Automatically Tuning Task-Based Programs for Multi-core Processors

Jin Zhou
Brian Demsky

Department of Electrical Engineering and Computer Science
University of California, Irvine

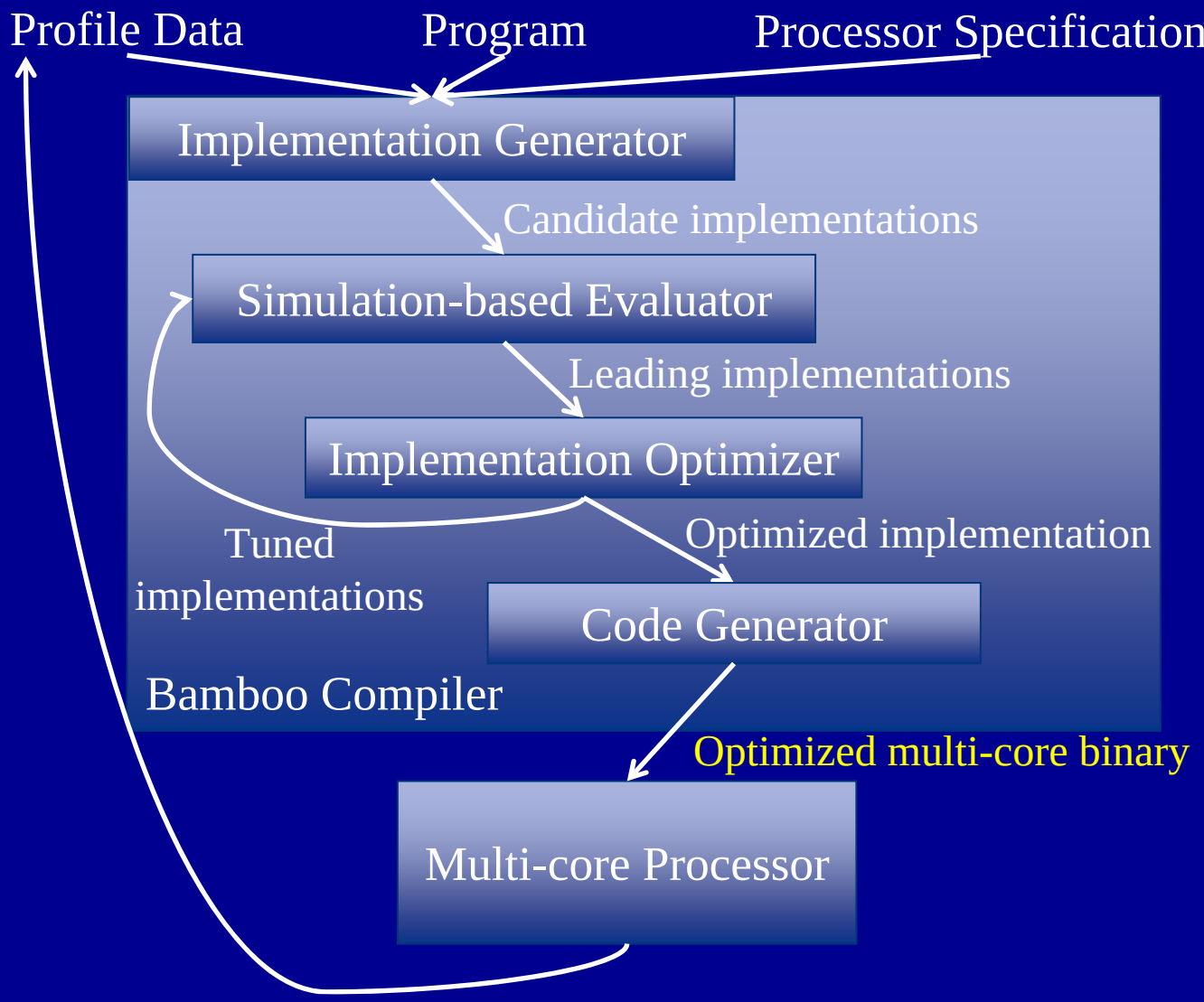
Motivation

- Recent microprocessor trends
  - Number of cores increased rapidly
  - Architectures vary widely
- Challenges for software development
  - Parallelization is now key for performance
  - Current parallel programming model: threads + locks
    - Hard to develop correct and efficient parallel software
    - Hard to adapt software to changes in architectures

Goals

- Automatically generate parallel implementation
- Automatically tune parallel implementation

Overview


Profile Data
Program
Processor Specification
Implementation Generator
Candidate implementations
Simulation-based Evaluator
Leading implementations
Implementation Optimizer
Tuned implementations
Optimized implementation
Code Generator
Bamboo Compiler
Optimized multi-core binary
Multi-core Processor

Example

- MonteCarlo Example
  - Partitions problem into several simulations
  - Executes the simulations in parallel
  - Aggregates results of all simulations

Bamboo Language

- A hybrid language combines data-flow and Java
  - Programs are composed of tasks
  - Tasks compose with dataflow-like semantics
  - Tasks contain Java-like object-oriented code internally
  - Programs cannot explicitly invoke tasks
  - Runtime automatically invokes tasks
- Supports standard object-oriented constructs including methods and classes

Bamboo Language

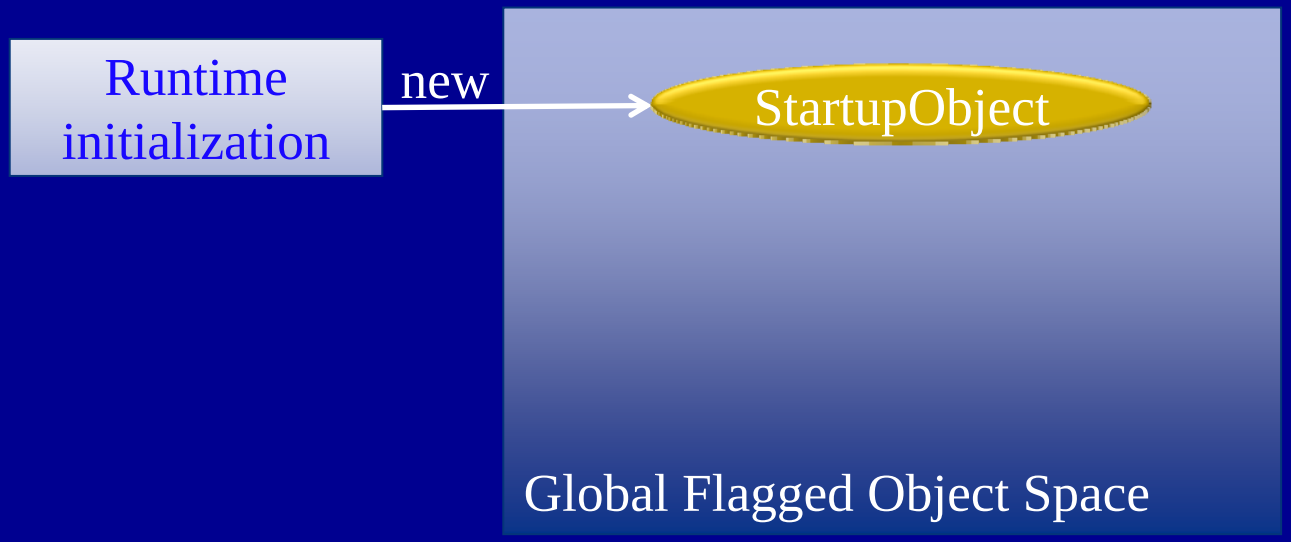
- Flags
  - Capture current role (type state) of object in computation
  - Each flag captures an aspect of the object's state
  - Change as the object's role evolves in program
  - Support orthogonal classifications of objects

```
class Simulator {
    flag run;
    flag submit;
    flag finished;
    ...
}

class Aggregator {
    flag merge;
    flag finished;
    …
}
```

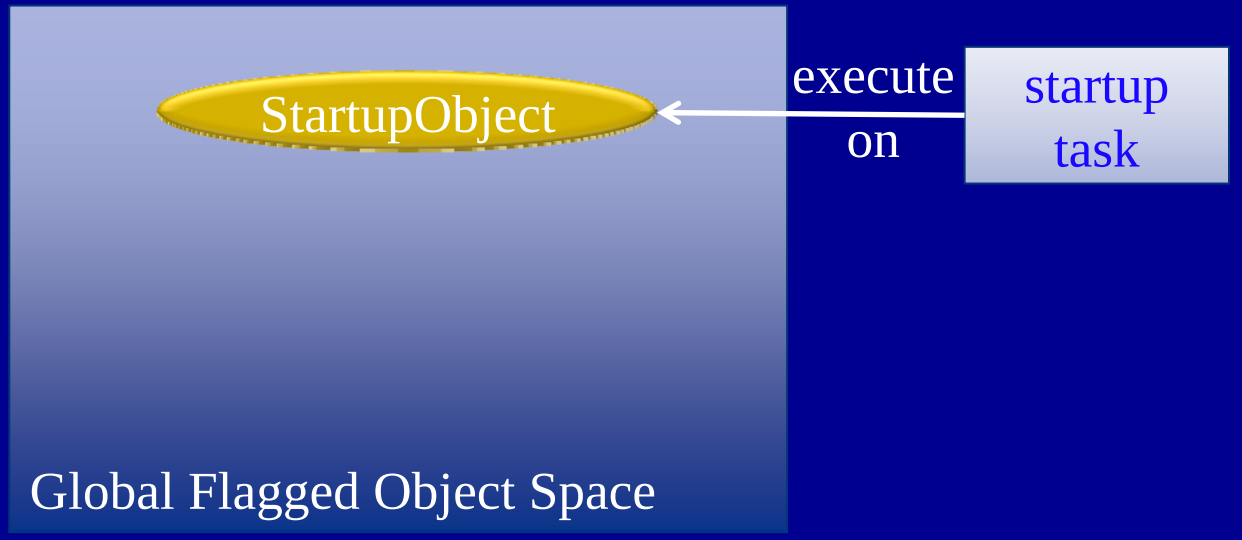
```
task startup(StartupObject s in initialstate) {
    Aggregator aggr = new Aggregator(s.args[0]){merge:=true};
    for(int i = 0; i < 4; i++)
        Simulator sim = new Simulator(aggr){run:=true};
    taskexit(s: initialstate:=false);
}
task simulate(Simulator sim in run) {
    sim.runSimulate();
    taskexit(sim: run:=false, submit:=true);
}
task aggregate(Aggregator aggr in merge,
                    Simulator sim in submit) {
    boolean allprocessed = aggr.aggregateResult(sim);
    if (allprocessed)
        taskexit(aggr: merge:=false, finished:=true;
                    sim: submit:=false, finished:=true);
    taskexit(sim: submit:=false, finished:=true);
}
```

Bamboo Program Execution



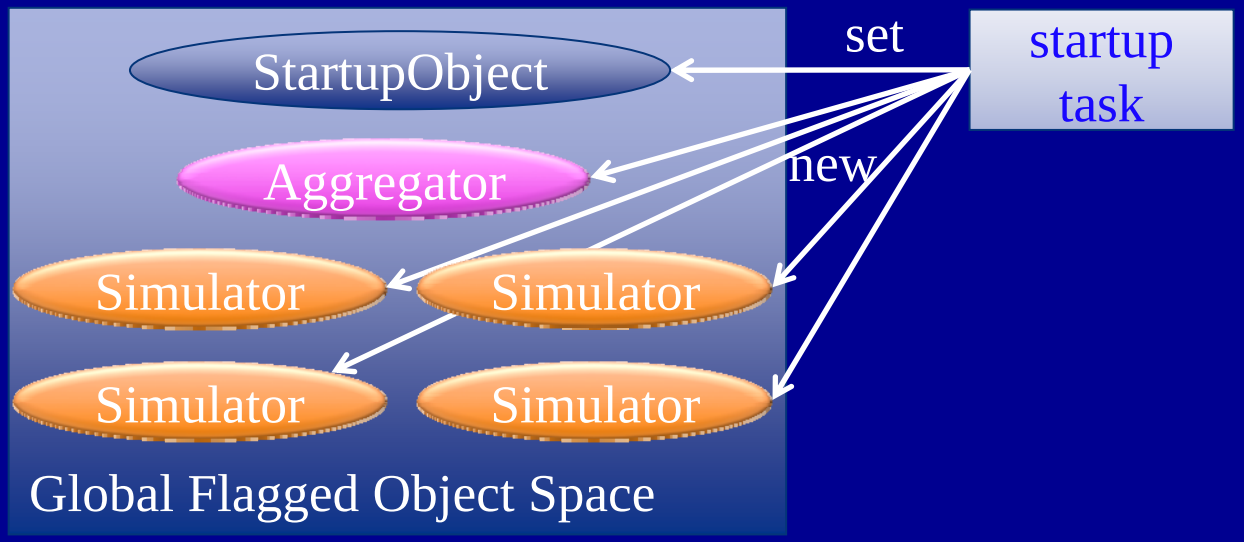
StartupObject initialstate state finished state

Bamboo Program Execution



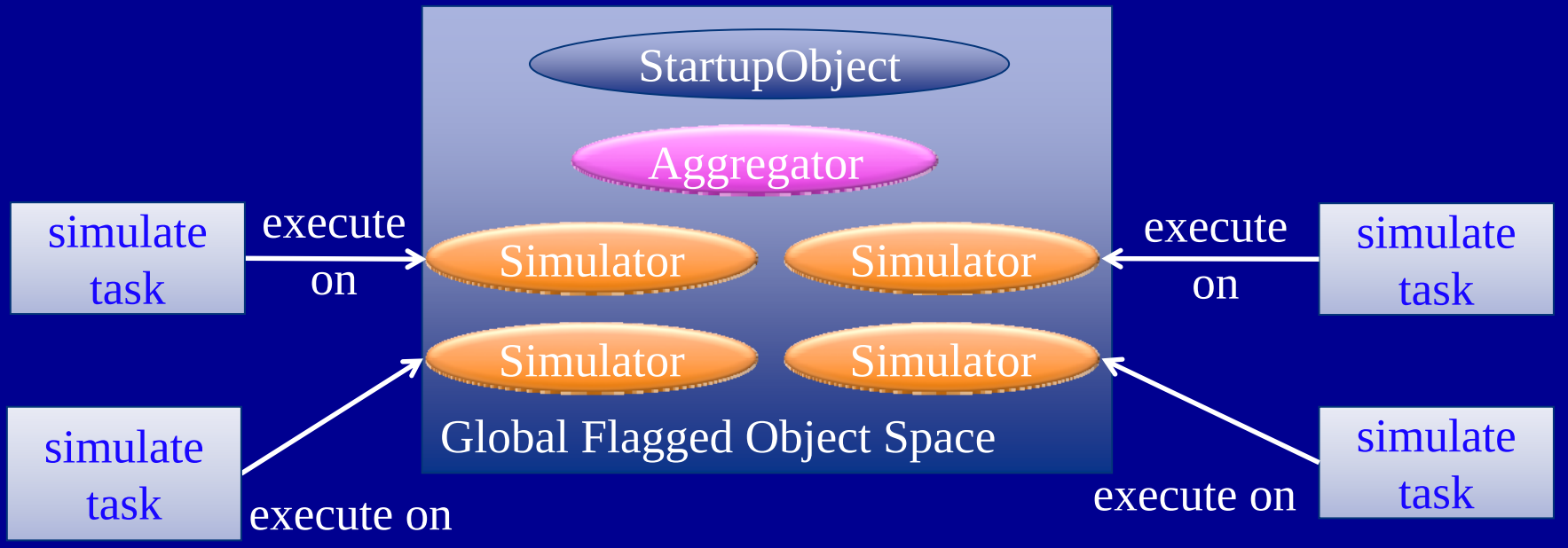
StartupObject initialstate state finished state

