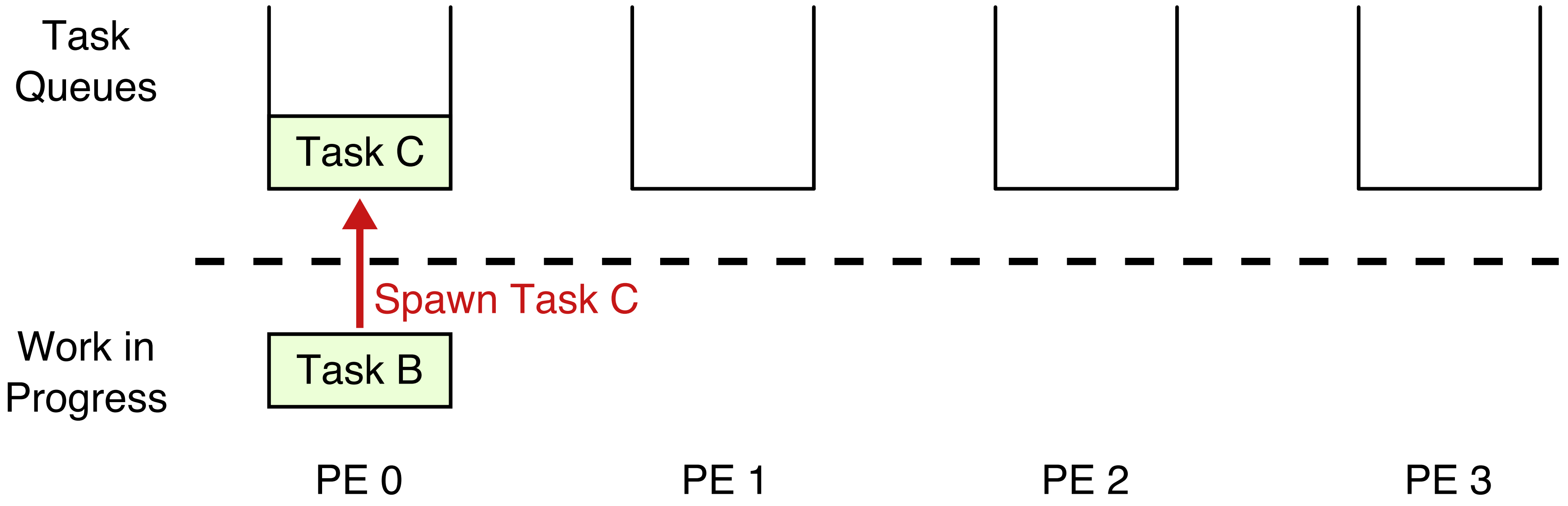
- Work stealing has good performance, space requirements, and communication overheads in both theory and practice

Scheduling Tasks with Work Stealing



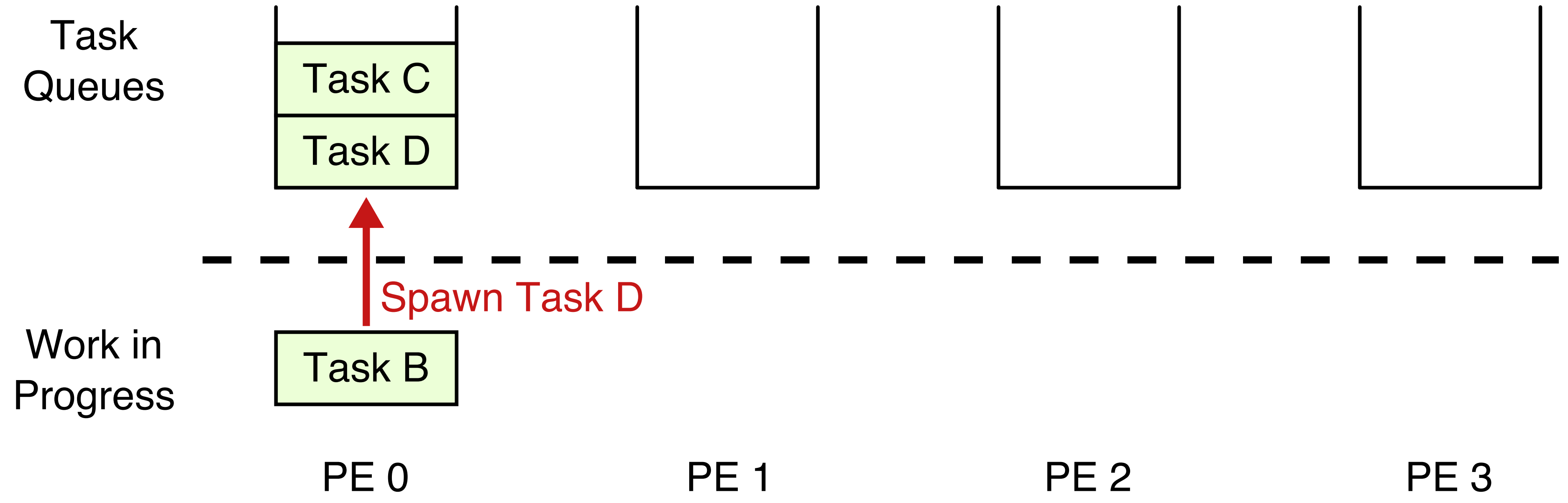
- Work stealing has good performance, space requirements, and communication overheads in both theory and practice

Scheduling Tasks with Work Stealing



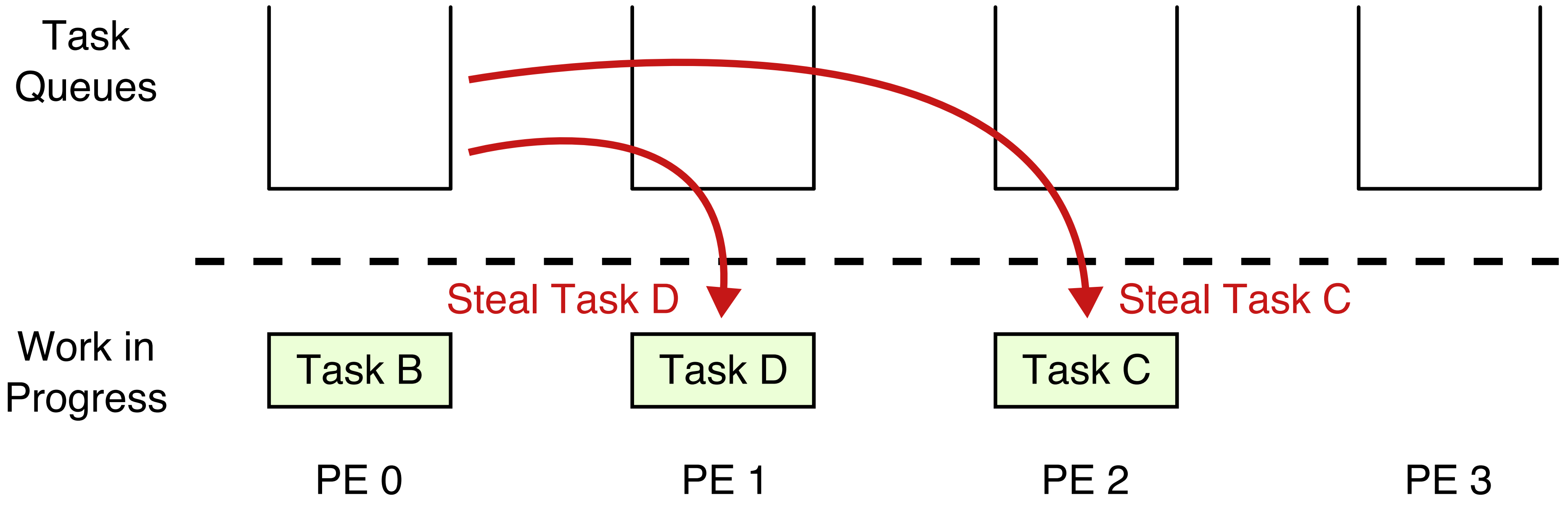
- Work stealing has good performance, space requirements, and communication overheads in both theory and practice

Scheduling Tasks with Work Stealing



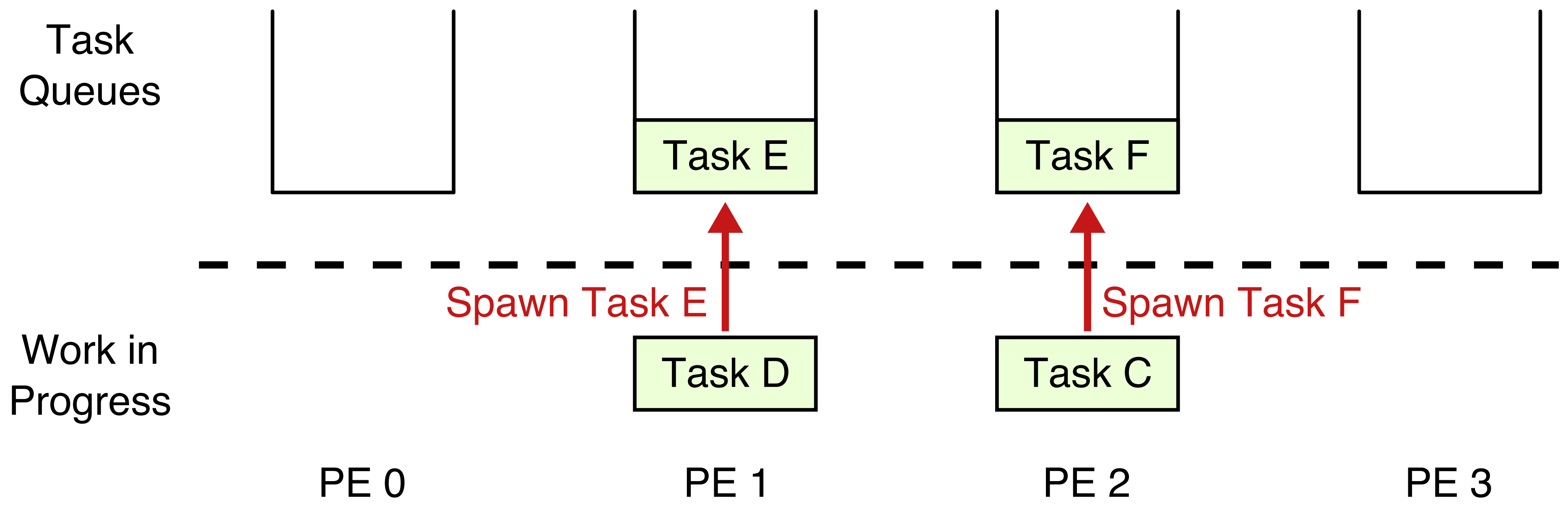
- Work stealing has good performance, space requirements, and communication overheads in both theory and practice

