Hydra: Auto Parallelism

Kunle Olukotun, Lance Hammond and Mark Willey
James Chicken

Presented by: Chengyao Li, Yang Shi, Jiaxin Li, Liang Zhang

Overview

- Introduction
- Methodology
- Implementation
- Analysis and Evaluation

Introduction

- Motivation
  - Instruction-Level Parallelism (ILP) still have limits
  - Writing Thread-Level Parallelism code is difficult
  - Automatically parallelize function calls!
- Related Work
  - Improving the Performance of Speculatively Parallel Applications on the Hydra CMP [K. Olukotun, 99]
  - Cilk[R. Blumofe, 95] and OpenMP[L. Dagum, 98]

Methodology

- Function Offloading

```
void f(float x) {
  float y;
  y = sqrt(x);

  //...


  std::cout << y;
}
```

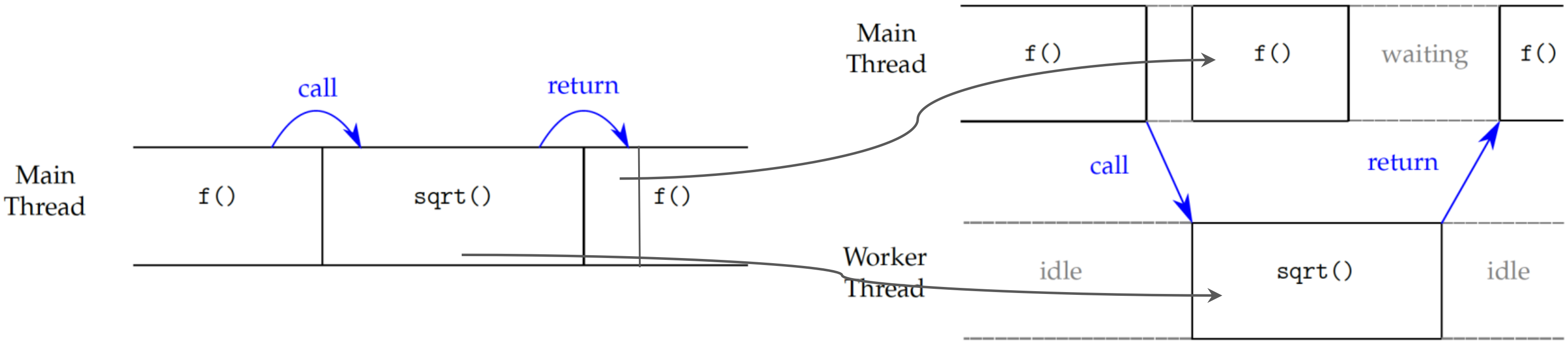
```
void f(float x) {
    float y;
    std::thread t(sqrt, x, y);

    //...

    t.join();
    std::cout << y;
}
```