Bamboo Program Execution



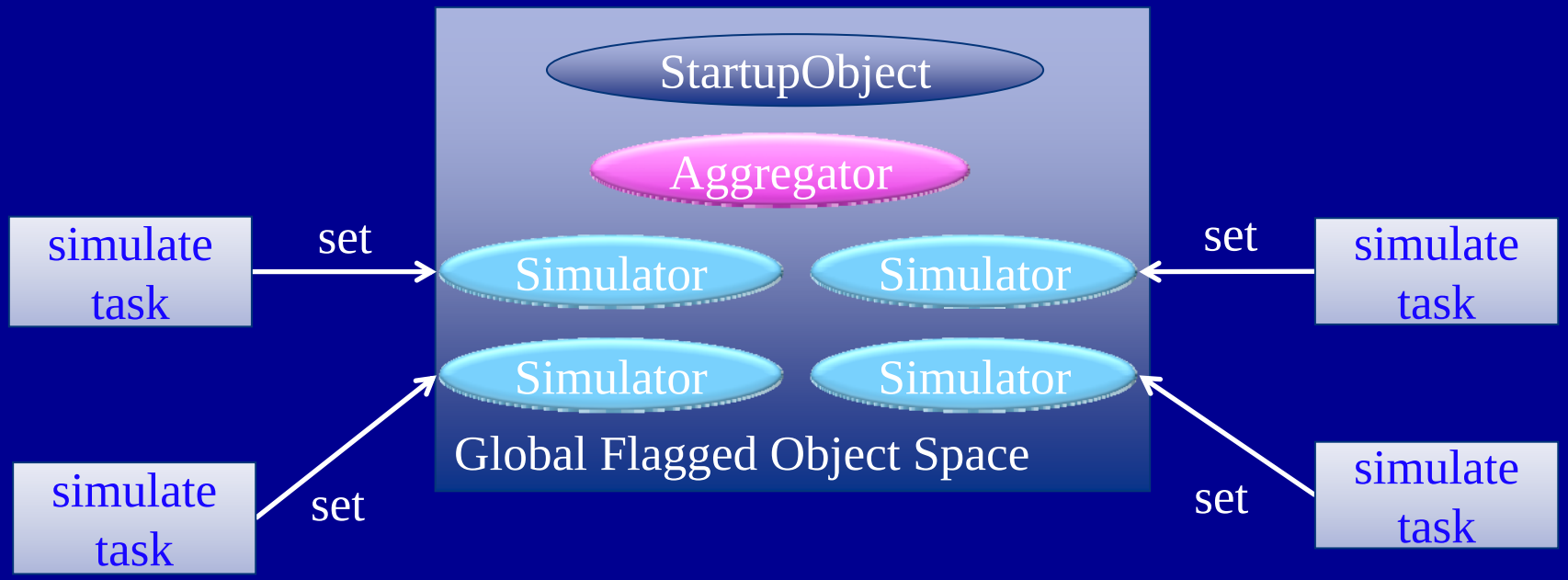
StartupObject	initialstate state	finished state	
Aggregator	merge state	finished state	
Simulator	run state	submit state	finished state

Bamboo Program Execution



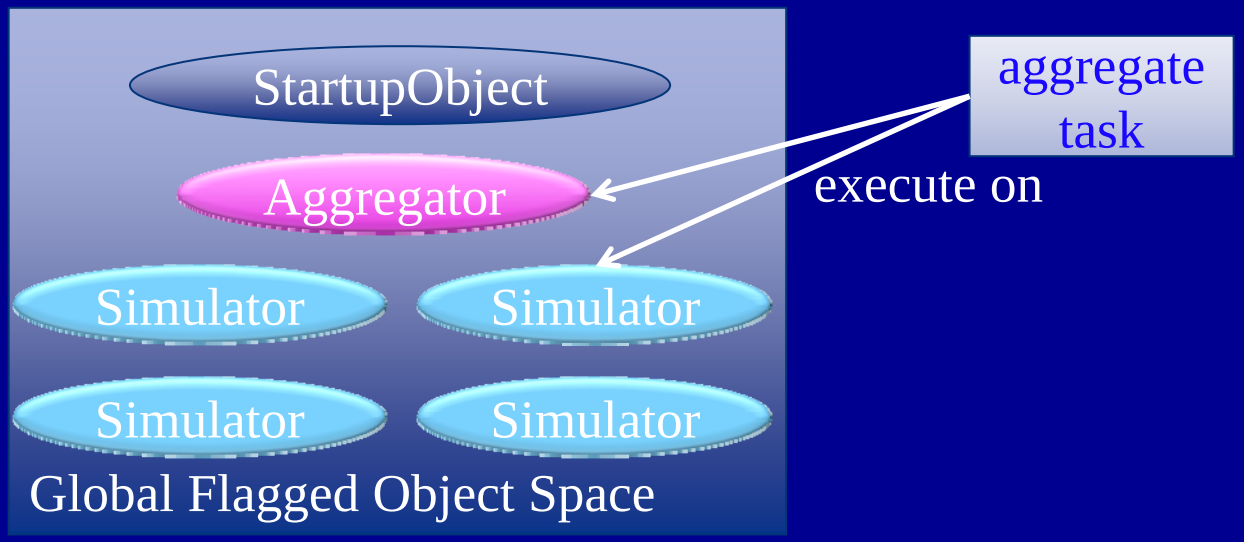
StartupObject	initialstate state	finished state	
Aggregator	merge state	finished state	
Simulator	run state	submit state	finished state

Bamboo Program Execution



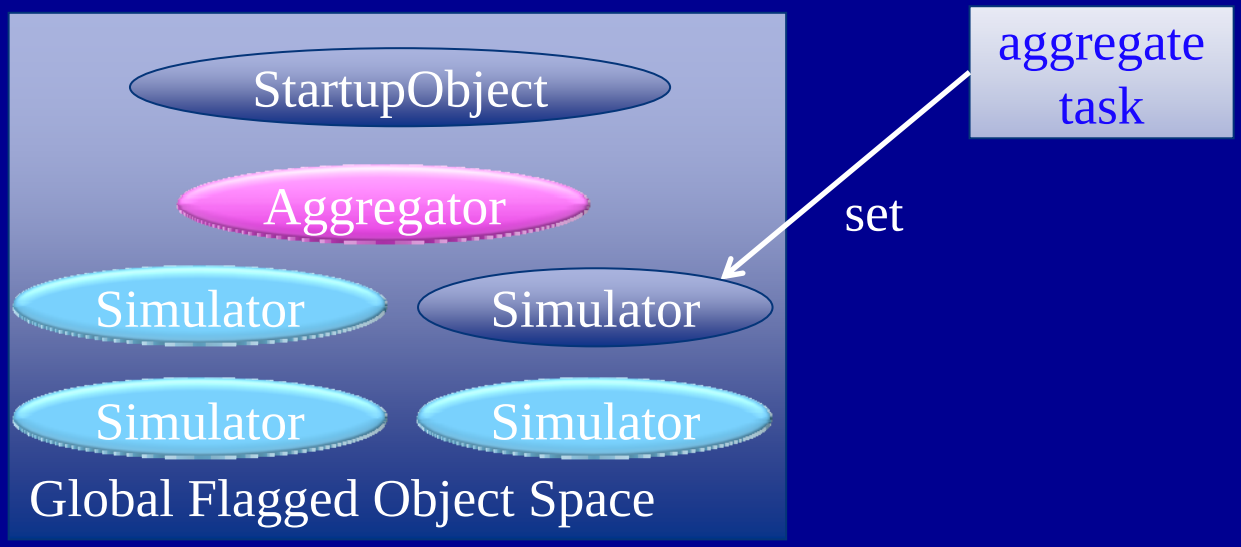
StartupObject	initialstate state	finished state	
Aggregator	merge state	finished state	
Simulator	run state	submit state	finished state

Bamboo Program Execution



StartupObject	initialstate state	finished state	
Aggregator	merge state	finished state	
Simulator	run state	submit state	finished state

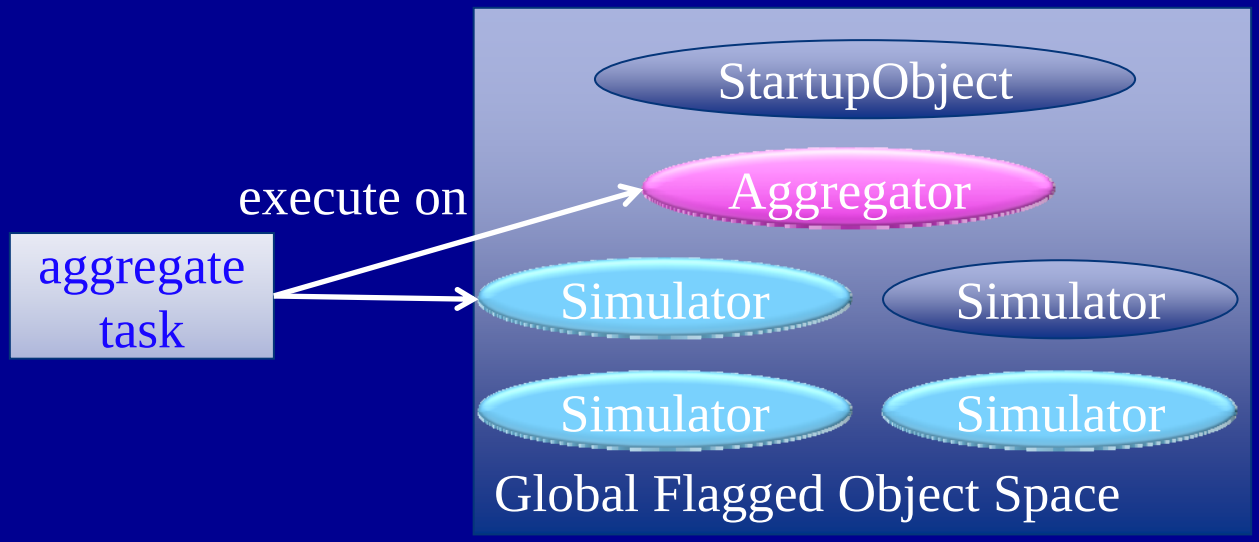
Bamboo Program Execution



StartupObject	initialstate state	finished state
Aggregator	merge state	finished state
Simulator	run state	submit state

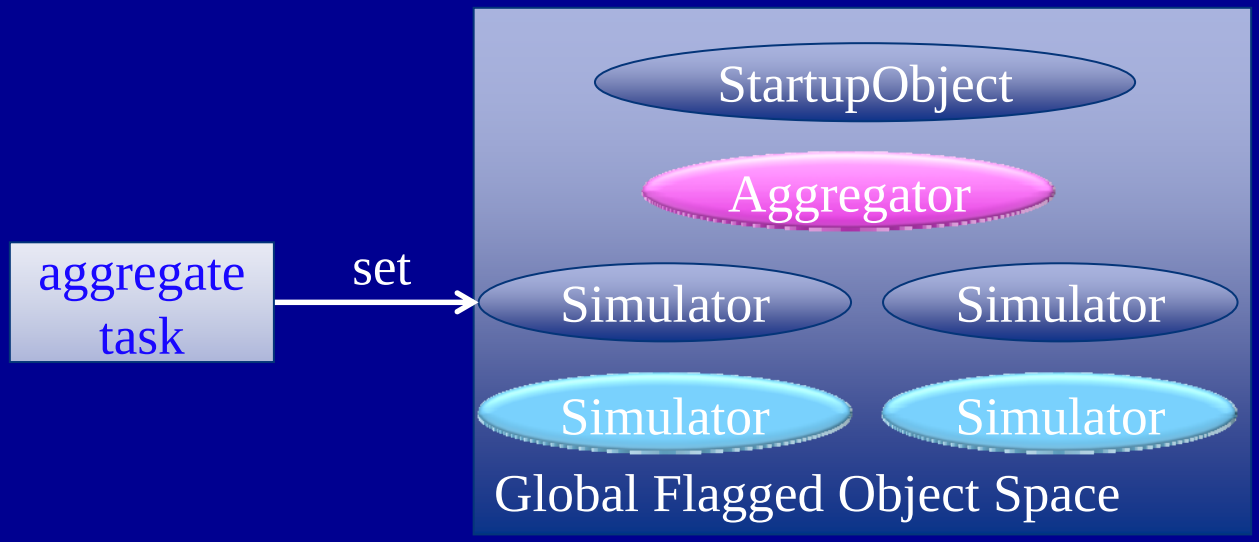
finished state

Bamboo Program Execution



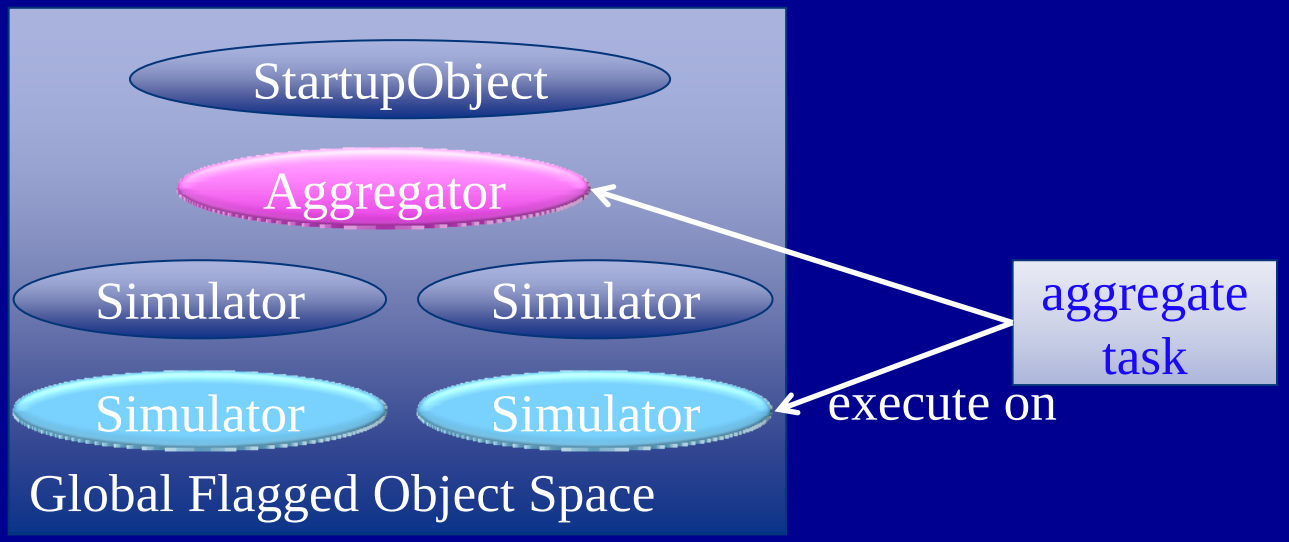
StartupObject	initialstate state	finished state	
Aggregator	merge state	finished state	
Simulator	run state	submit state	finished state