Scheduling Tasks with Work Stealing



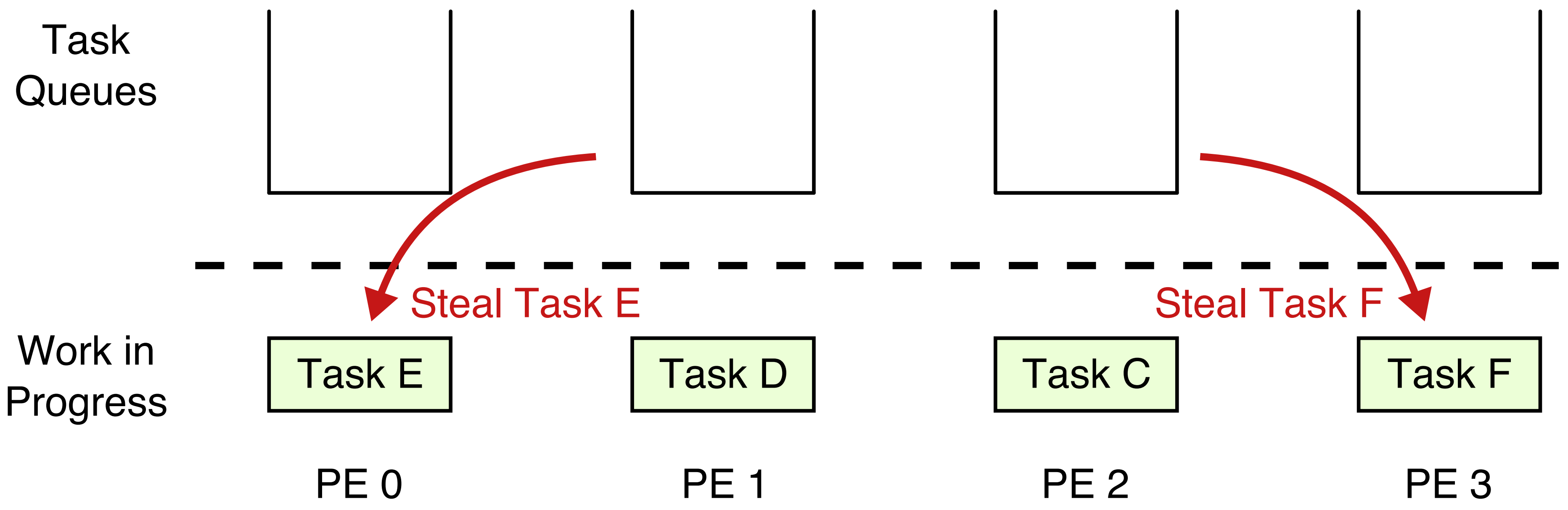
- Work stealing has good performance, space requirements, and communication overheads in both theory and practice

Scheduling Tasks with Work Stealing



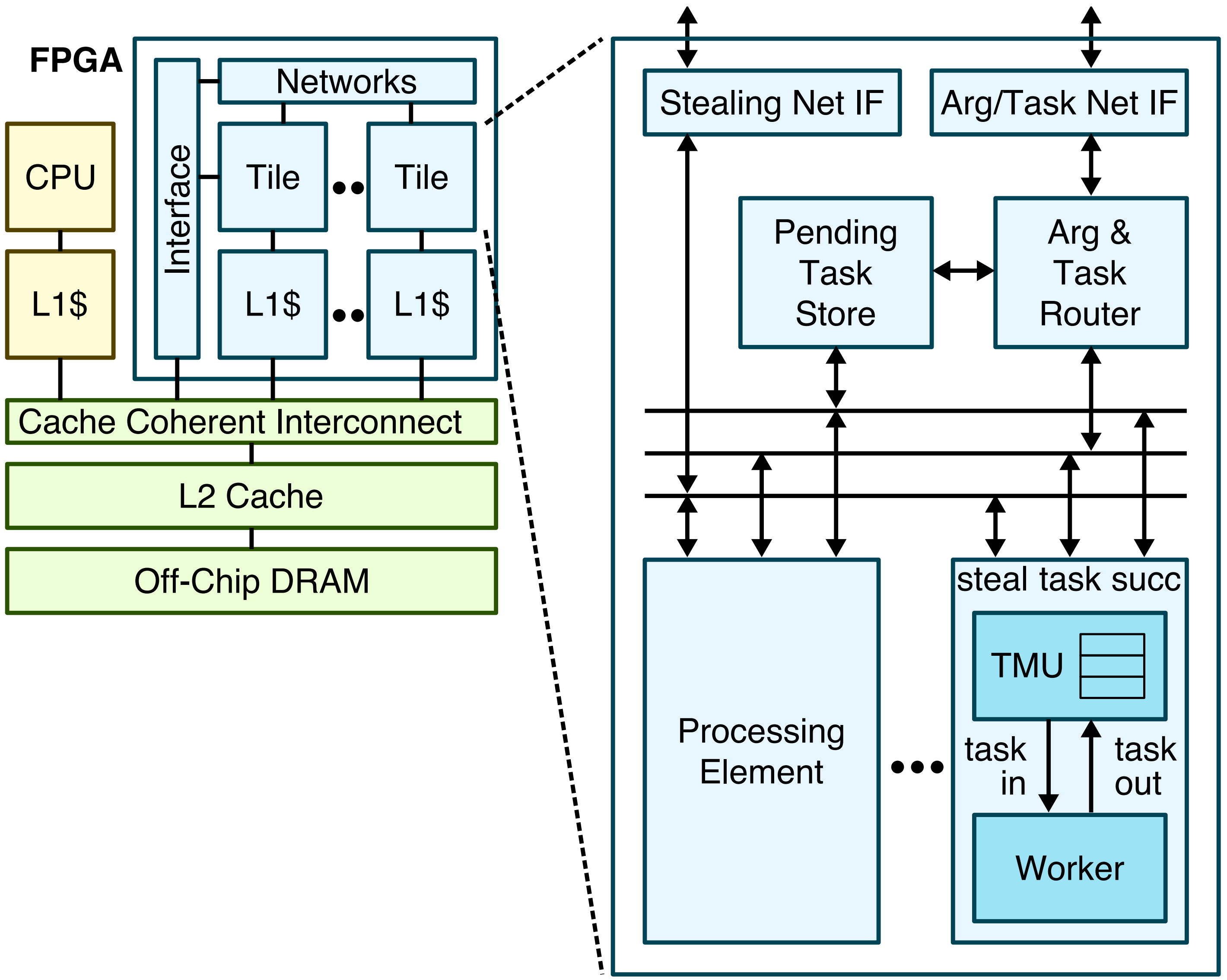
- Work stealing has good performance, space requirements, and communication overheads in both theory and practice

Scheduling Tasks with Work Stealing

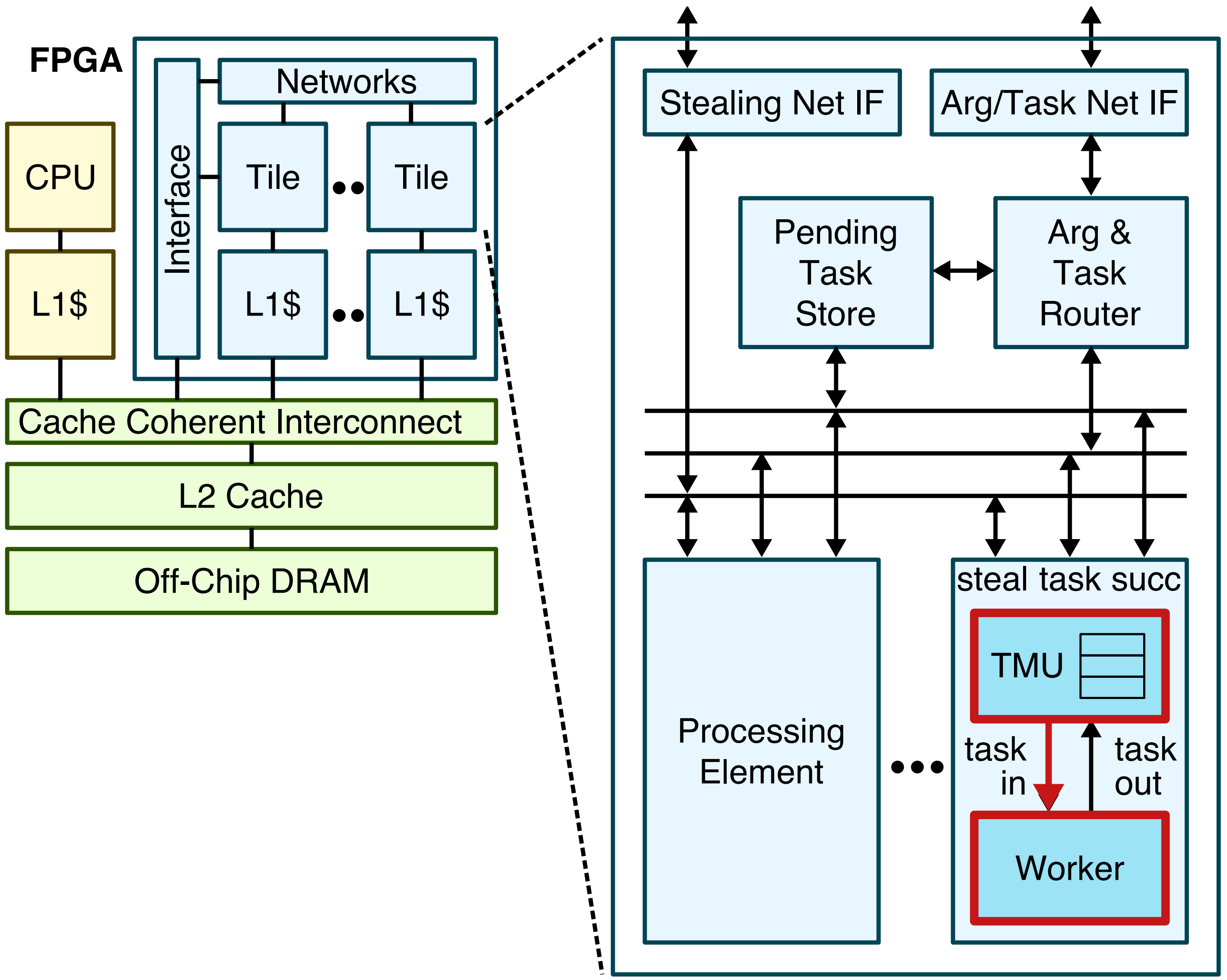


- Work stealing has good performance, space requirements, and communication overheads in both theory and practice