Methodology

- Function Offloading



$$SerialCost = cost(f_{extra}) + cost(sqrt)$$

$$ParallelCost = max\{cost(f_{extra}), cost(sqrt)\} + cost(spawn)$$

Methodology

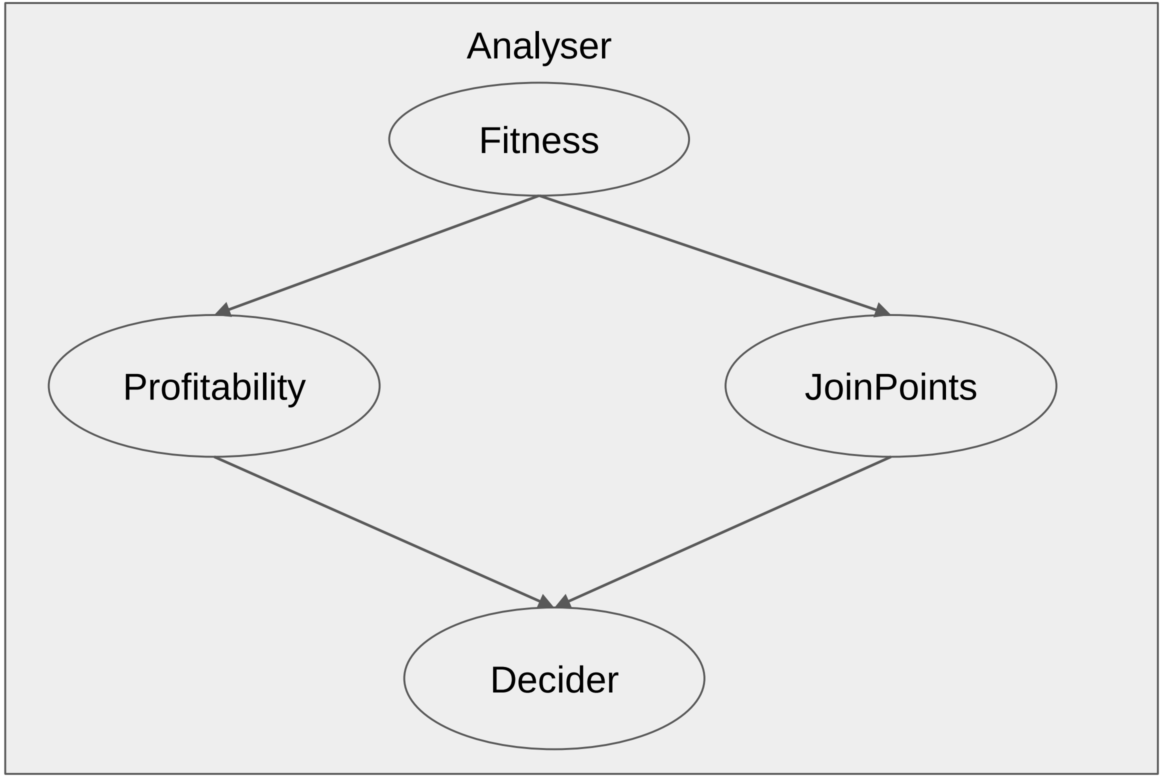
- Function Offloading

$$max\{cost(caller), cost(callee)\} + cost(spawn)$$
$$<$$
$$cost(caller) + cost(callee)$$

Implementation

Basic architecture

contains 4 passes:



Fitness Analysis Pass

Function: Check which functions can be offloaded

- ❖ Depending only on their arguments instead of global state
    - no pointer
    - no global variable

Pointer check

➔ Instead of alias analysis, just check function arguments type

```
bool hasPointerArgs(const llvm::Function &F) {
  return std::any_of(F.arg_begin(), F.arg_end(),
    [](const llvm::Argument &arg) {
      return arg.getType()->isPointerTy();
    });
}
```

Fitness Analysis Pass

Global variable check

```
bool referencesGlobalVariables(const llvm::Function &F) {
  return std::any_of(inst_begin(F), inst_end(F),
    [](const llvm::Instruction &I) {
      return std::any_of(I.op_begin(), I.op_end(),
        [](const llvm::Use &U) {
          return isa<GlobalVariable>(U) ||
                 isa<GlobalAlias>(U);
        });
    });
}
```

$$\mathcal{O}\left(\max_{I \in insts(F)} \left\{ |ops(I)| |insts(F)| \right\}\right)$$

The complexity is actually linear, because LLVM uses Three-Address Instructions, hence |*ops(I)*| is approximately constant

Profitability Analysis Pass

- Dedicated to estimating how much work is performed by the callee
  - count num of instructions (naive approach, underestimate all)
  - only count emitted instructions
  - dealing with function calls
  - dealing with loops (more precise estimate)

Profitability Analysis Pass

Final heueristic

$$h_3(F) = \sum_{I \in insts(F)} tripCount(I) cost_3(I), \quad \text{where}$$

$$cost_3(I) = \begin{cases} h_3(calledFun(I)) + 1 & \text{if } I \text{ is a non-recursive call,} \\ 1 & \text{if } I \text{ emits,} \\ 0 & \text{otherwise.} \end{cases}$$

$$tripCount(I) = \begin{cases} t & \text{if } I \text{ is in a loop with trip count provably } t, \\ 1 & \text{otherwise.} \end{cases}$$