Bamboo Program Execution



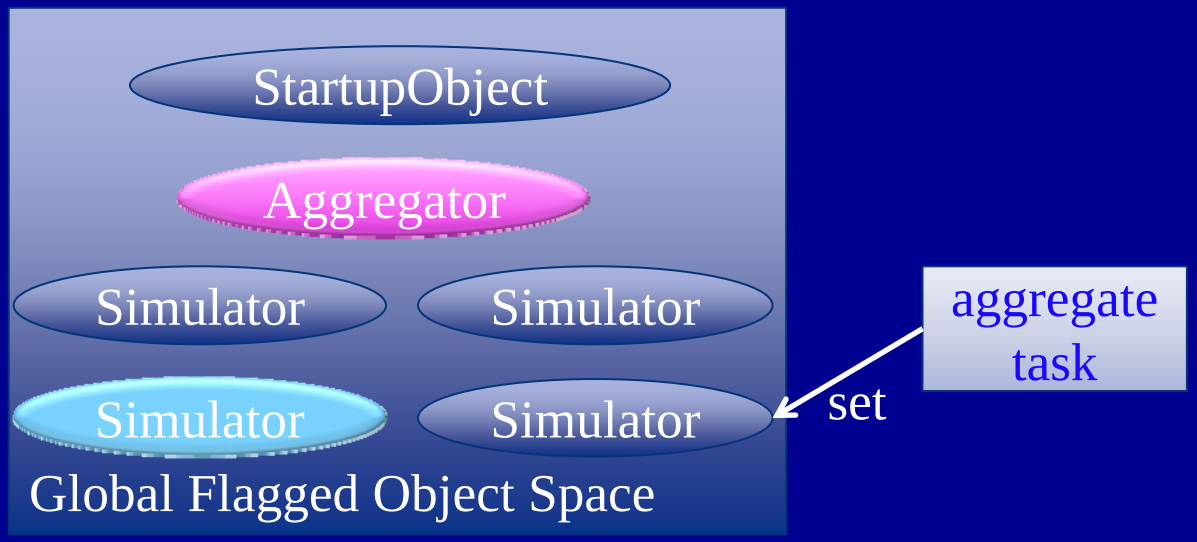
StartupObject	initialstate state	finished state	
Aggregator	merge state	finished state	
Simulator	run state	submit state	finished state

Bamboo Program Execution



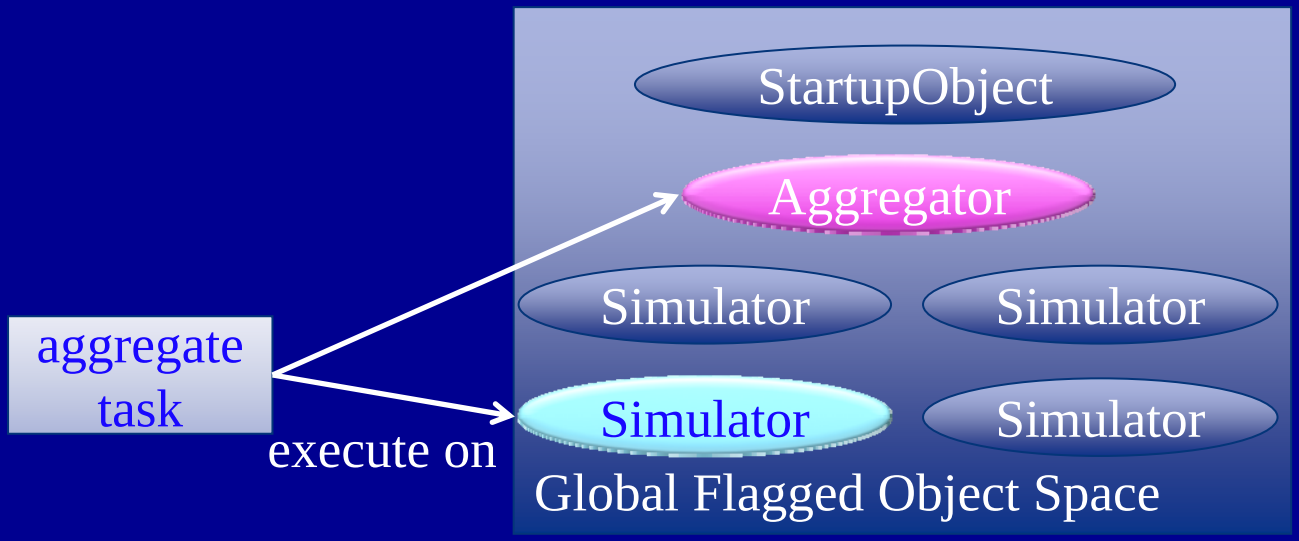
StartupObject	initialstate state	finished state	
Aggregator	merge state	finished state	
Simulator	run state	submit state	finished state

Bamboo Program Execution



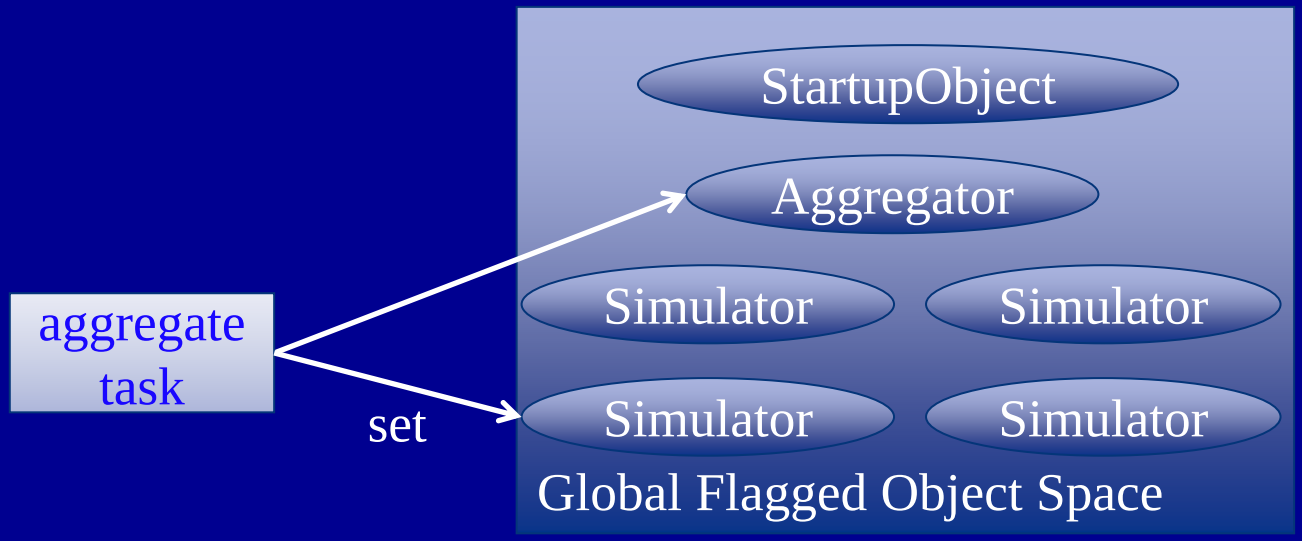
StartupObject	initialstate state	finished state	
Aggregator	merge state	finished state	
Simulator	run state	submit state	finished state

Bamboo Program Execution



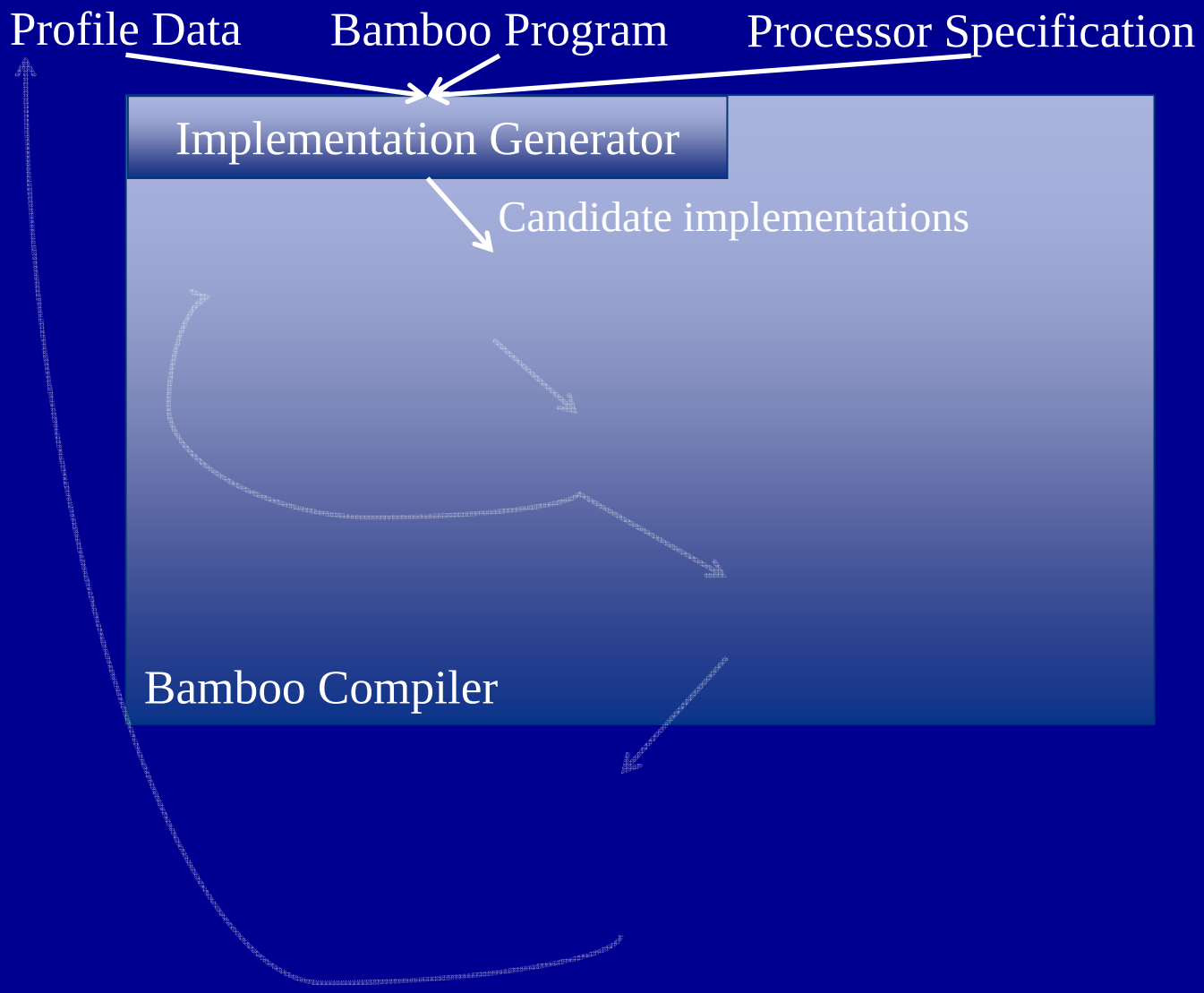
StartupObject	initialstate state	finished state	
Aggregator	merge state	finished state	
Simulator	run state	submit state	finished state

Bamboo Program Execution



StartupObject	initialstate state	finished state	
Aggregator	merge state	finished state	
Simulator	run state	submit state	finished state

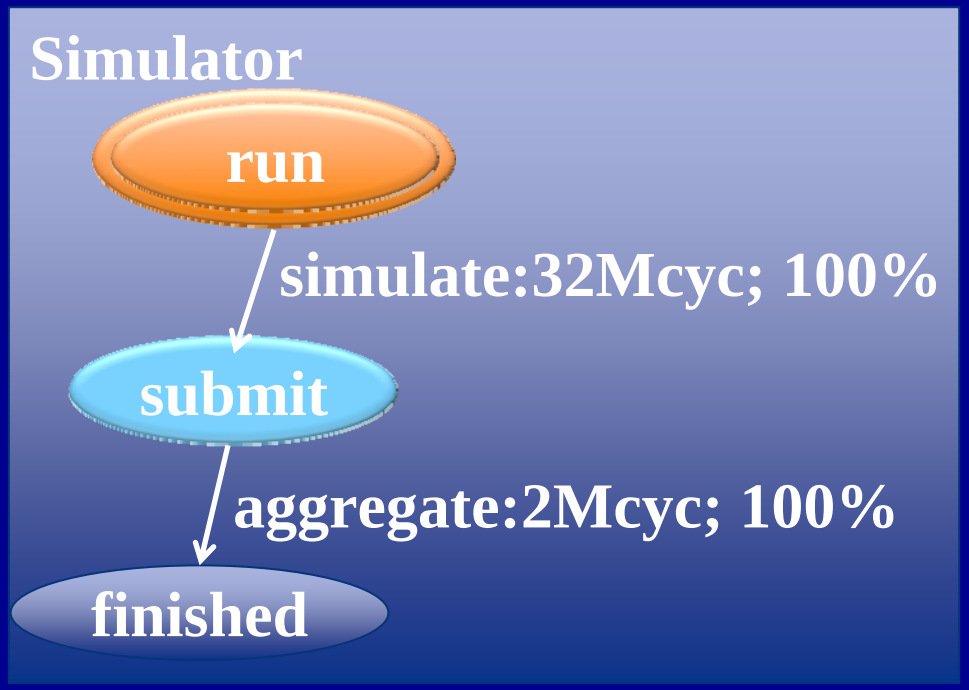
Implementation Generation



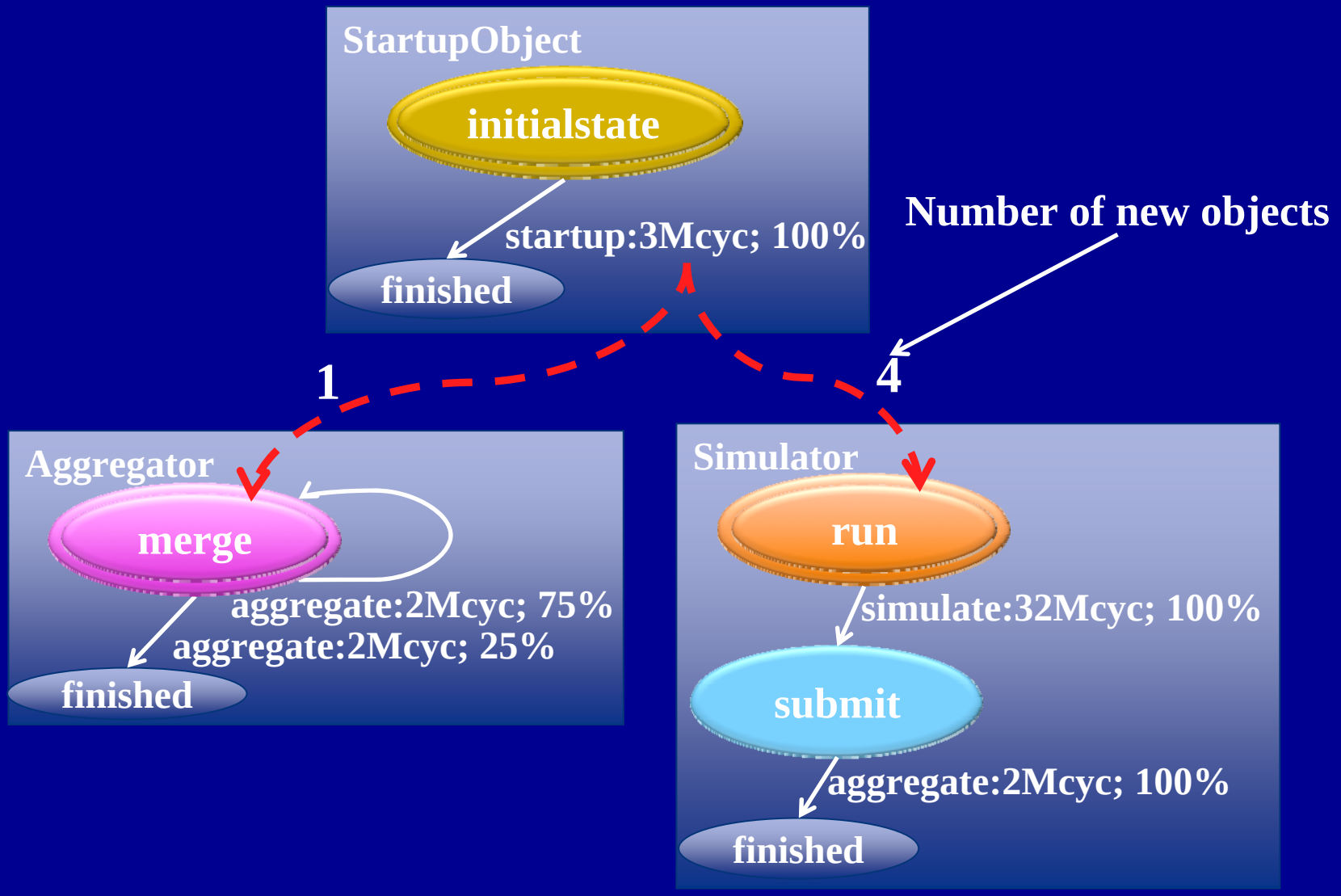
Implementation Generation

- Dependence Analysis: analyzes data dependence between tasks
- Parallelism Exploration: extracts potential parallelism
- Mapping to Cores: maps the program to real processor