"Flexible" Architectural Template

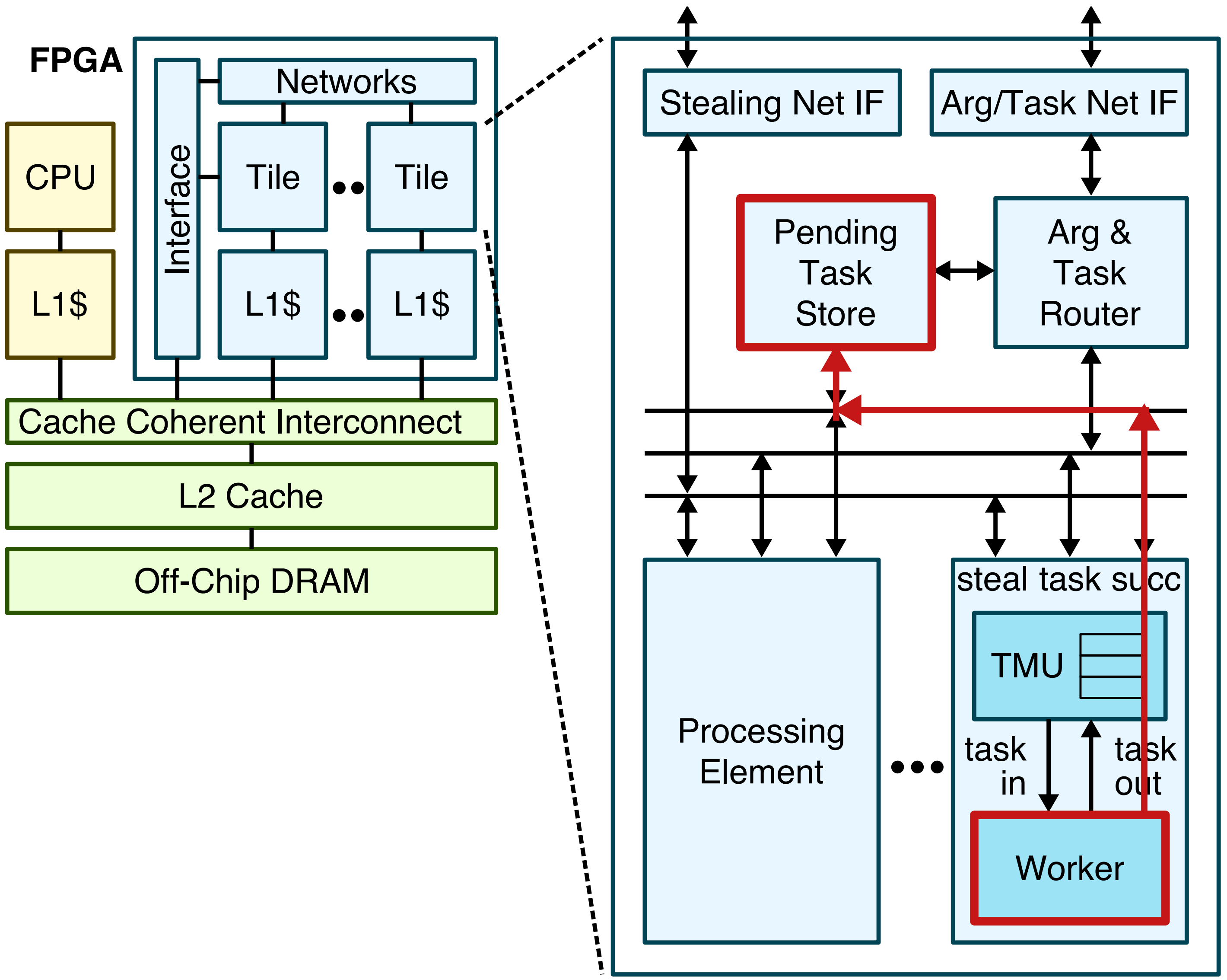


“Flexible” Architectural Template



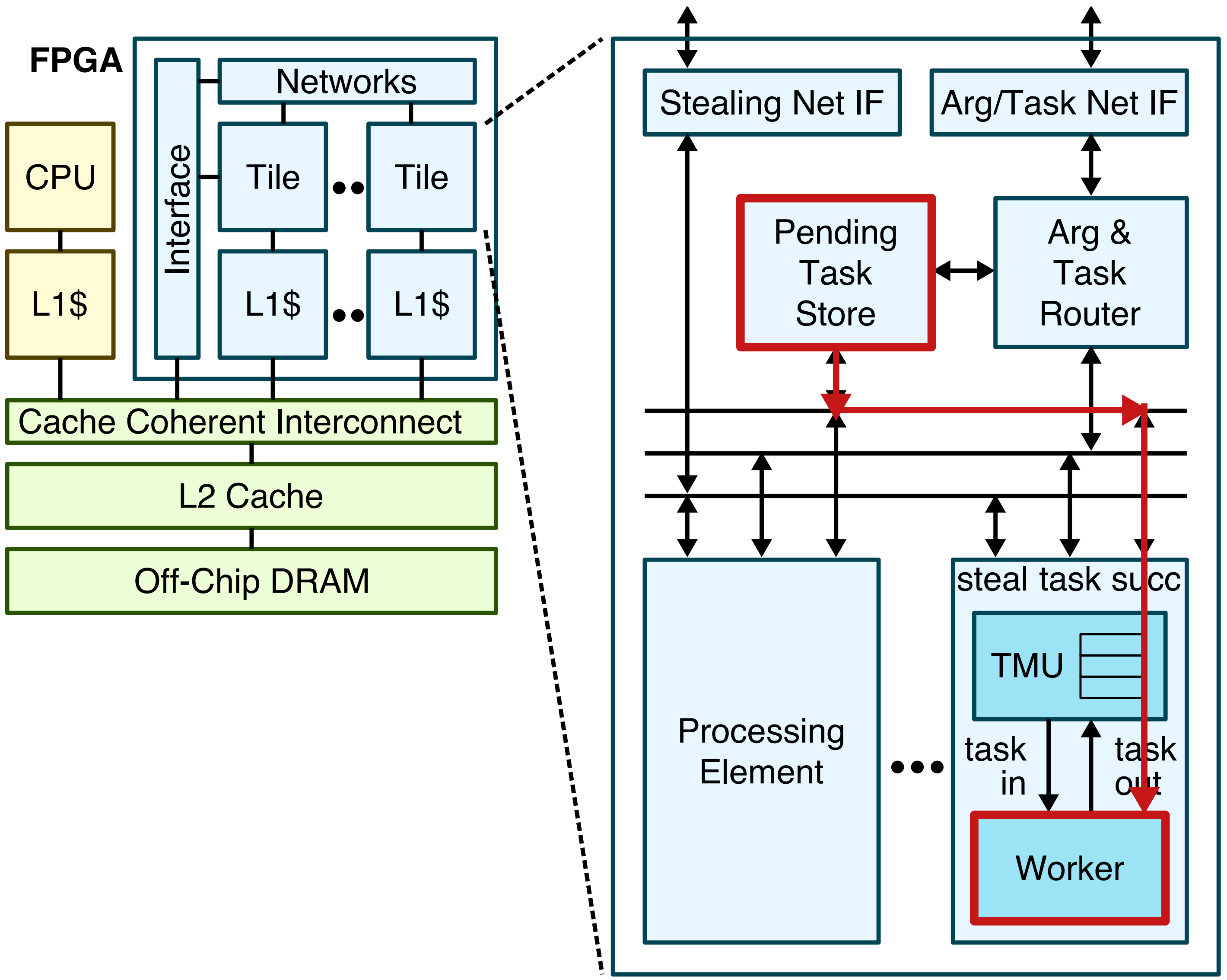
TMU dequeues task and sends to worker

"Flexible" Architectural Template