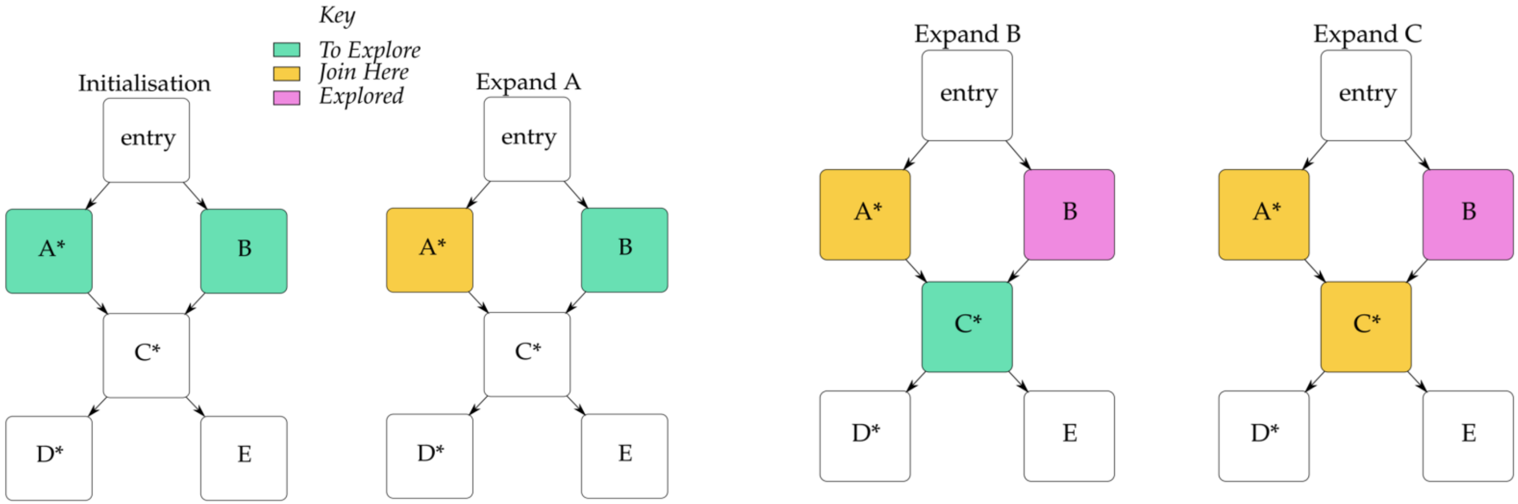
JoinPoints Analysis Pass

- Finding where to join with an offloaded function → ensure correctness
  - prerequisite: SSA (no worry about aliases)
  - basic approach: find joint points (detect dependencies)

```
llvm::Instruction *findJoinPoint(llvm::CallInst *ci,
                                 const bb_iter I,
                                 const bb_iter E) {
  auto join = std::find_if(I, E,
    [&](llvm::Instruction &inst) {
      return std::any_of(inst.value_op_begin(),
                         inst.value_op_end(),
                         [&](Value *v) { return v == ci; });
    });
  return (join != E ? &*join : nullptr);
}
```

JoinPoints Analysis Pass

- ❖ Preferable method: At-Least-Once Joining (based on thread pool runtime)
  - ➢ The spawning happens at the entry point
  - ➢ * represent the usage of return value from the function of spawning thread