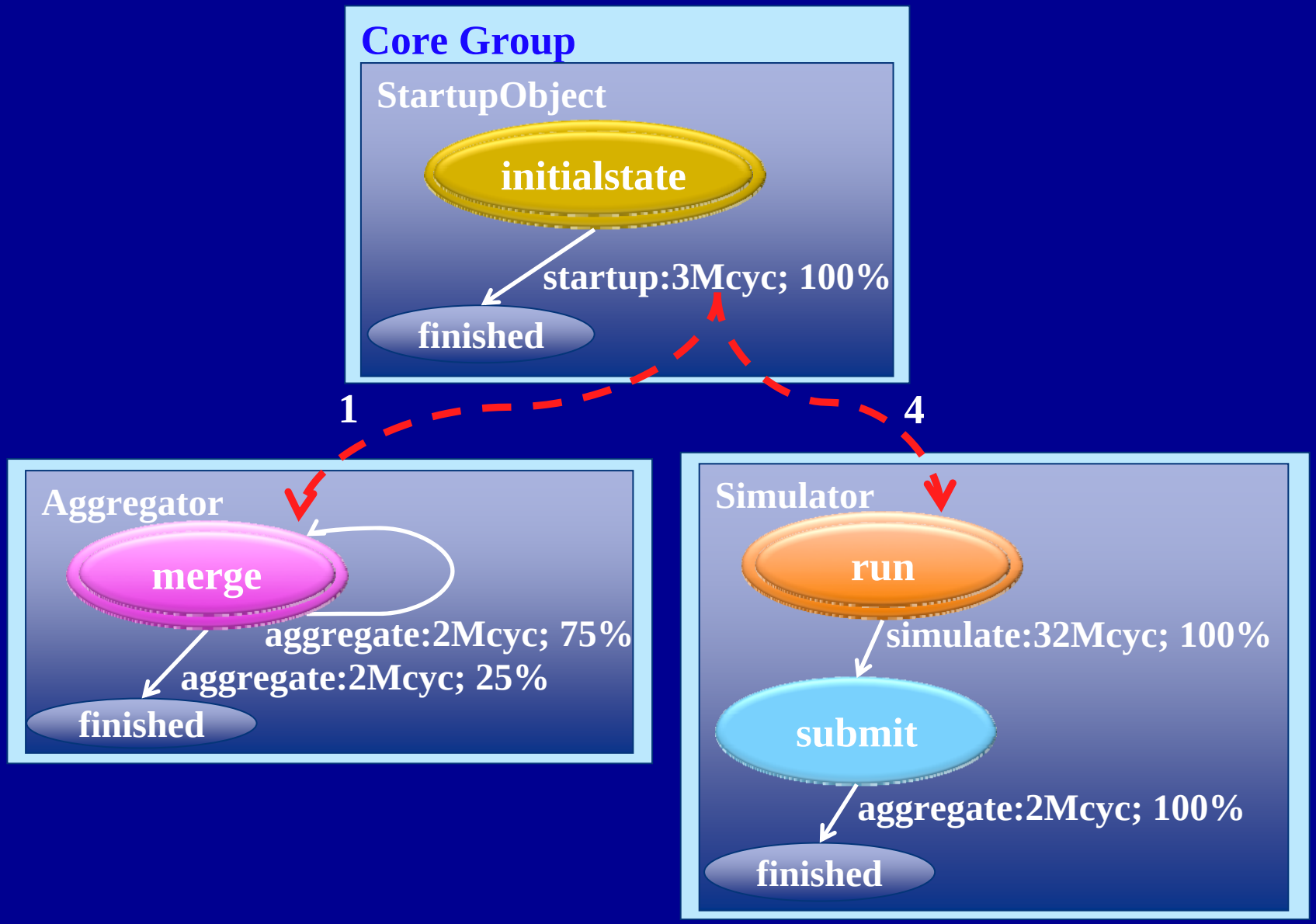
Flag State Transition Graph (FSTG)



Combined Flag State Transition Graph (CFSTG)



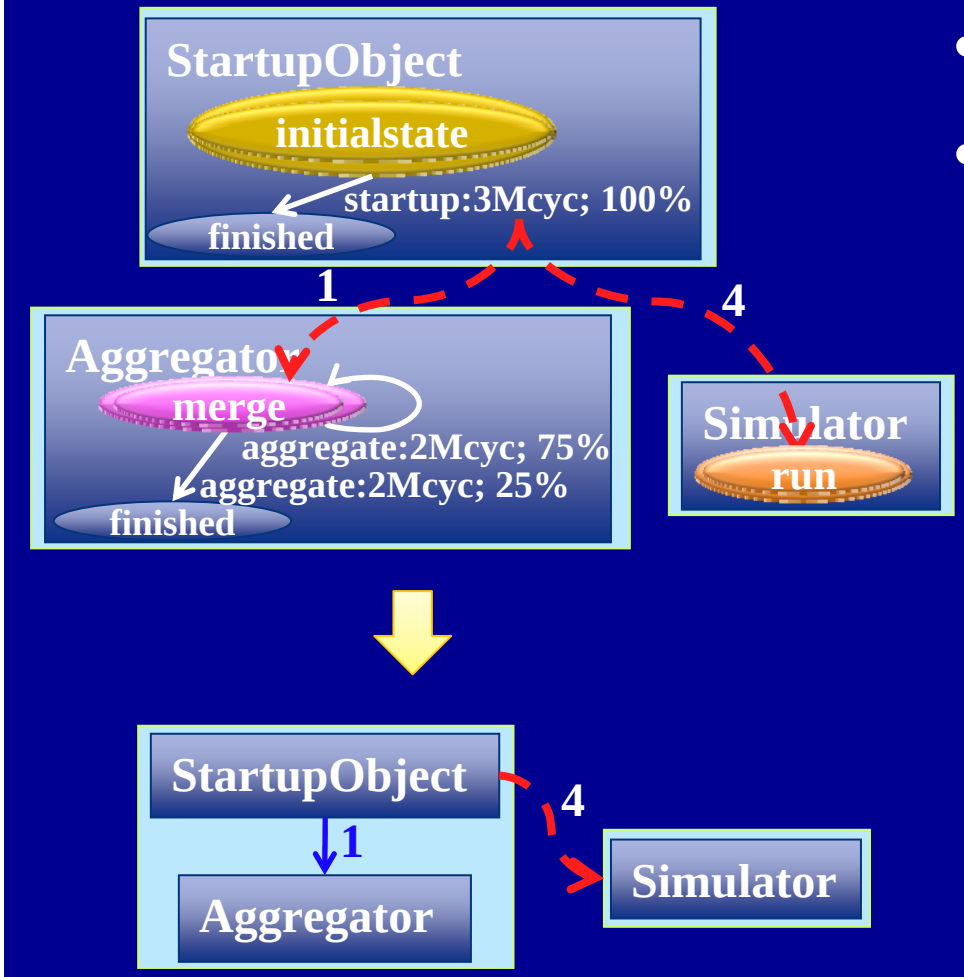
Initial Mapping



Preprocessing Phase

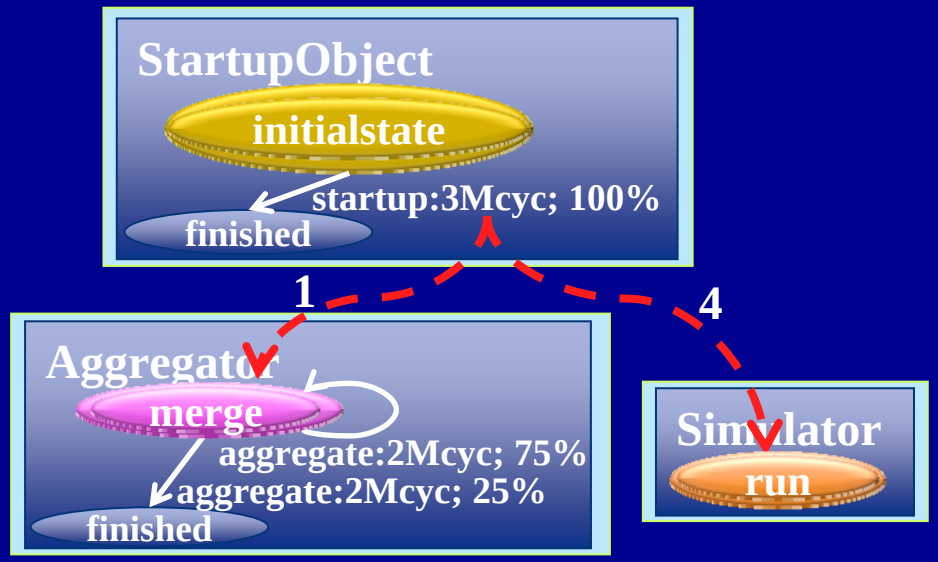
- Identifies strongly connected components (SCC) and merges them into a single core group
- Converts CFSTG into a tree of core groups by replicating core groups as necessary

Data Locality Rule

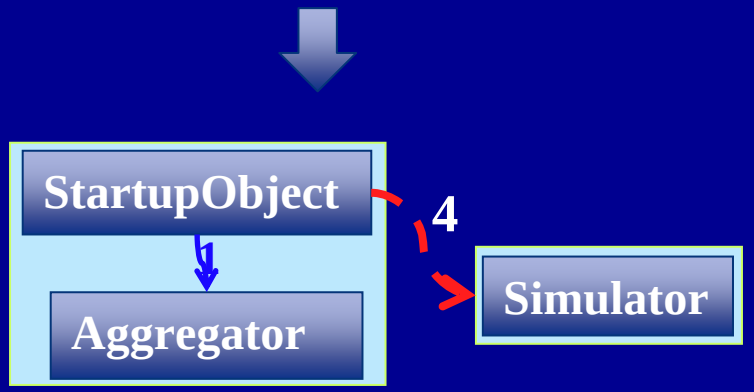


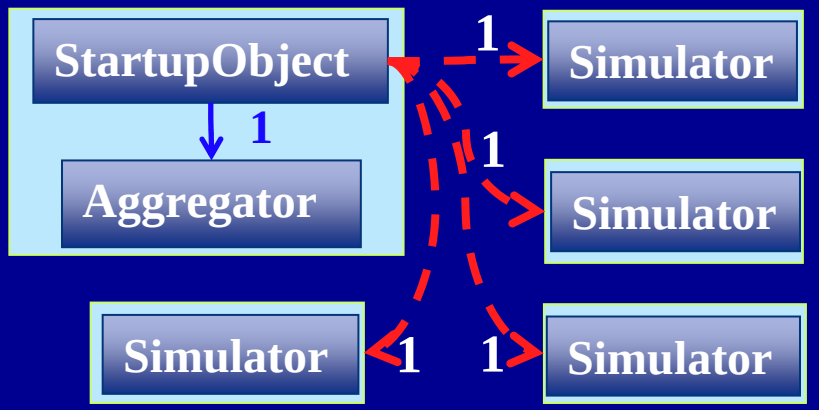
- Default rule
- Maximize data locality to improve performance
  - Minimizes inter-core communications
  - Improves cache behavior