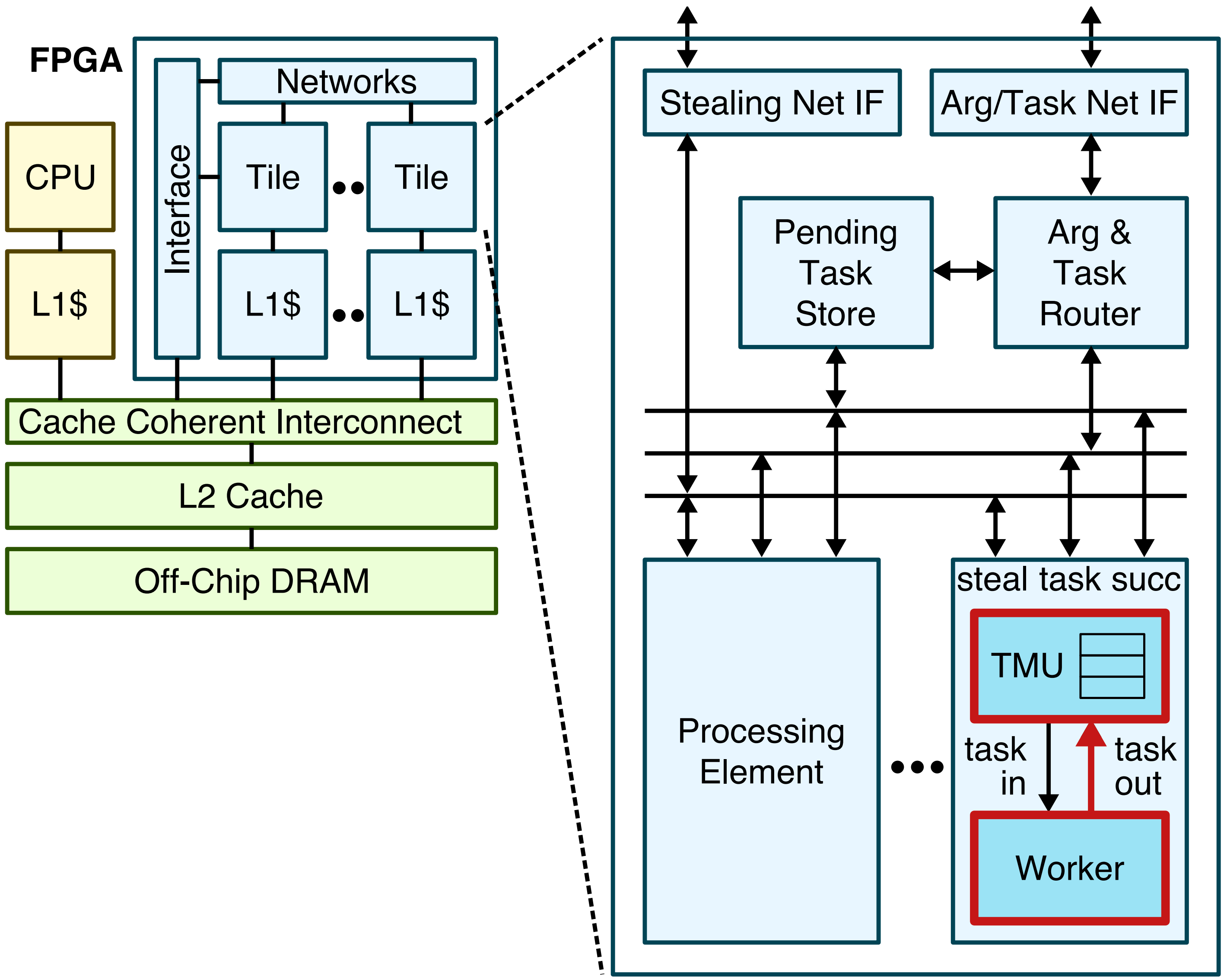
Worker sends request to pending task store to create a successor

“Flexible” Architectural Template



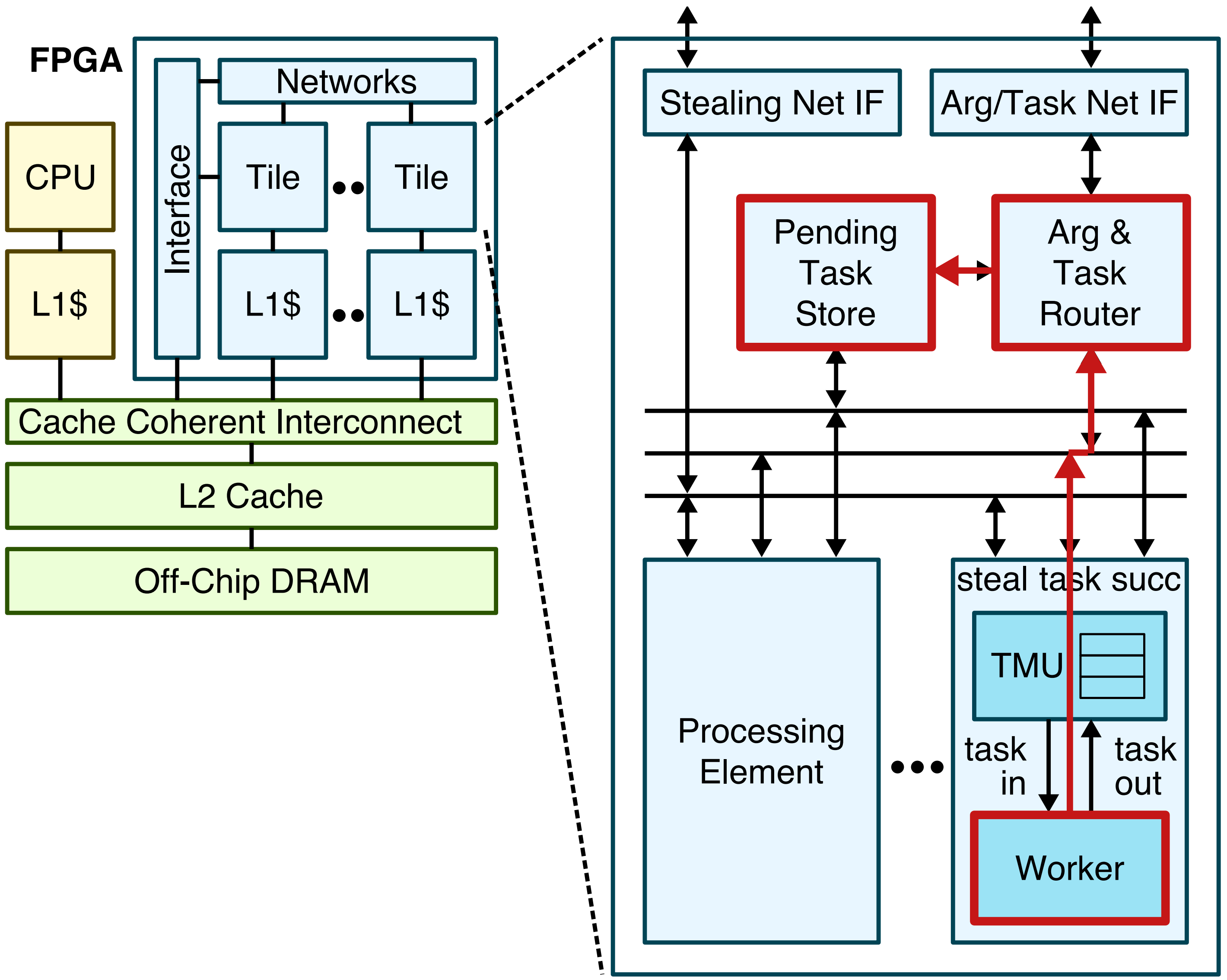
Pending task store sends worker response with ID for creating continuations