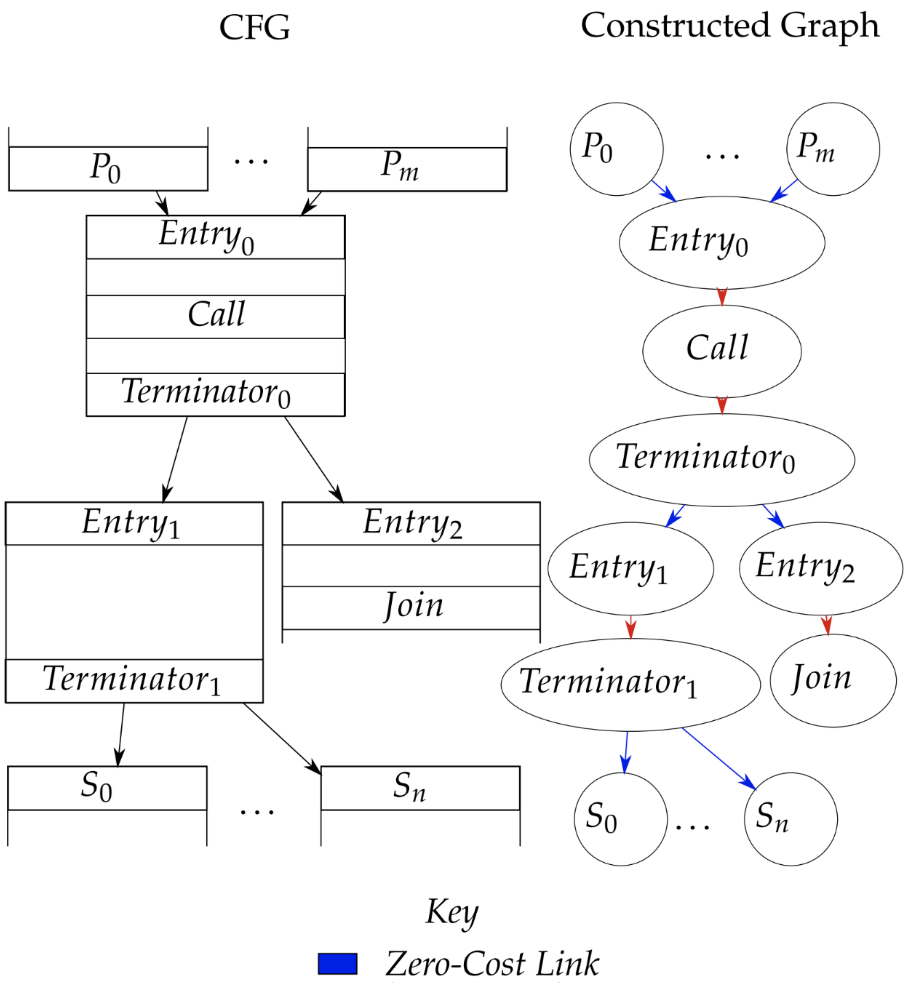


Decider Analysis Pass

- Compute the cost of caller
  - “exactly once” runtimes
    - hueristic: $$cost(caller) = \sum_{I \in range(S,J)} cost_3(I).$$
    - not general
  - “at least once” runtimes
    - naive approach: applying Cost3(I) to every possible instruction between the S and J.

Decider Analysis Pass

- ❖ problems with naive approach
  - ➢ time complexity
  - ➢ overestimate
- ❖ conservative alternative approach
  - ➢ build a DAG
  - ➢ find shortest paths from S to each J
  - ➢ compute the costs of all the paths as results



Accumulate the Results

❖ Existing approaches

There is the optimist's approach:

$$cost(caller) = \max_{p \in paths} \{cost(p)\}$$

The pessimist's approach:

$$cost(caller) = \min_{p \in paths} \{cost(p)\}$$

And the realist's approach:

$$cost(caller) = \frac{1}{|paths|} \sum_{p \in paths} cost(p)$$

❖ Better approach (we are implementing)

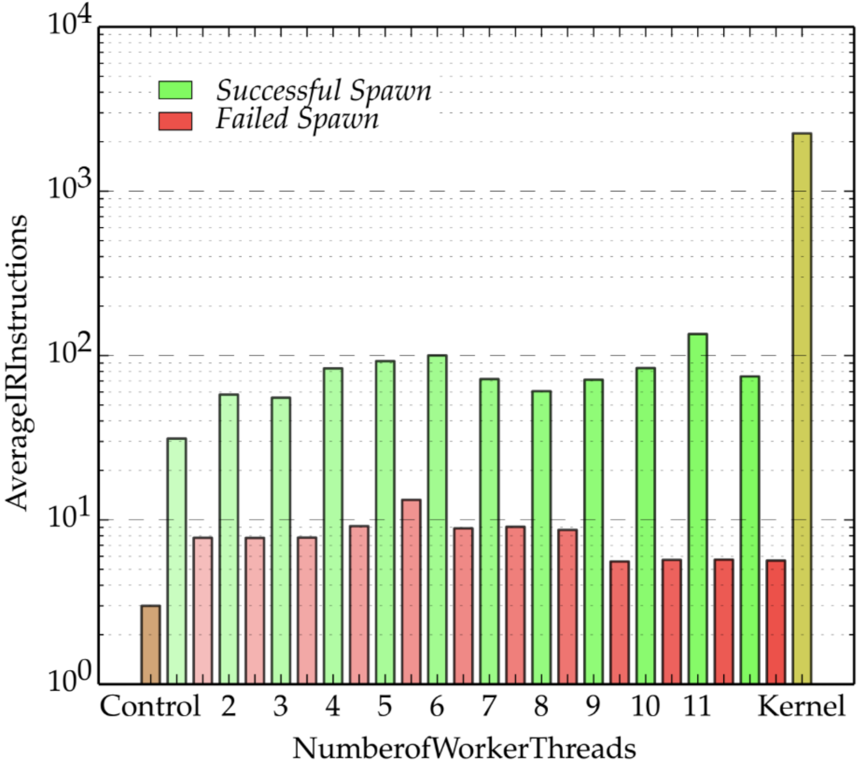
$$cost(caller) = \sum_{p \in paths} \mathbb{P}(p) cost(p).$$