Data Parallelization Rule

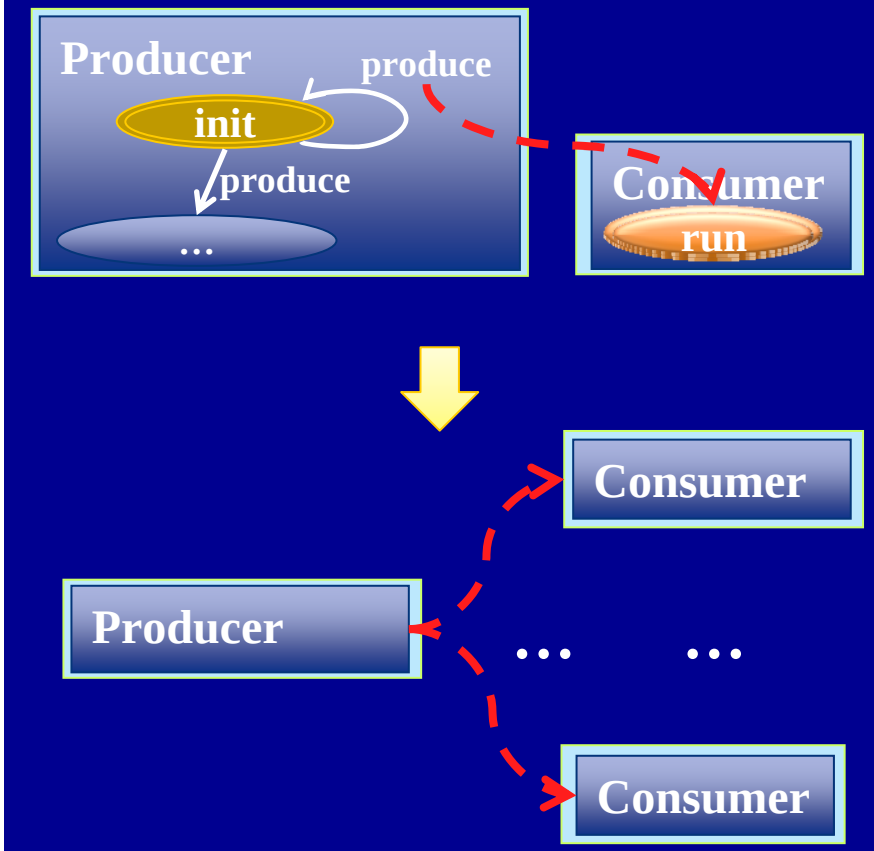


- To explore potential data parallelism





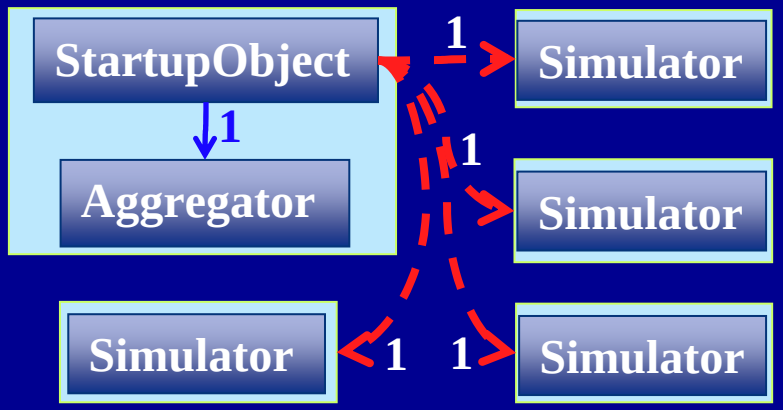
Rate Matching Rule



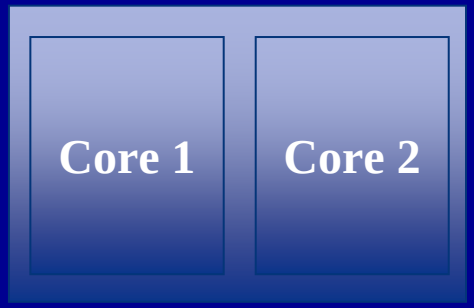
- If the producer executes multiple times in a cycle, how many consumers are required?
- Match two rates to estimate the number of consumers
  - Peak new object creation rate
  - Object consumption rate

Mapping to Processor

- Extended CFSTG



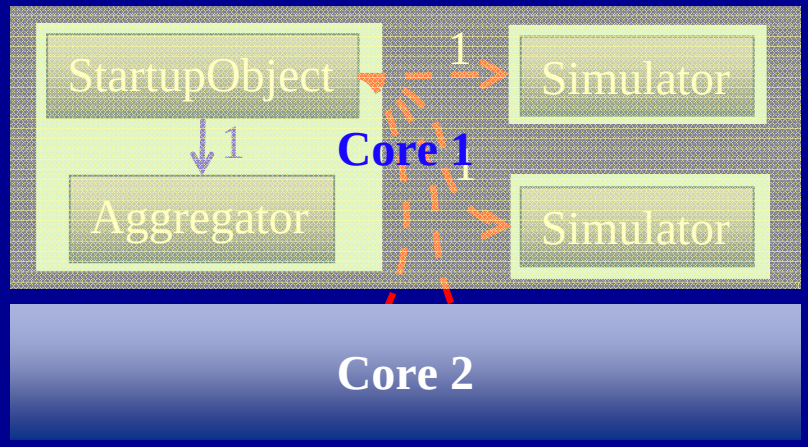
- Constraint: limited cores



- Map CFSTG core groups to physical cores

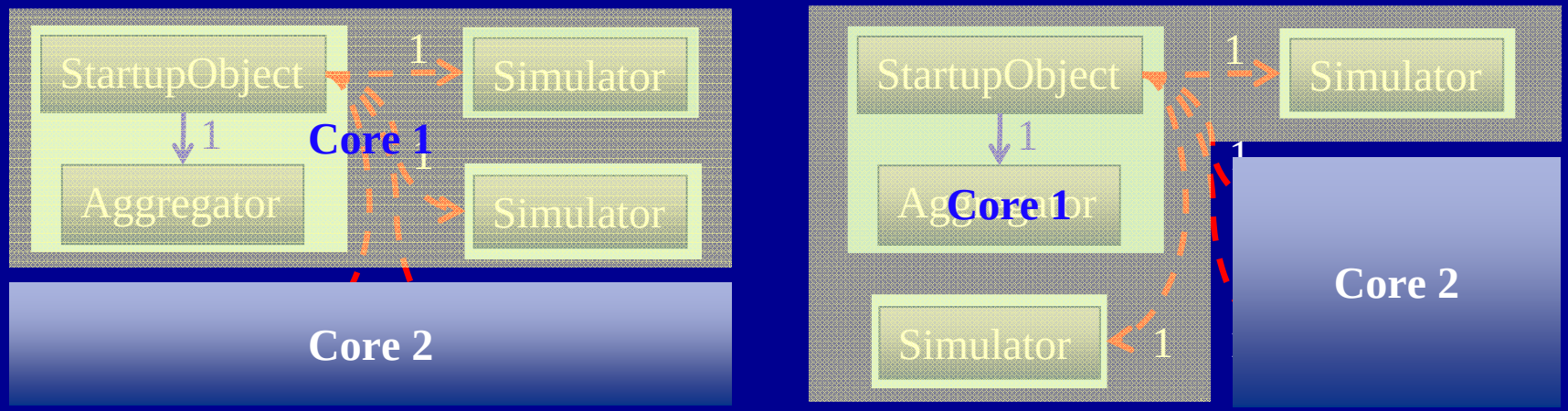
Mapping to Cores

- One possible mapping



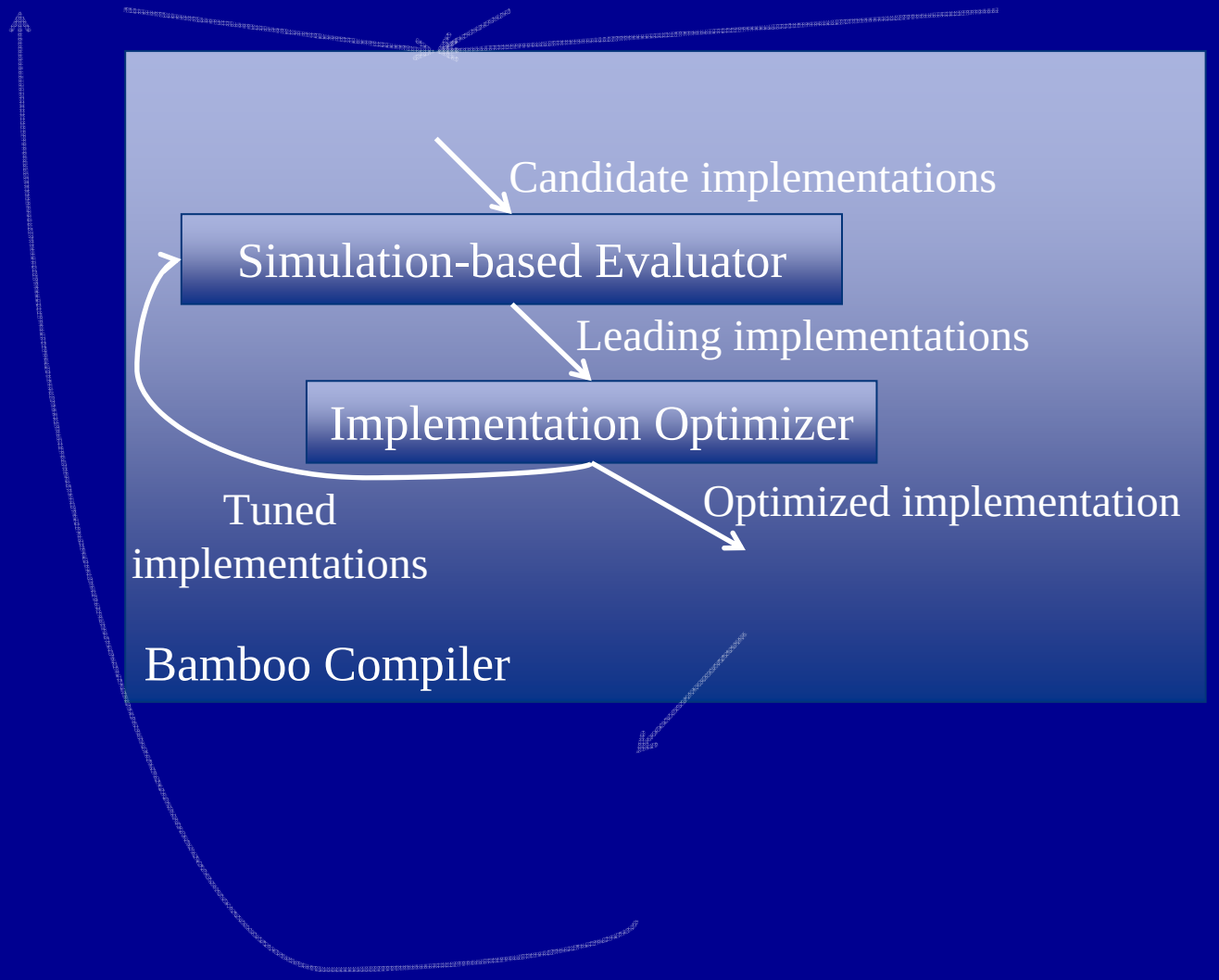
Mapping to Cores

- Isomorphic mappings: have same performance



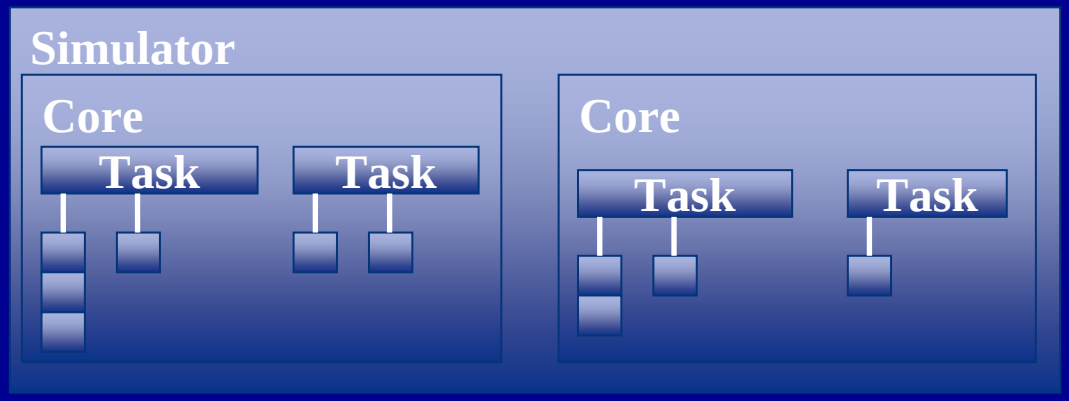
- Backtracking-based search: to generate non-isomorphic implementations

Implementation Generation



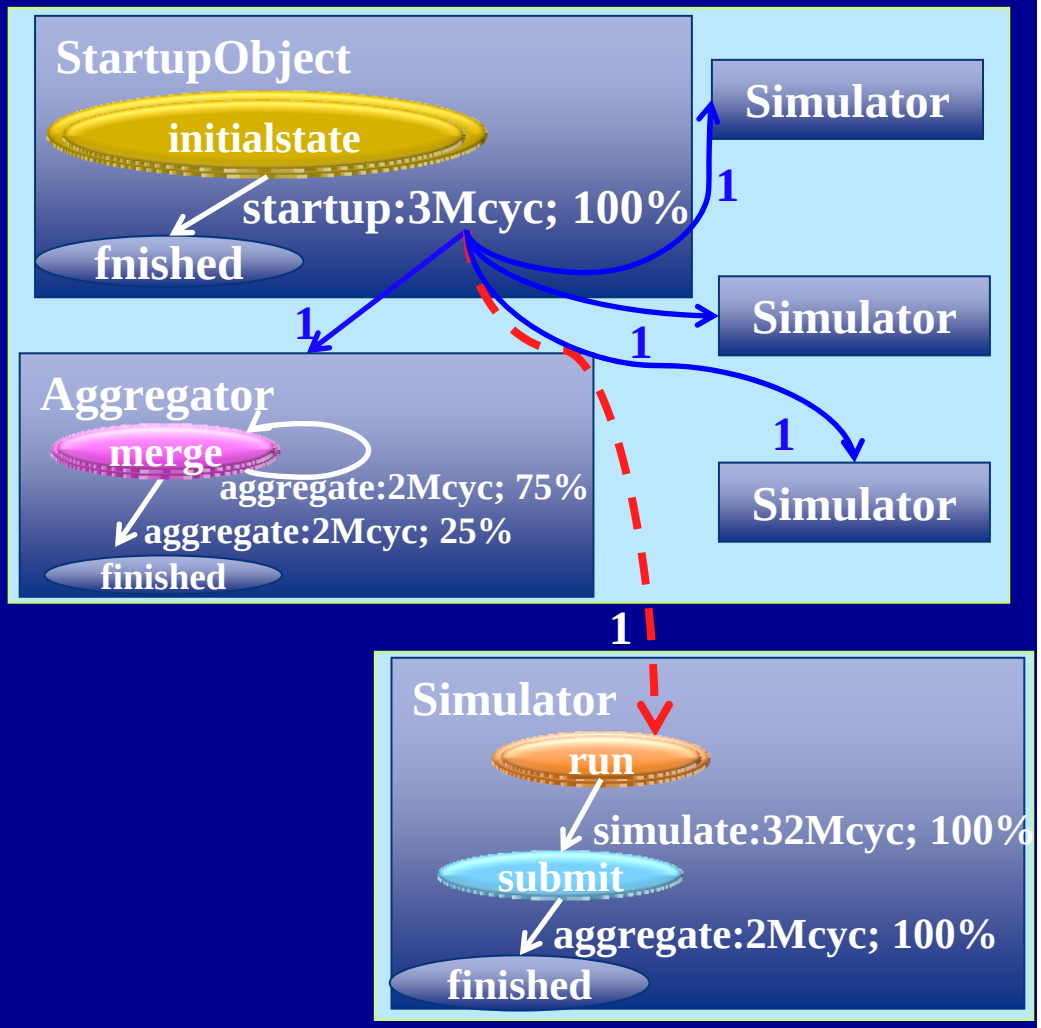
Simulation-Based Evaluation

- To select the best candidate implementation
- High-level simulation
  - Does NOT actually execute the program
  - Constructs abstract execution trace with similar statistics
  - Compare the execution time or throughput and core usage