“Flexible” Architectural Template



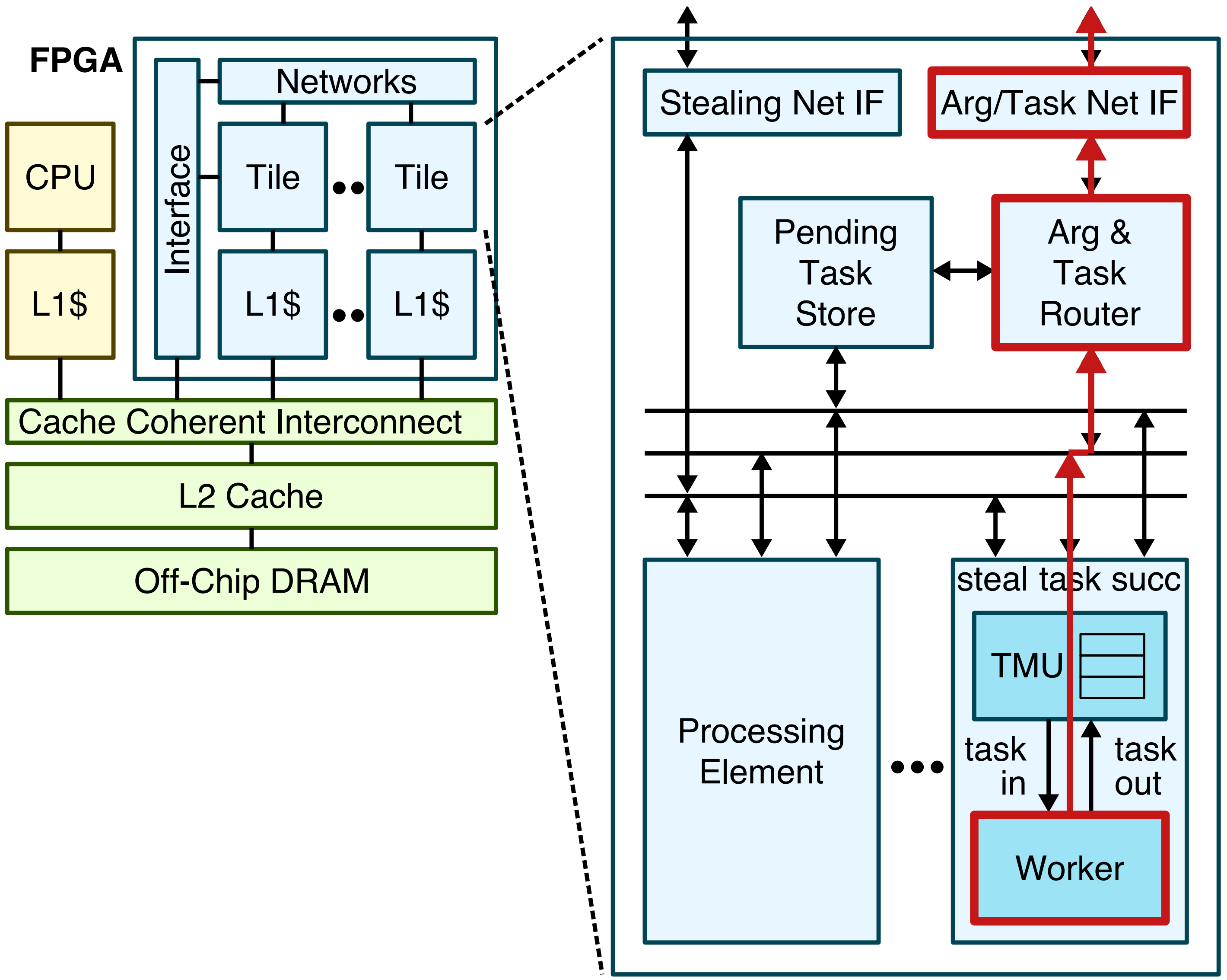
Worker sends spawed tasks to TMU

“Flexible” Architectural Template



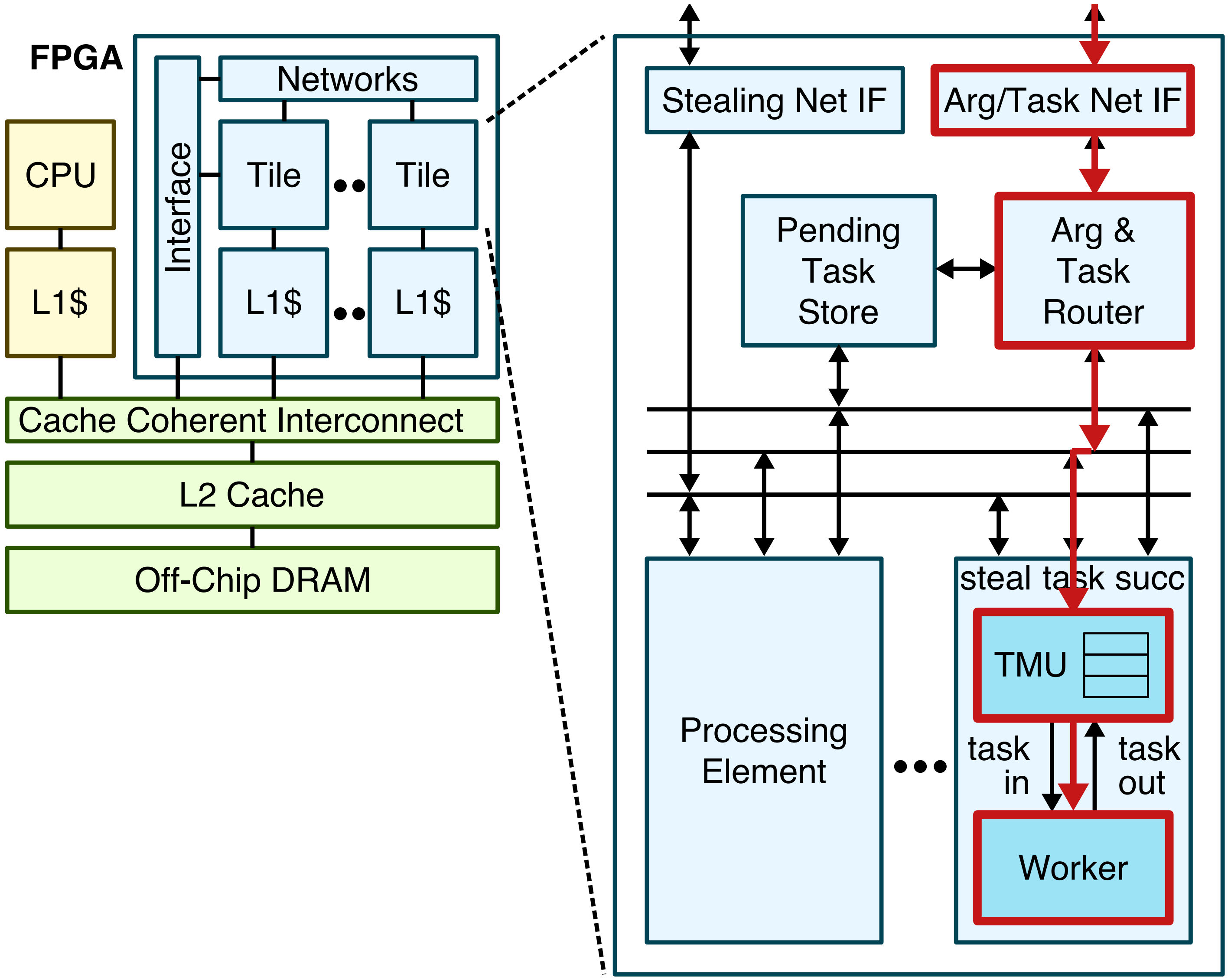
Worker sends return value to arg/task router

"Flexible" Architectural Template



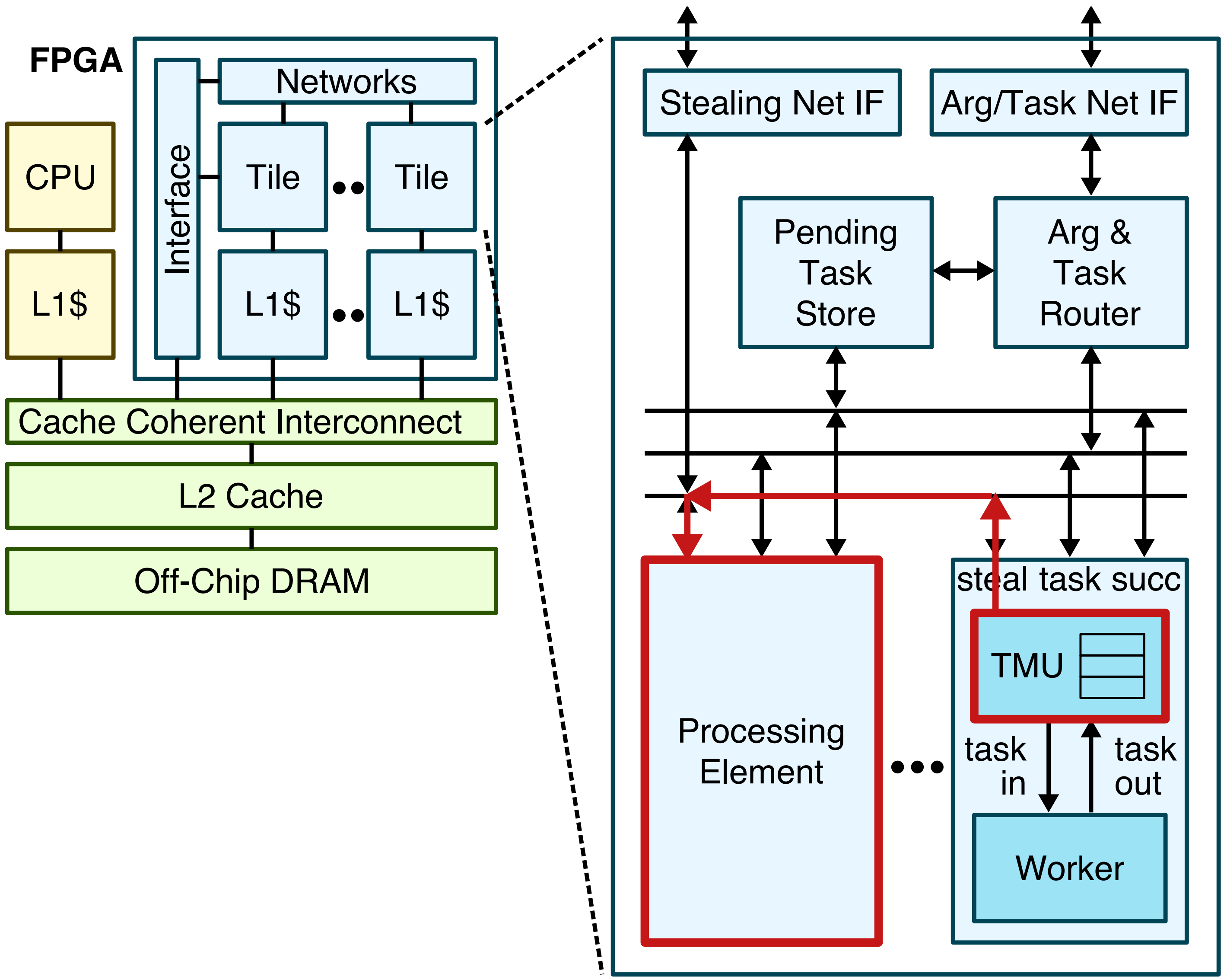
Worker sends return value to arg/task router

