Autotuning of a Cut-off for Task Parallel Programs

Shintaro Iwasaki, Kenjiro Taura
Graduate School of Information Science and Technology
The University of Tokyo

September 22, 2016
@ ATMG '16 (special session in MCSoC '16)

Short Summary

- We focus on a fork-join task parallel programming model.

Keyword: divide-and-conquer

- “Cut-off” is an optimization technique for task parallel programs to control granularity.
- We had developed a series of compiler optimization techniques for automatic cut-off (“static cut-off”[*])

[*] Iwasaki et al., A Static Cut-off for Task Parallel Programs, PACT '16, 2016.

Short Summary

- We focus on a fork-join task parallel programming model.

Keyword: divide-and-conquer

- "Cut-off" is an optimization technique for task parallel programs to control granularity.
- We had developed a series of compiler optimization techniques for automatic cut-off ("static cut-off"[*])
- This study proposes an automatic cut-off technique with an autotuning method to obtain the best combination of these techniques for higher performance.

[*] Iwasaki et al., A Static Cut-off for Task Parallel Programs, PACT '16, 2016.

Index

Index

Importance of Multi-threading

- The number of CPU cores gets larger and larger.

Intel Xeon Phi (Knights Corner) is a typical example:
it has 60 cores, supporting over 200 hardware threads.
http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html

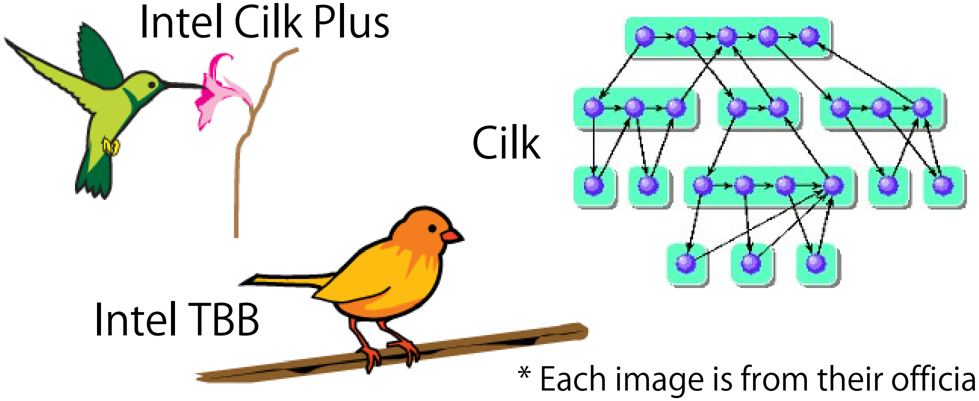
We didn't use it for evaluation, though.

- Multi-threading is essential to exploiting modern processors.

  → A task parallel model is one of the most promising parallel programming models.

Task Parallel Programming Models

- Task parallelism is a popular parallel programming model.
  - Adopted by many famous systems/libraries:
    - e.g., OpenMP (since ver. 3.0), Cilk / Cilk Plus, Intel TBB ⋯



* Each image is from their official pages.

- It has two major features:
  - Dynamic load balancing
  - Suitability for divide-and-conquer algorithms
- In this talk, we focus on a “fork-join task parallel model.”

Fork-join Task Parallelism

- We use program examples given in Cilk syntax.
- Two basic keywords are provided to express task parallelism: *spawn* and *sync*.
  - Spawn (≒ fork) : create a task as a child, which will be executed concurrently.
  - Sync (≒ join) : wait all tasks created (or spawned) by itself.

```
void vecadd(float* a, float* b, int n){
  for(int i = 0; i < n; i++)
    a[i] += b[i];
}
```

Same meaning.

```
void vecadd(float* a, float* b, int n){
  if(n == 1){
    *a += *b;
  }else{
    spawn vecadd(a, b, n/2);
    spawn vecadd(a+n/2, b+n/2, n-n/2);
    sync;
}
```

Fork-join Task Parallelism

- We use program examples given in Cilk syntax.
- Two basic keywords are provided to express task parallelism: *spawn* and *sync*.
  - Spawn (≒ fork) : create a task as a child, which will be executed concurrently.
  - Sync (≒ join) : wait all tasks created (or spawned) by itself.
- The main target is a divide-and-conquer algorithm.
  - e.g., sort, FFT, FMM, AMR, cache-oblivious GEMM

```
void vecadd(float* a, float* b, int n){
  if(n == 1){
    *a += *b;
  }else{
    spawn vecadd(a, b, n/2);
    spawn vecadd(a+n/2, b+n/2, n-n/2);
    sync;
  }
}
```

Overheads of Task Parallel Program

- In general, task parallel runtime is designed to handle fine-grained parallelism efficiently.
- However, extreme fine granularity imposes large overheads, degrading the performance.