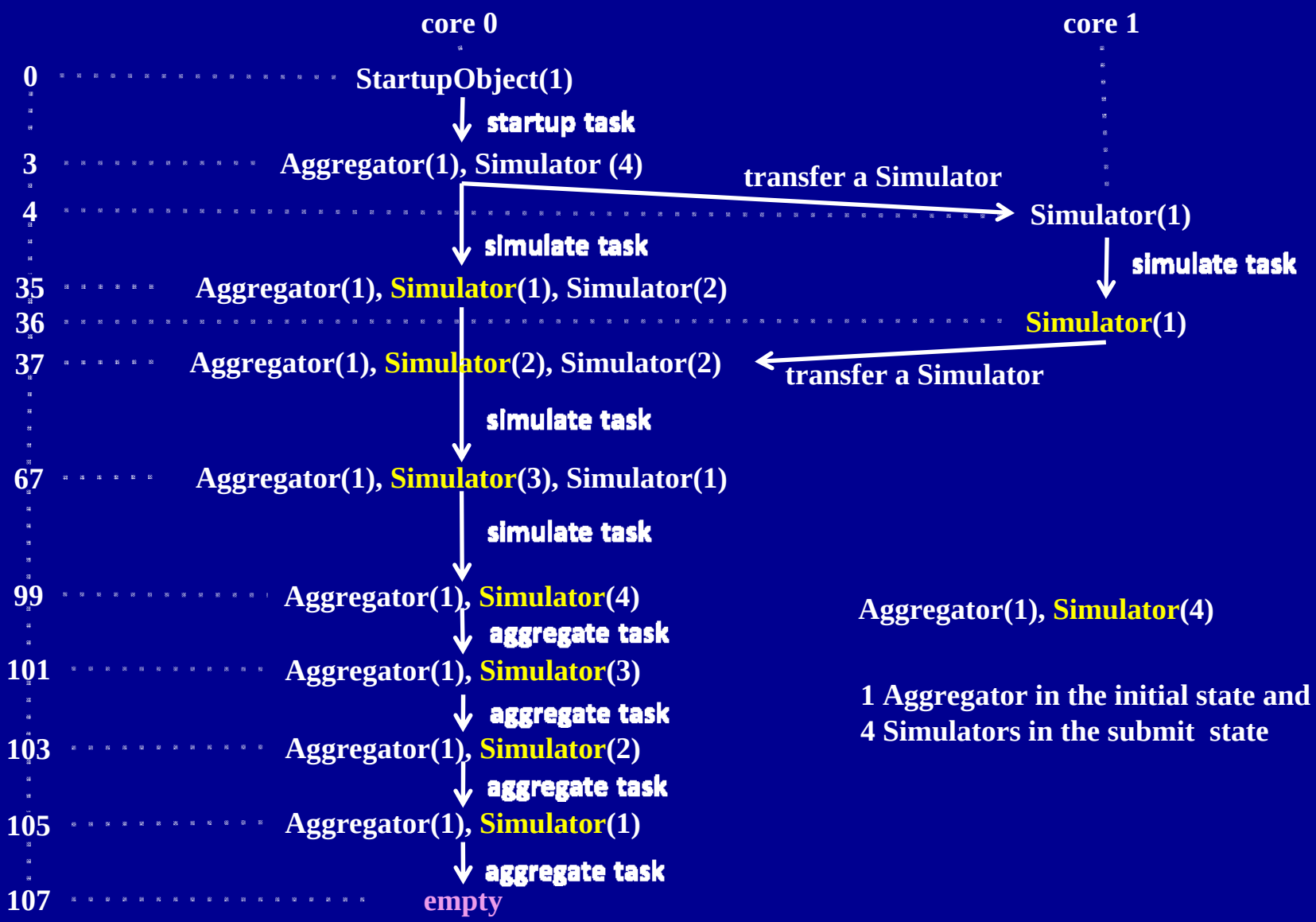


Simulation-Based Evaluation

- Markov model
  - Built from profile data
  - For each task estimates:
    - The destination state
    - The execution time
    - A count of each type of new objects



Simulated Execution Trace



Aggregator(1), Simulator(4)

1 Aggregator in the initial state and
4 Simulators in the submit state

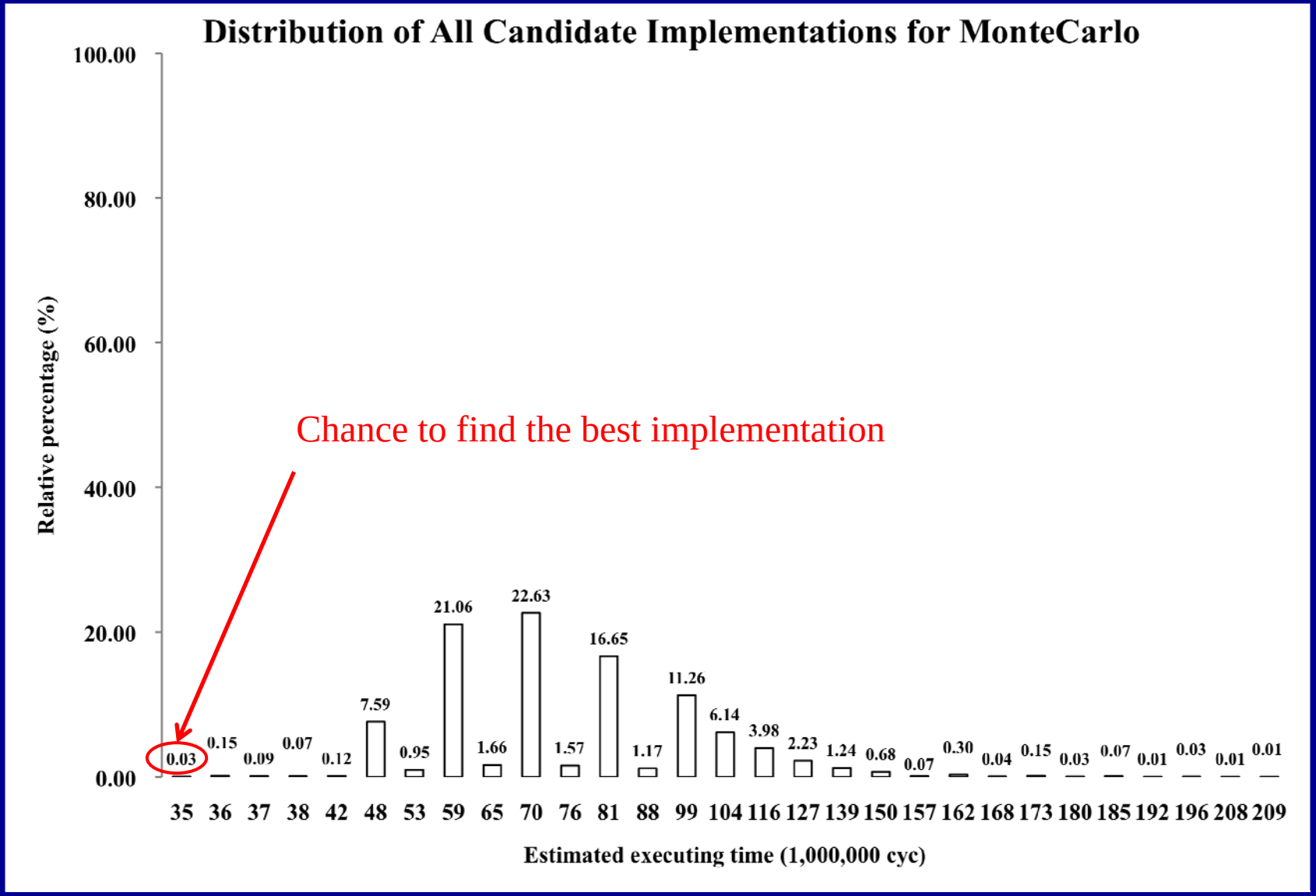
Problem of Exhaustive Searching

Number of CFSTG Core Groups	Number of Cores	Number of Candidates
32	16	> 6,000
64	32	> 14,000,000

- The search space expands quickly
- Exhaustive search is not feasible for complicated applications

Random Search?

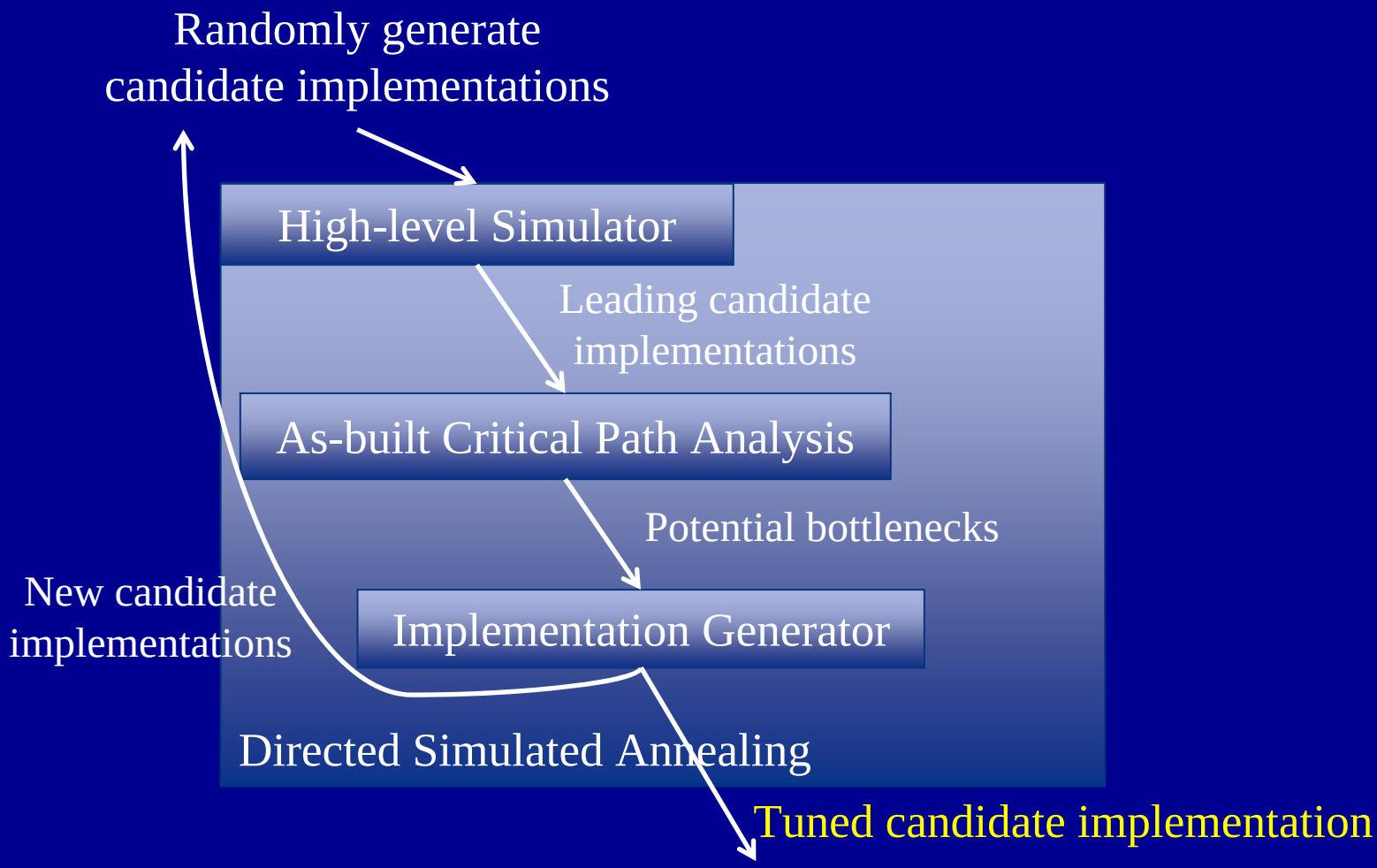
- Very low chance to find the best implementation



Developer Optimization Process

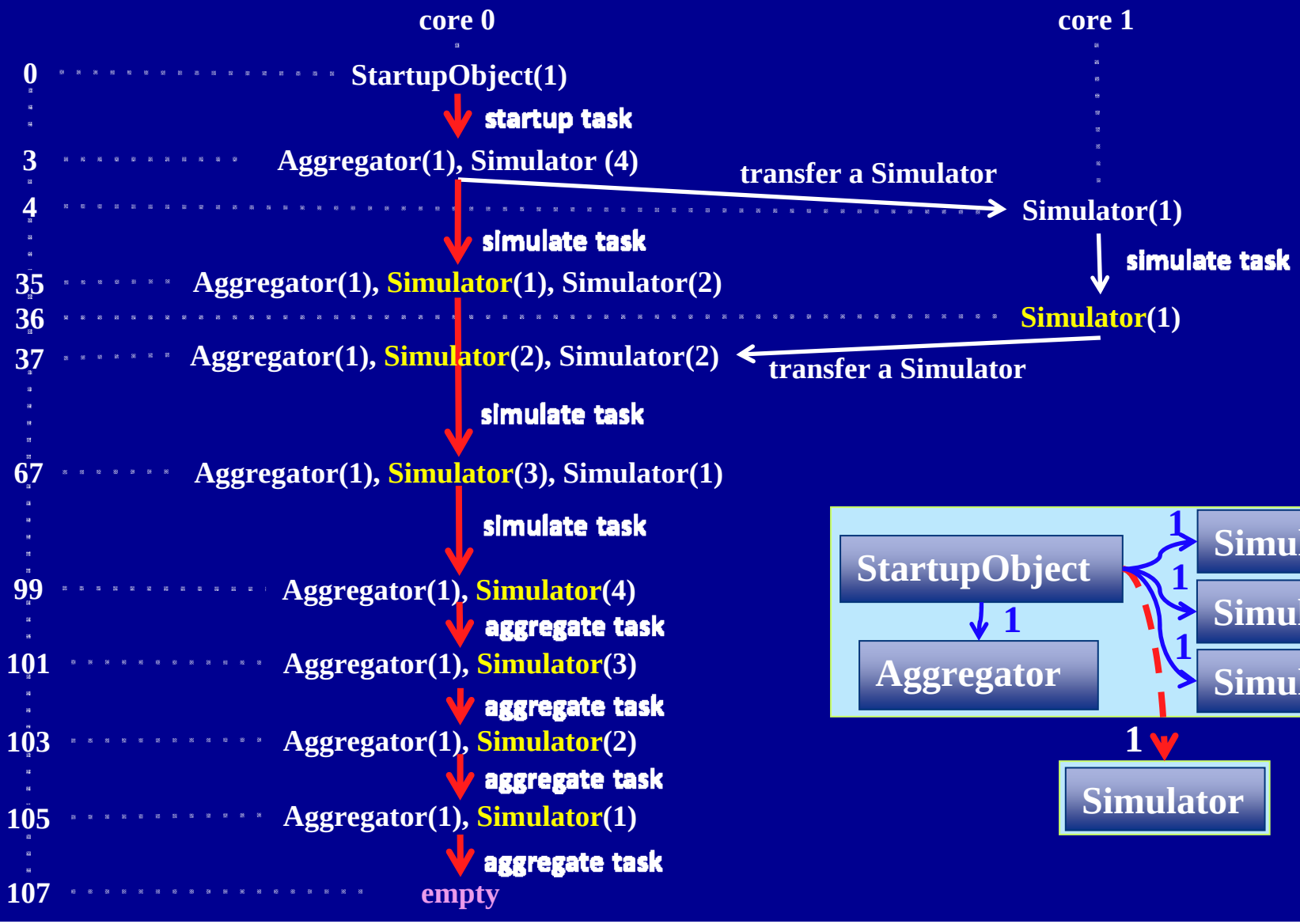
- Create an initial implementation
- Evaluate it and identify performance bottlenecks
- Heuristically develop new implementations to remove bottlenecks
- Iteratively repeat evaluation and optimization

Directed Simulated Annealing (DSA)