“Flexible” Architectural Template



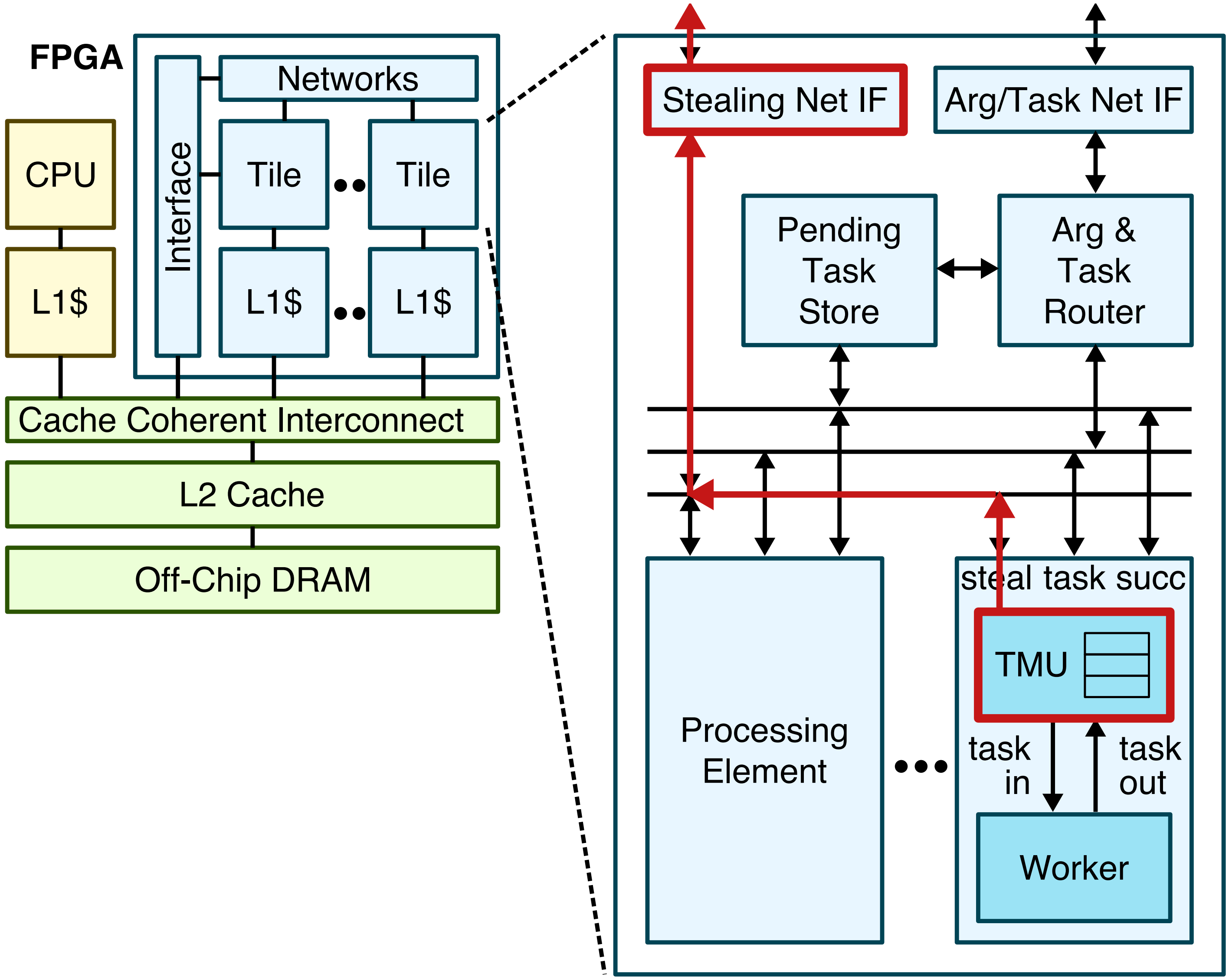
If this is the final argument pending task store sends now ready task back to TMU

“Flexible” Architectural Template



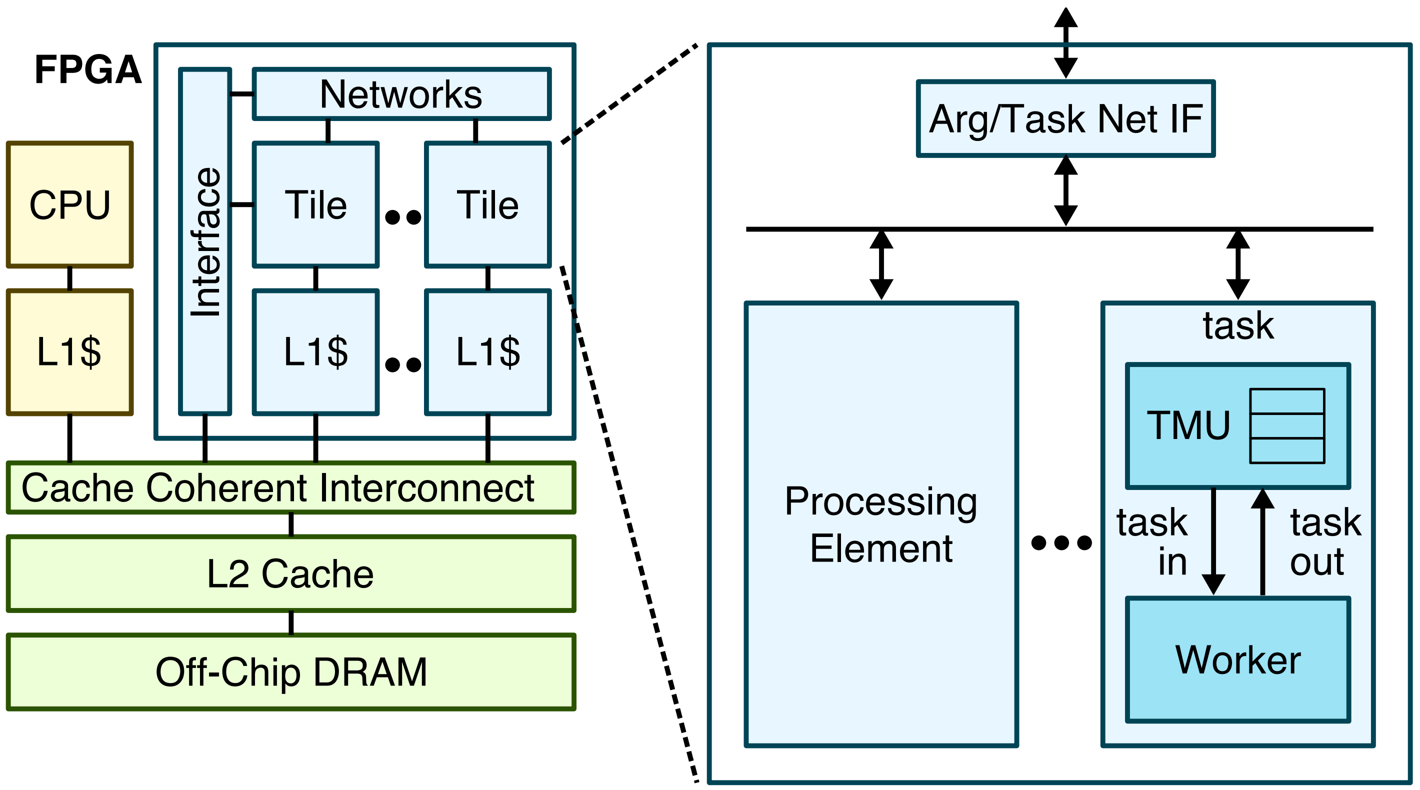
If task queue is empty, TMU randomly selects victim to steal from

“Flexible” Architectural Template



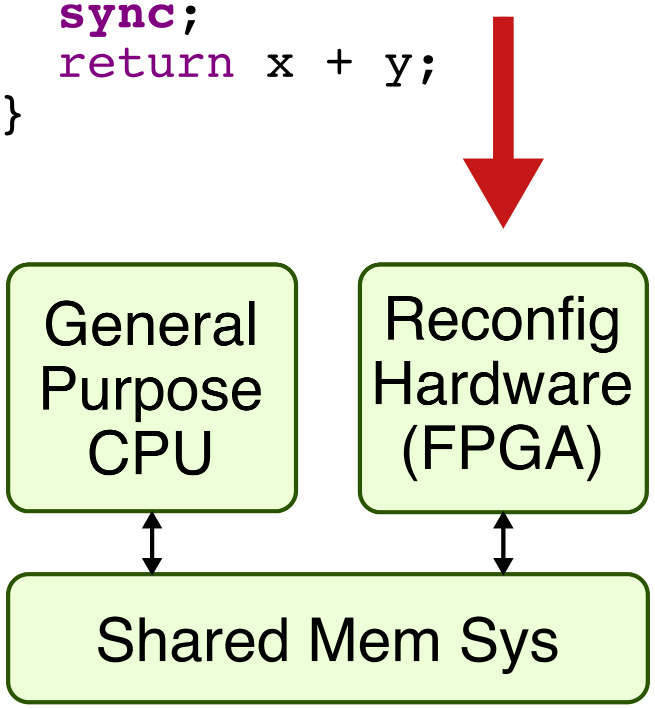
If task queue is empty, TMU randomly selects victim to steal from