Evaluation

- For the evaluation part, the paper evaluate from the following three aspects
  - Runtime Microbenchmarks
  - Performance Testing
  - Scalability Tests

Runtime Microbenchmarks

The right figure shows: Mean number of instructions to spawn and join with tasks on Hydra's supported runtimes, compared to an increment operation. Green bars denote an empty thread pool, while red bars denote a pool at capacity. The y-axis uses a log10-scale.



Performance Testing

	Arithmetic Mean (99% confidence interval)	Standard Deviation
Serial	$5.676736s \pm 0.00004244s$	0.0005201s
Parallel	$2.908464s \pm 0.001249s$	0.01530s

The above table shows the arithmetic mean of the results, as well as a 99% confidence interval. We can see that there is no doubt that the project has resulted in a significant average speedup.

Scalability Tests

From the graph, we can see consistent improvements to mean performance from four to seven worker threads.

Beyond seven worker threads, performance gets worse

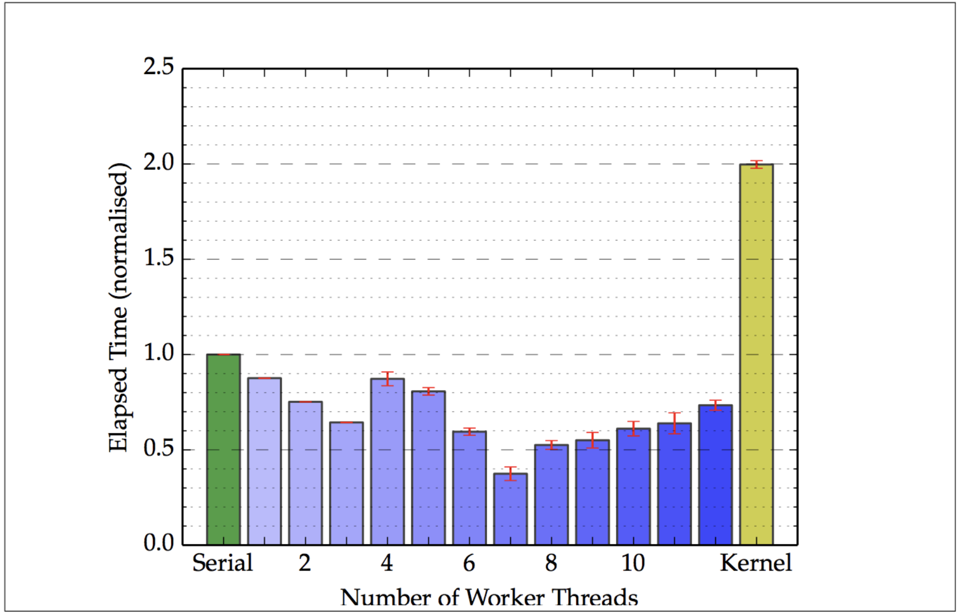


Figure 4.3: Means and Standard Deviations of 50 *n*-body simulations (100 bodies, 200 steps) on the quad-core machine.

Strength and Weakness of Hydra

- Strength
  - Hydra aims to work without any programmer input
  - Hydra aims to be independent of source language
  - Hydra provides an implementation of a high-quality thread pool
- Weakness of Hydra
  - Hydra does not support for exceptions
  - Hydra does not allow pointer arguments

Thank You for Listening!

Any Questions?