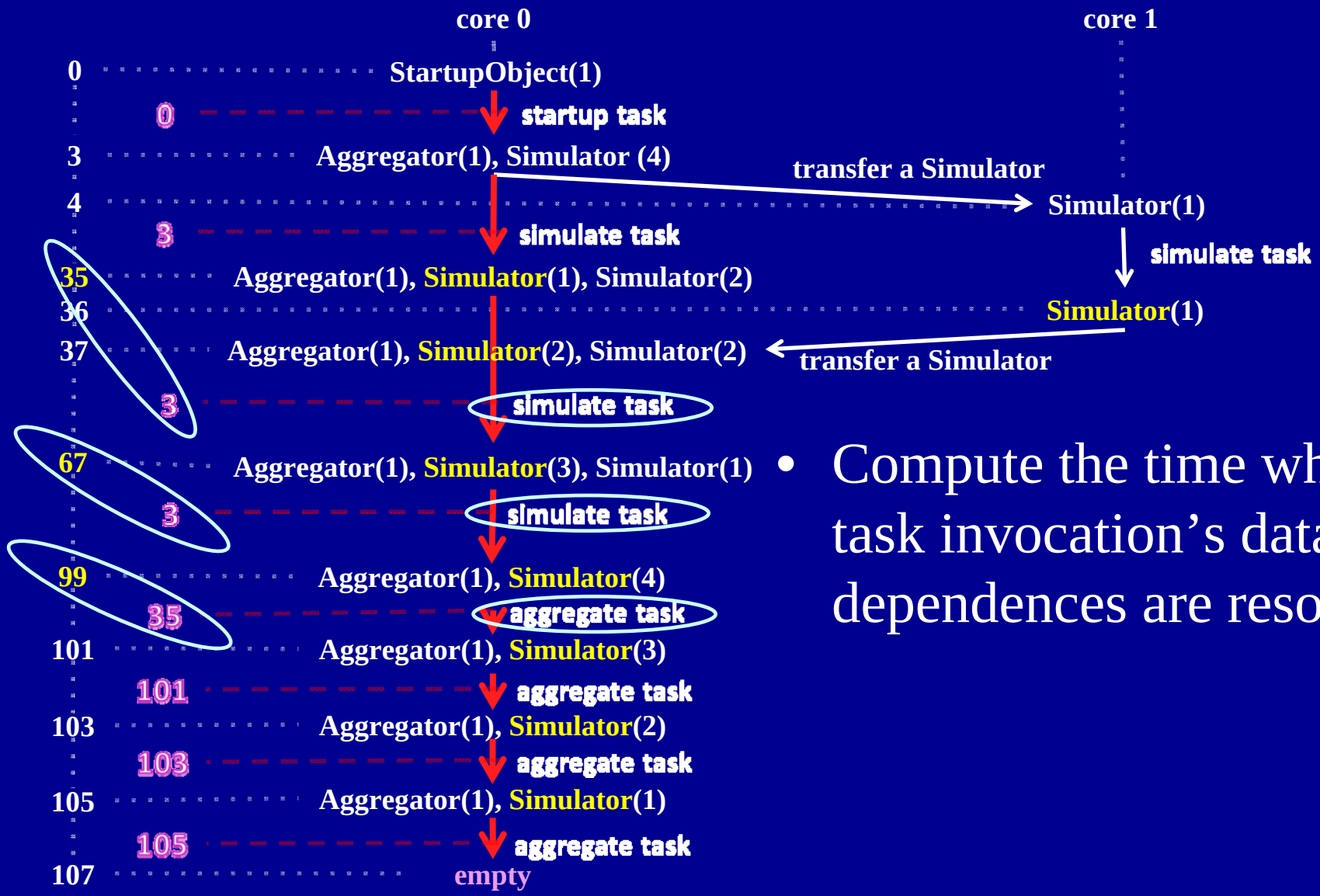


As-Built Critical Path (ABCP)

- Provide post-mortem analysis of project management



As-Built Critical Path Analysis



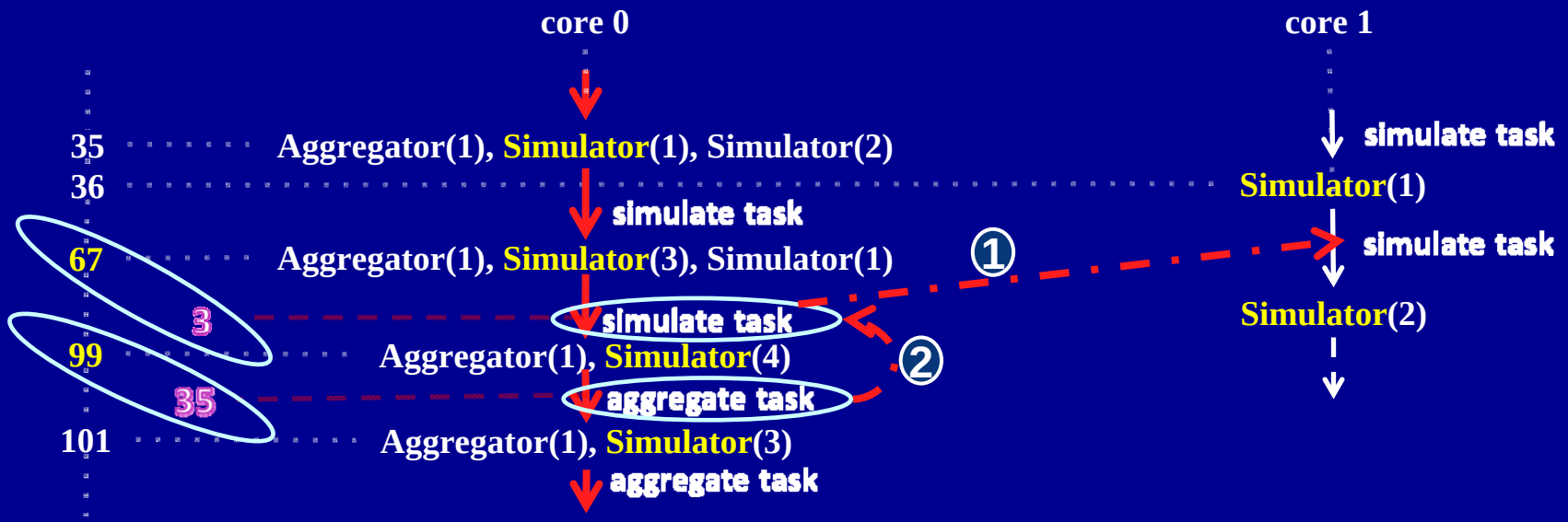
- Compute the time when a task invocation's data dependences are resolved

Waiting Task Optimization

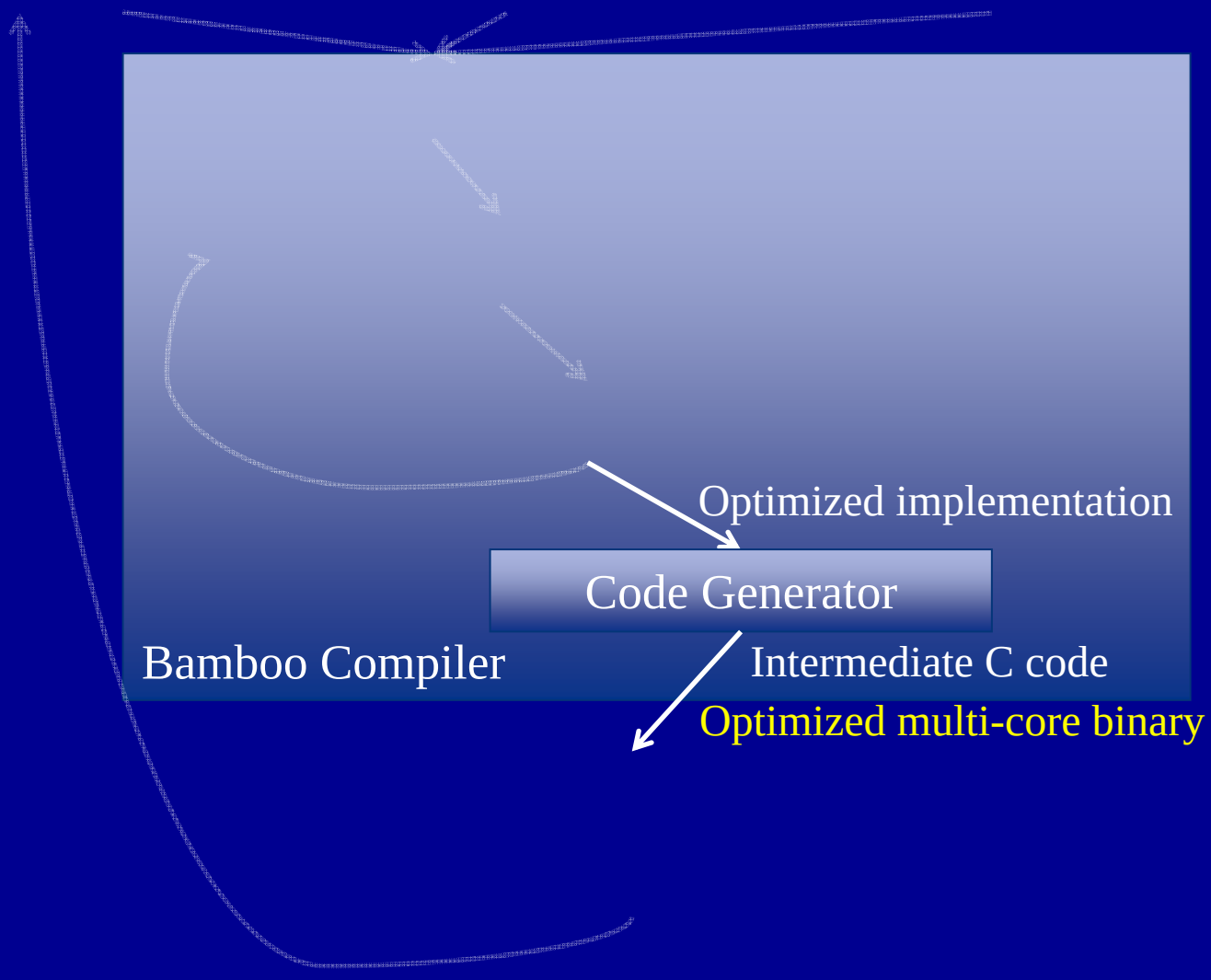
- Waiting tasks:
  - Tasks whose real invocation time is later than the time when all its data dependences are resolved
  - Delayed because of resource conflicts
  - Bottlenecks, remove them from ABCP
- Optimization
  - Migrate waiting tasks to spare cores
  - Shorten the ABCP to improve performance

Critical Task Optimization

- There may not exist spare cores to move waiting tasks to
- Identify critical tasks: tasks that produce data that is consumed immediately
- Attempt to execute critical tasks as early as possible
- Migrate other tasks which blocked some critical task to other cores



Code Generator



Evaluation

- MIT RAW simulator
  - Cycle accurate simulator configured for 16 cores
  - RAW chip: tiled chip, shared memory, on-chip network
- Benchmarks:
  - Series: Java Grande benchmark suite
  - MonteCarlo: Java Grande benchmark suite
  - FilterBank: StreamIt benchmark suite
  - Fractal

Speedups on 16 cores

- Successfully generated implementations with good performance

Benchmark	Clock Cycles (10^6 cyc)		Speedup to 1-Core Bamboo
	1-Core Bamboo	16-Core Bamboo	
Series	26.4	1.8	14.7
Fractal	38.4	3.3	11.6
MonteCarlo	191.7	19.0	10.1
FilterBank	91.2	6.7	13.6

Comparison to Hand-Written C Code

Benchmark	Clock Cycles (10^6 cyc)			Speedup to 1-Core C	Overhead of Bamboo
	1-Core C	1-Core Bamboo	16-Core Bamboo		
Series	25.0	26.4	1.8	13.9	5.6%
Fractal	36.2	38.4	3.3	11.0	6.1%
MonteCarlo	138.8	191.7	19.0	7.3	38.1%
FilterBank	71.1	91.2	6.7	10.6	28.3%

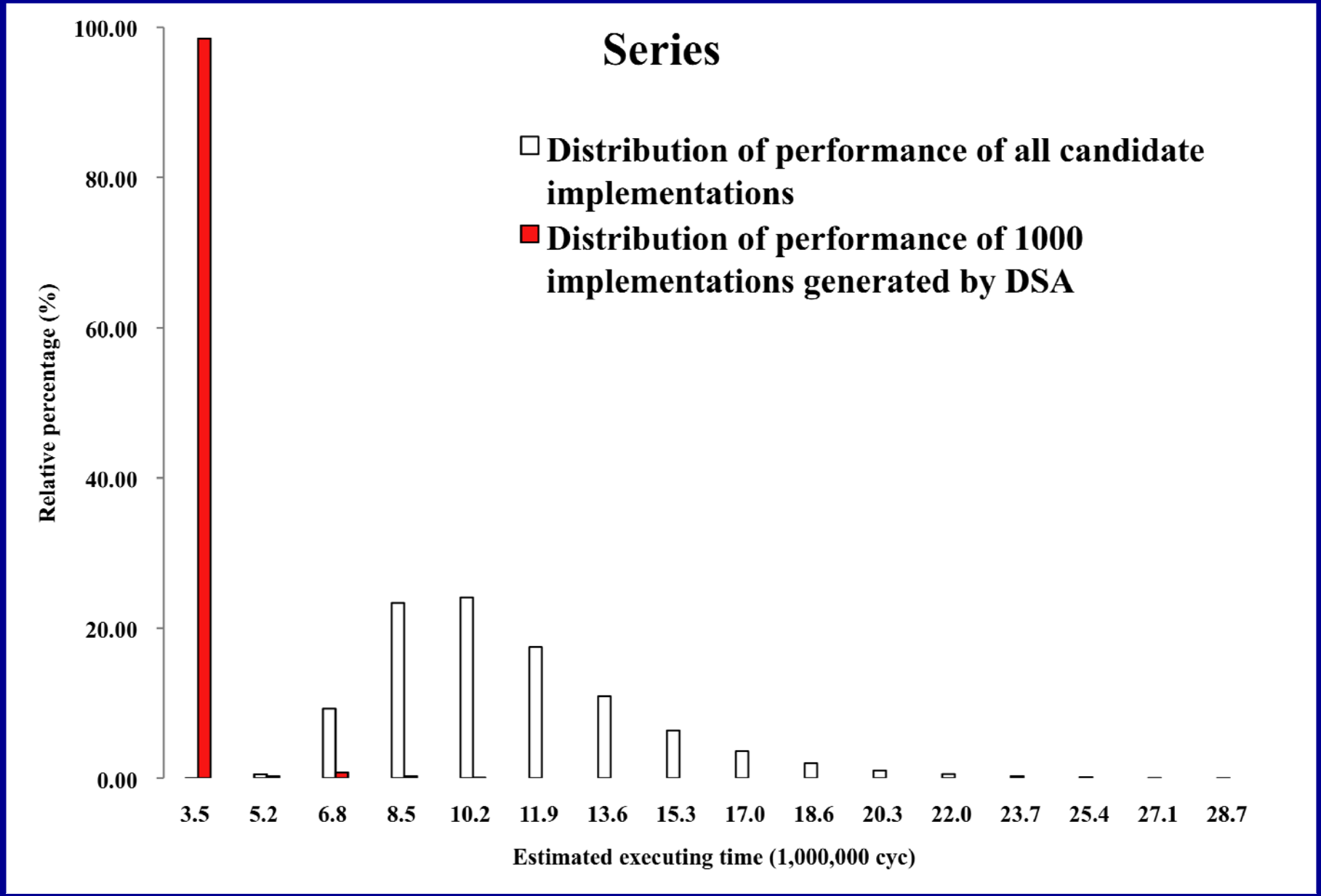
- Overhead of Bamboo:
  - Small for Series and Fractal
  - Larger overhead for MonteCarlo and FilterBank:
    - GCC cannot reorder instructions to fill floating-point delay slots for Bamboo implementations due to imprecise alias results
    - Easy to add alias information to facilitate the reordering