"Lite" Architectural Template

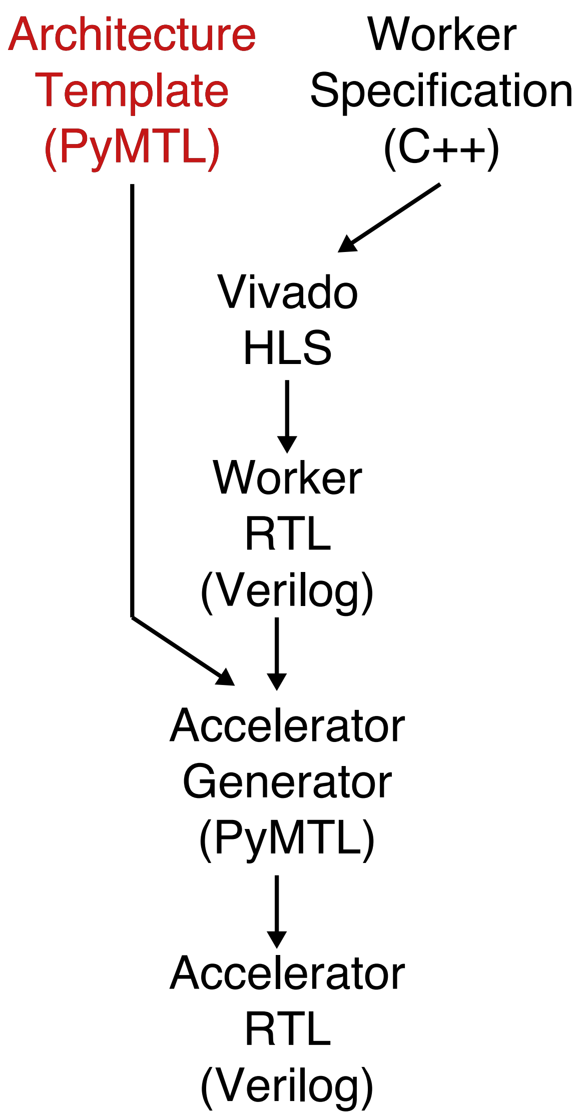


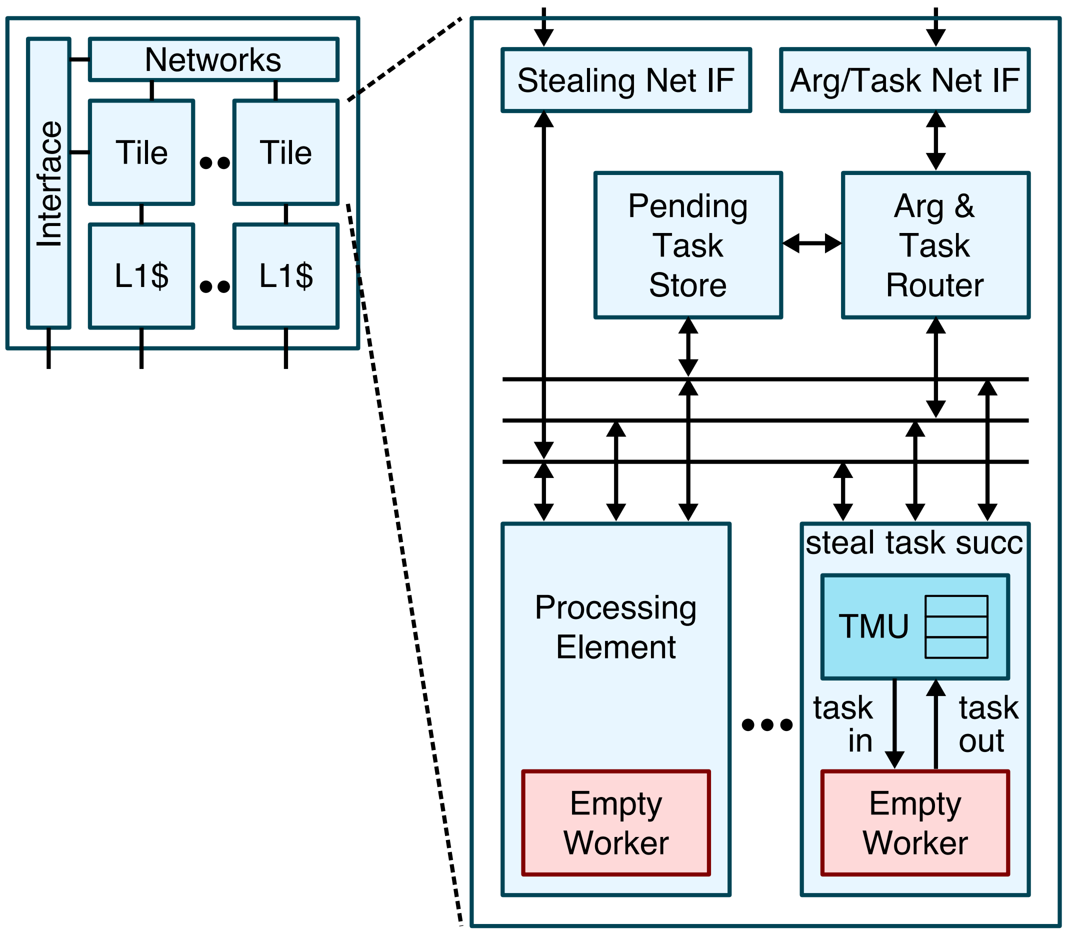
Accelerating *Dynamic* Parallel Algorithms on Reconfigurable Hardware

```
int fib( int n )
{
  if (n < 2)
    return n;
  int x = spawn fib(n-1);
  int y = fib(n-2);
  sync;
  return x + y;
}
```

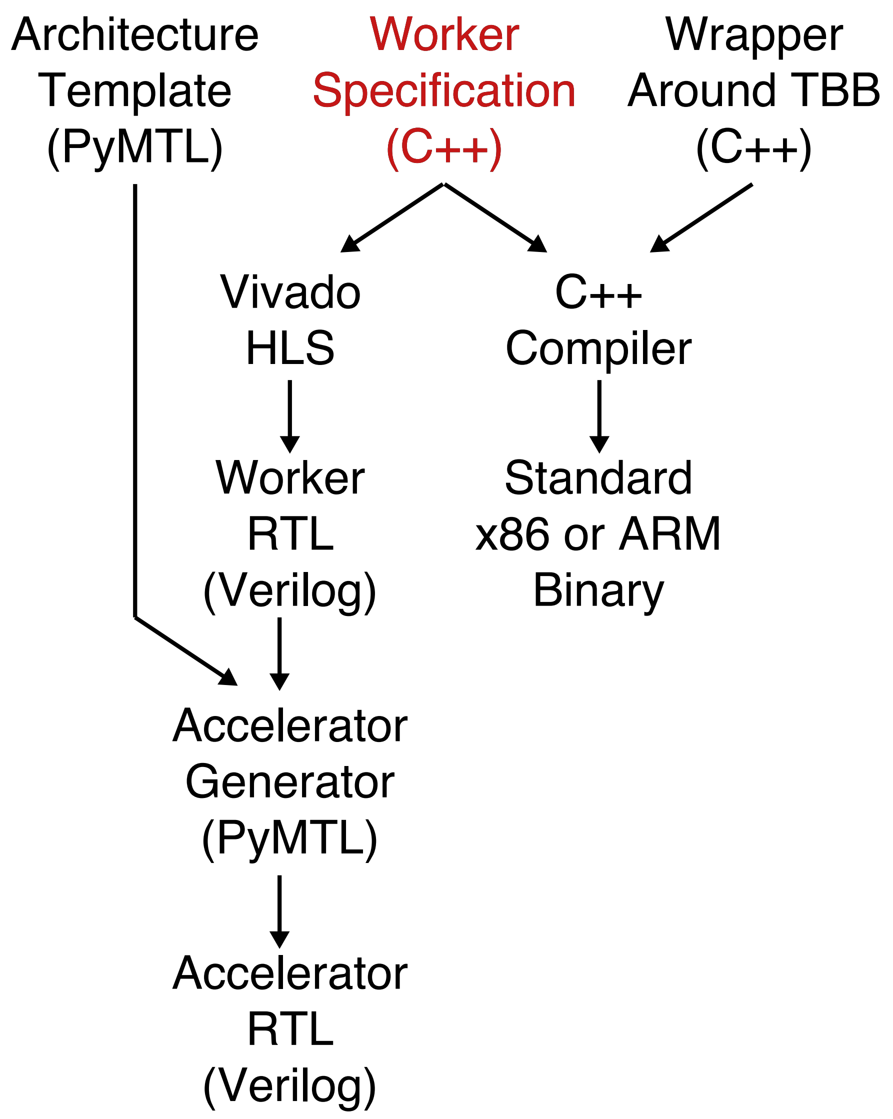


Design Methodology


Architecture Template (PyMTL)
Worker Specification (C++)
Vivado HLS
Worker RTL (Verilog)
Accelerator Generator (PyMTL)
Accelerator RTL (Verilog)



Design Methodology

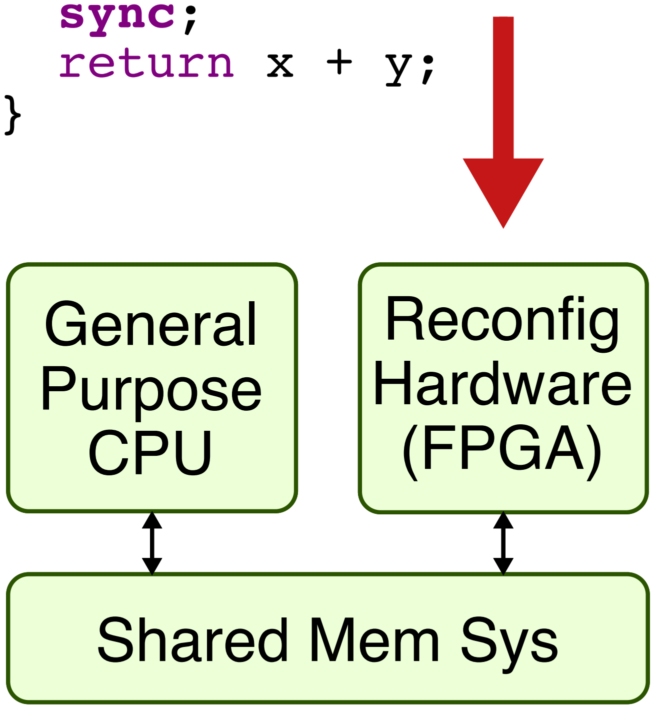

Architecture Template (PyMTL)
Worker Specification (C++)
Wrapper Around TBB (C++)
Vivado HLS
C++ Compiler
Worker RTL (Verilog)
Standard x86 or ARM Binary
Accelerator Generator (PyMTL)
Accelerator RTL (Verilog)

```
void FibWorkerHLS(
  TaskInPort<FibTask>  tin,
  TaskOutPort<FibTask> tout,
  SuccReqPort          sreq,
  SuccRespPort         sresp,
  ArgOutPort           aout )
{
  FibTask  task = task_in.read();
  task_k_t k    = task.k;

  if (task.type == FIB) {
    int n = task.x;
    if (n < 2)
      send_arg( Arg(k, n), aout );
    else {
      k = make_succ(SUM,k,2,sreq,sresp);
      spawn(FibTask(FIB,k,1,n-2), tout);
      spawn(FibTask(FIB,k,0,n-1), tout);
    }
  }
  else if (task.type == SUM) {
    int sum = task.x + task.y;
    send_arg(Arg(k, sum), aout);
  }
}
```

Accelerating *Dynamic* Parallel Algorithms on Reconfigurable Hardware

```
int fib( int n )
{
  if (n < 2)
    return n;
  int x = spawn fib(n-1);
  int y = fib(n-2);
  sync;
  return x + y;
}
```