Comparison of Estimation and Real Execution

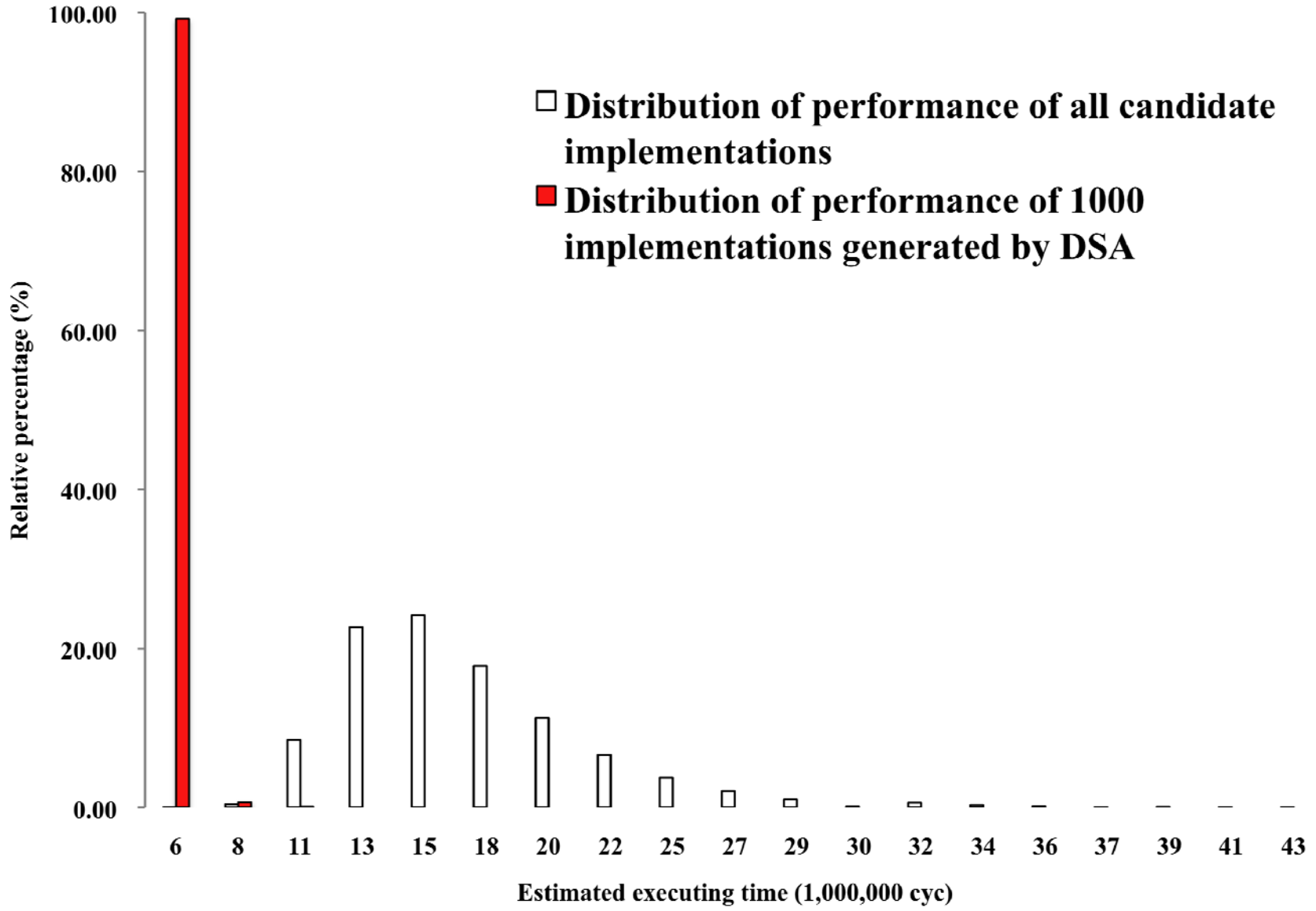
Benchmark	1-Core Bamboo Binary			16-Core Bamboo Binary		
	Clock Cycles (10^6 cyc)		Error	Clock Cycles (10^6 cyc)		Error
	Estimation	Real		Estimation	Real	
Series	26.3	26.4	0.38%	1.7	1.8	5.56%
Fractal	38.4	38.4	0%	3.1	3.3	6.06%
MonteCarlo	191.0	191.7	0.37%	18.3	19.0	3.68%
FilterBank	91.2	91.2	0%	6.5	6.7	2.99%

- The simulation estimations are close to the real execution time

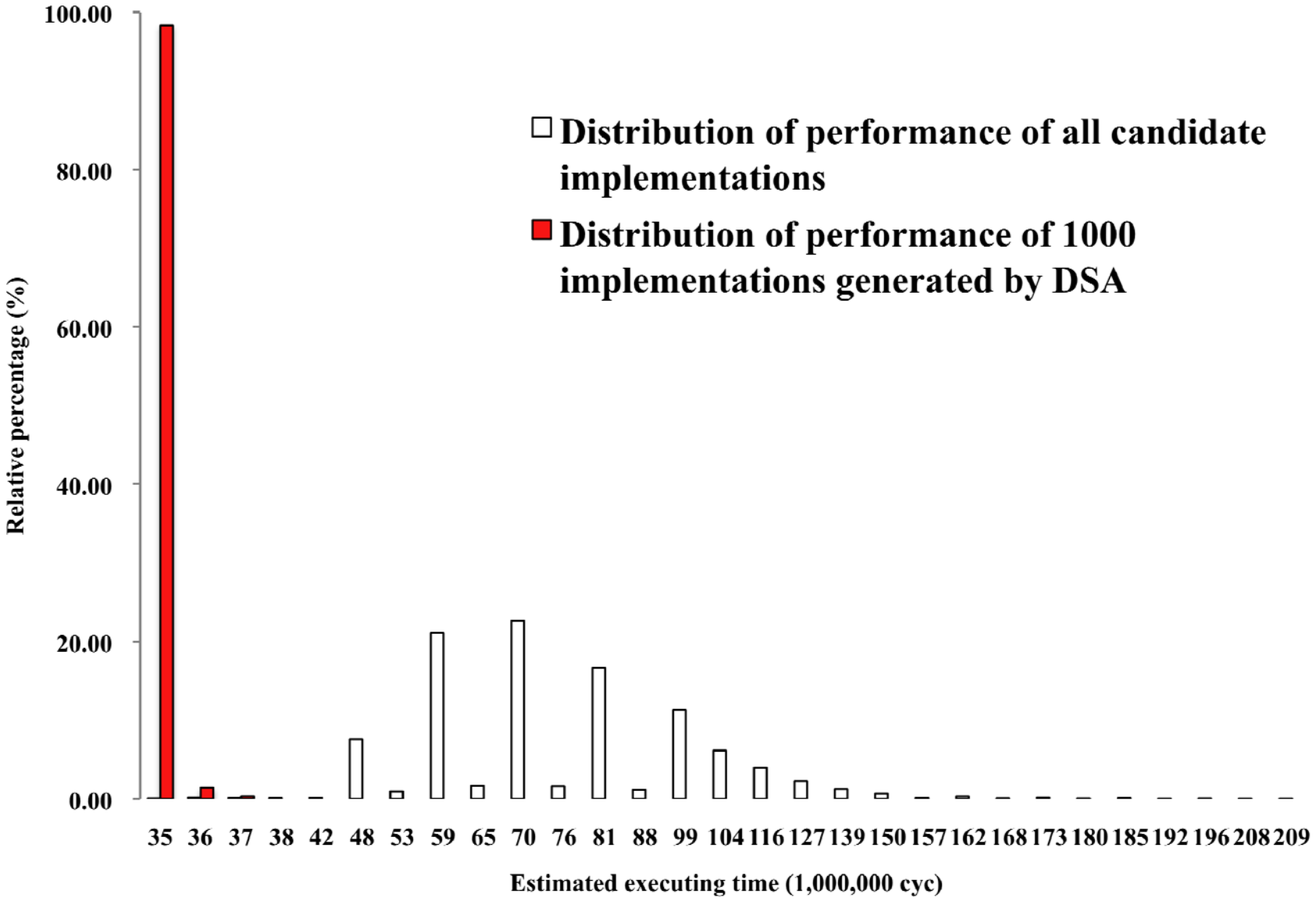
Optimality of Directed Simulated Annealing



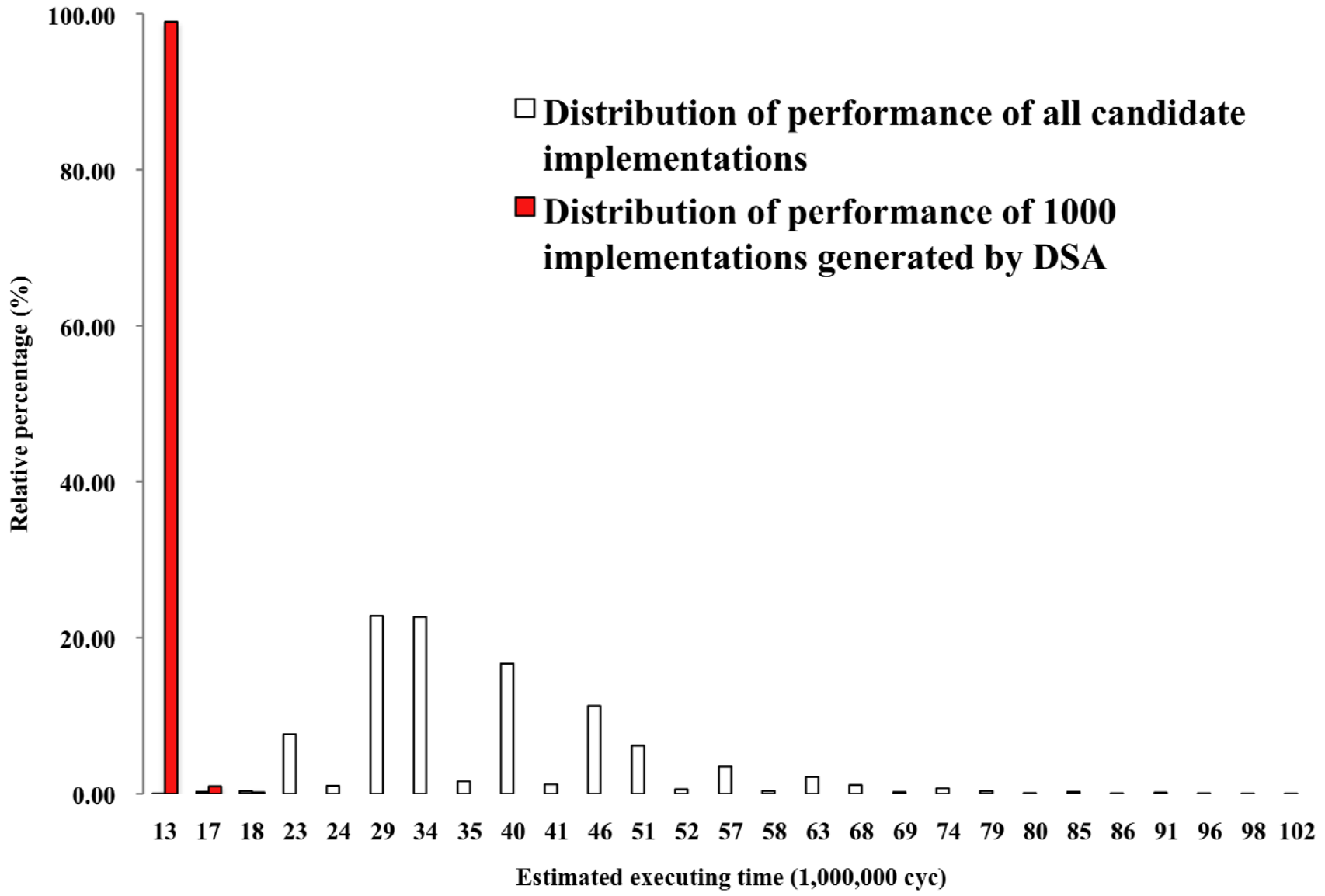
Fractal


100.00
80.00
60.00
40.00
20.00
0.00
Relative percentage (%)
Distribution of performance of all candidate implementations
Distribution of performance of 1000 implementations generated by DSA
6
8
11
13
15
18
20
22
25
27
29
30
32
34
36
37
39
41
43
Estimated executing time (1,000,000 cyc)

MonteCarlo


100.00
80.00
60.00
40.00
20.00
0.00
Relative percentage (%)
Distribution of performance of all candidate implementations
Distribution of performance of 1000 implementations generated by DSA
35 36 37 38 42 48 53 59 65 70 76 81 88 99 104 116 127 139 150 157 162 168 173 180 185 192 196 208 209
Estimated executing time (1,000,000 cyc)

FilterBank


100.00
80.00
60.00
40.00
20.00
0.00
Relative percentage (%)
Distribution of performance of all candidate implementations
Distribution of performance of 1000 implementations generated by DSA
13 17 18 23 24 29 34 35 40 41 46 51 52 57 58 63 68 69 74 79 80 85 86 91 96 98 102
Estimated executing time (1,000,000 cyc)

Generality of Synthesized Implementation

Benchmark	Profile_original, Input_double			Profile_double, Input_double	
	Clock Cycles (10^6 cyc)		Speedup	Clock Cycles (10^6 cyc)	Speedup
	1-Core	16-Core		16-Core	
Series	54.2	3.6	15.1	3.6	15.1
Fractal	76.6	6.5	11.8	6.5	11.8
MonteCarlo	383.2	37.8	10.1	35.7	10.7
FilterBank	182.3	13.3	13.7	13.3	13.7

- The speedups of both 16-core Bamboo versions are similar
- Successfully generate a sophisticated implementation utilizing pipelining for MonteCarlo

Related Work

- Data-flow and streaming languages:
  - Bamboo relaxes typical restrictions in these models to permit:
    - Flexible mutation of data structures
    - Data structures of arbitrarily complex constructs
  - Bamboo supports applications that non-deterministically access data
- Tuple-space language: compiler cannot automatically create multiple instantiations to utilize multiple cores
- Self-tuning libraries: mostly address specific computations

Conclusion

- We developed a new approach to automatically tune task-based programs for multi-core processors
  - Automatically generate parallel implementations
  - Automatically tune according to specific architecture
- The approach was evaluated on MIT RAW simulator
  - Successfully generated implementations with good performance
  - Successfully generated a sophisticated implementation utilizing pipelining
- Can be extended to the broader context of traditional programming languages

Thank you!