Motivation

Computation Model

Accelerator Architecture

Design Methodology

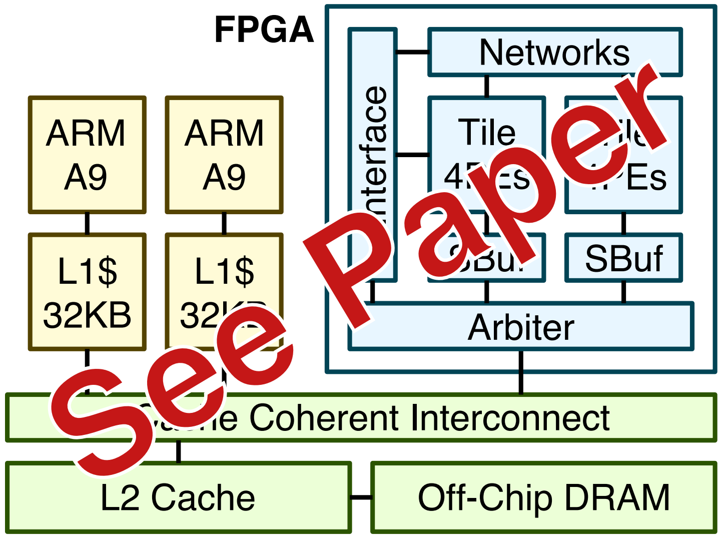
Evaluation

Applications

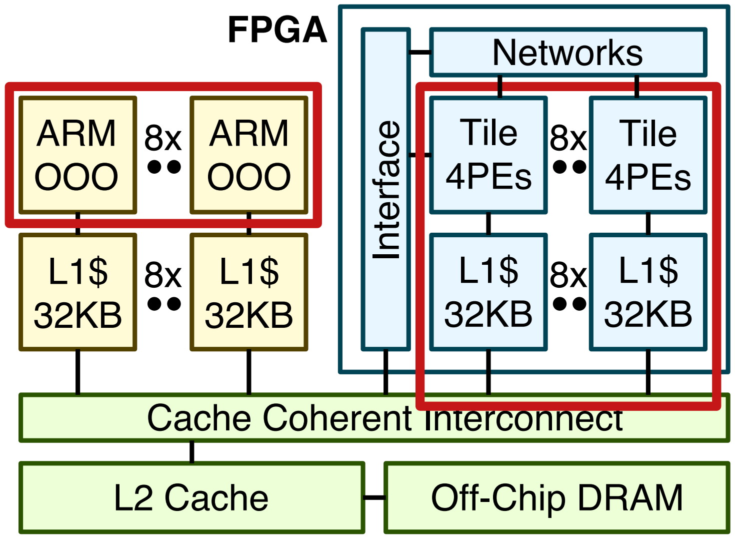
Name	Suite	Description	Pattern
nw	in-house	Needleman-Wunsch Algorithm	data-flow
quicksort	in-house	quicksort algorithm	fork/join
cilksort	Cilk apps	parallel merge sort algorithm	fork/join
queens	Cilk apps	N-queens problem	fork/join
knapsack	Cilk apps	0-1 knapsack problem	fork/join
uts	UTS	unbalanced tree search	fork/join
bbgemm	MachSuite	blocked matrix multiplication	data-parallel
bfsqueue	MachSuite	breadth first search	data-parallel
spmvcrs	MachSuite	sparse matrix-vector mult	data-parallel
stencil2d	MachSuite	3D stencil computation	data-parallel

- Optimized software baseline implemented using Intel Cilk Plus with ARM NEON auto-vectorization
- C++ application driver/worker implemented with design methodology

Current and Future CPU+FPGA Platforms

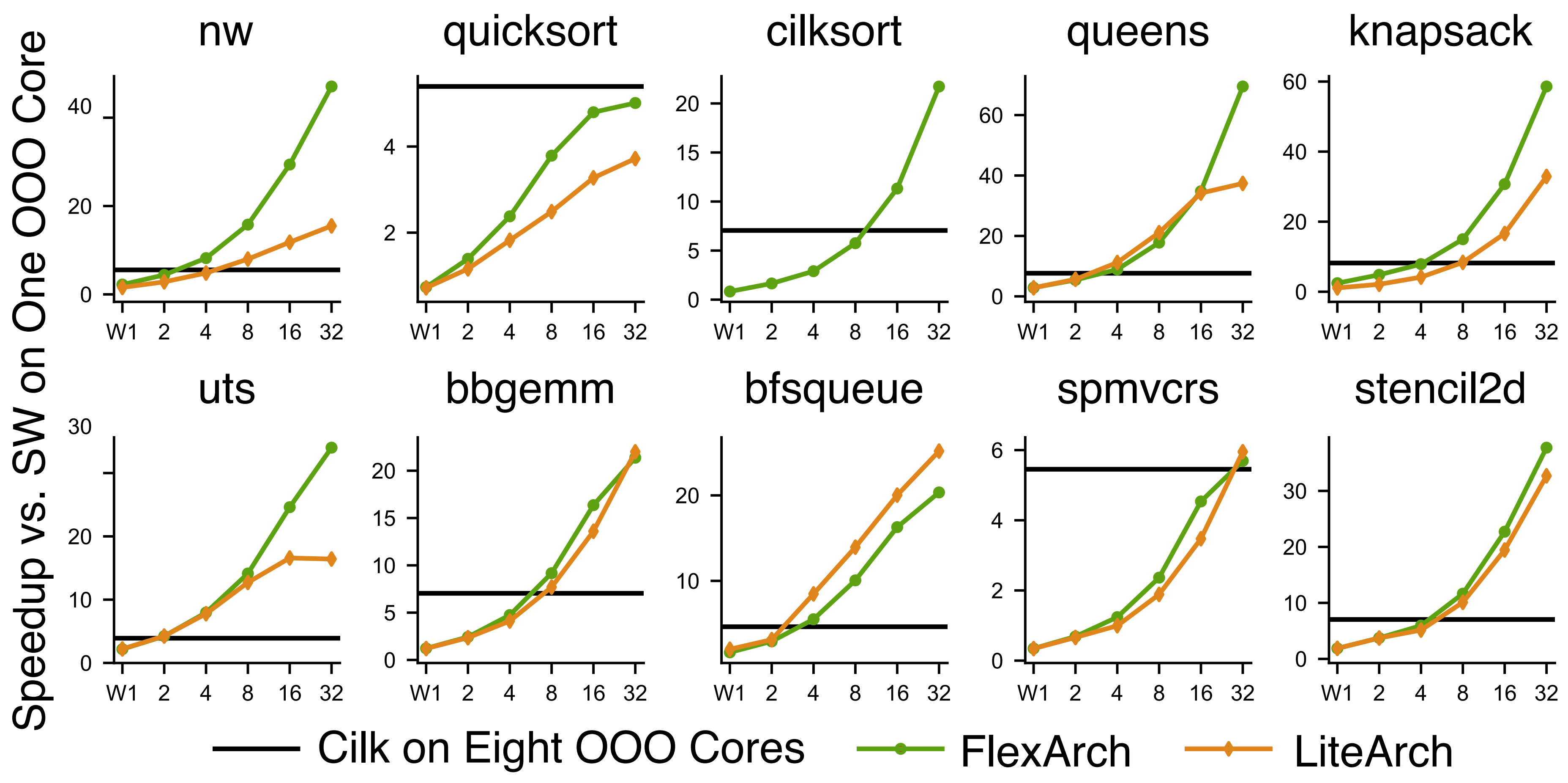


- **Current Zynq-7000 SoC Platform**
  - Prototype using Zedboard
  - Two 667MHz ARM Cortex-A9 cores
  - Xilinx 7-series integrated FPGA fabric (modest capacity, 142MHz)
  - Xcel uses stream buffers
  - Lower BW: FPGA ↔ coherent mem sys



- **Future CPU+FPGA Platform**
  - Simulation study using gem5
  - Eight 1GHz ARM 4-way OOO cores
  - Xilinx 7-series integrated FPGA fabric (larger capacity, 200MHz)
  - Xcel uses coherent 2x-pumped 32KB L1$
  - Higher BW: FPGA ↔ coherent mem sys

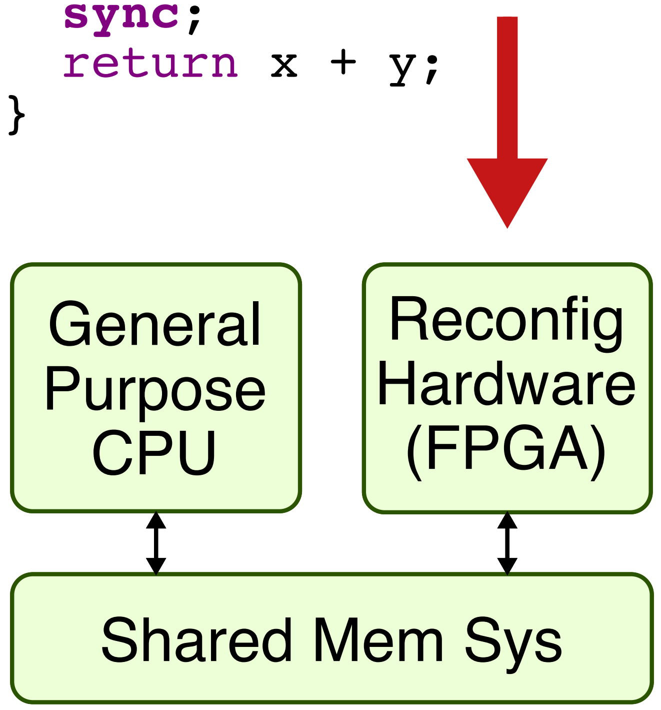
Speedup on Future CPU+FPGA Platform



- FlexArch is 4× faster than 8 cores, 24× faster than 1 core (geo mean)
- FlexArch is faster than LiteArch for more dynamic algorithms (load balancing)
- See paper for scalability, resource usage, energy efficiency, cache size study

Accelerating *Dynamic* Parallel Algorithms on Reconfigurable Hardware

```
int fib( int n )
{
  if (n < 2)
    return n;
  int x = spawn fib(n-1);
  int y = fib(n-2);
  sync;
  return x + y;
}
```



- Importance of exploring techniques for accelerating more complex applications on reconfigurable hardware
- We have described a promising approach to accelerate dynamic parallel algorithms
  - computation model using explicit continuation passing
  - accelerator architecture based on work stealing
  - design methodology combining a PyMTL-based architectural template with high-level synthesis