This vecadd is a too fine-grained task; one leaf task only calculates `*a += *b`.

```
void vecadd(float* a, float* b, int n){
  if(n == 1){
    *a += *b;
  }else{
    spawn vecadd(a, b, n/2);
    spawn vecadd(a+n/2, b+n/2, n-n/2);
    sync;
  }
}
```

Overheads of Task Parallel Program

- In general, task parallel runtime is designed to handle fine-grained parallelism efficiently.
- However, extreme fine granularity imposes large overheads, degrading the performance.

This vecadd is a too fine-grained task; one leaf task only calculates `*a += *b`.

```
void vecadd(float* a, float* b, int n){
  if(n == 1){
    *a += *b;
  }else{
    spawn vecadd(a, b, n/2);
    spawn vecadd(a+n/2, b+n/2, n-n/2);
    sync;
  }
}
```

- Cut-off has been known as an effective optimization technique.

Cut-off: An Optimization Technique

- Cut-off is a technique to reduce a tasking overhead by stop creating tasks in a certain condition.
  - i.e., execute a task in serial in that condition.

```
void vecadd(float* a, float* b, int n){
  if(n == 1){
    *a += *b;
  }else{
    spawn vecadd(a, b, n/2);
    spawn vecadd(a+n/2, b+n/2, n-n/2);
    sync;
  }
}
```

Cut-off

```
void vecadd(float* a, float* b, int n){
  if(1<= n && n <=1000){
    vecadd_seq(a, b, n
  }else{
    spawn vecadd(a, b, n/2);
    spawn vecadd(a+n/2, b+n/2, n-n/2);
    sync;
  }
}
//Sequential version of vecadd
void vecadd_seq(float* a, float* b, int n){
  if(n == 1){
    *a += *b;
  }else{
    /*spawn*/vecadd_seq(a, b, n/2);
    /*spawn*/vecadd_seq(a+n/2, b+n/2, n-n/2);
    /*sync;*/
  }
}
```

A cut-off condition

Call a sequential vecadd if `1 <= n && n <= 1000`

- Programmers commonly apply it manually.

Cut-off + Further Optimizations

```
void vecadd(float* a, float* b, int n){
  if(n == 1){
    *a += *b;
  }else{
    spawn vecadd(a, b, n/2);
    spawn vecadd(a+n/2, b+n/2, n-n/2);
    sync;
  }
}
```

1. Cut-off

```
void vecadd(float* a, float* b, int n){
  if(1 <= n && n <= 4096){
    vecadd_seq(a, b, n);
  }else{
    spawn vecadd(a, b, n/2);
    spawn vecadd(a+n/2, b+n/2, n-n/2);
    sync;
  }
}
void vecadd_seq(float* a, float* b, int n){
  for(int i = 0; i < n; i++)
    a[i] += b[i];
}
```

2. Transformation

vecadd_seq() is loopified.

- In addition to reducing tasking overheads, further transformations are applicable to serialized tasks in some cases.

Related Work

Dynamic Cut-off

- Most previous studies on automatic cut-off [*1,*2,*3] focused on adaptive cut-off (dynamic cut-off)
  - Dynamic cut-off is a technique not creating tasks when runtime information tells task creation is not beneficial.
    - Runtime information:
      a total number of tasks, task queue length, execution time, depth of tasks, frequency of work stealing etc...
- Problems:
  Cost to evaluate a cut-off condition is large,
  Optimizations after the cut-off are less applicable.

Dynamic cut-off advantage:
wider applicable range.

[*1] Bi et al. An Adaptive Task Granularity Based Scheduling for Task-centric Parallelism, HPCC '14, 2014

[*2] Duran et al. An Adaptive Cut-offfor Task Parallelism, SC '08, 2008

[*3] Thoman et al. Adaptive Granularity Control in Task Parallel Programs Using Multiversioning, Euro-Par'13, 2013

Our Goal: Automatic Cut-off

- Our goal is developing automatic cut-off including further optimizations automatically for task parallel programs without any manual cut-off.

```
void vecadd(float* a, float* b, int n){
  if(1 <= n && n <= 4096){
    vecadd_seq(a, b, n);
  }else{
    spawn vecadd(a, b, n/2);
    spawn vecadd(a+n/2, b+n/2, n-n/2);
    sync;
  }
}
void vecadd_seq(float* a, float* b, int n){
  // Vectorize the following for-loop,
  // since task keywords implicitly reveal
  // each iteration is independent.
  for(int i = 0; i < n; i++)
    a[i] += b[i];
}
```

THE UNIVERSITY OF TOKYO

Our Goal: Automatic Cut-off

- Our goal is developing automatic cut-off including further optimizations automatically for task parallel programs without any manual cut-off.

Let's say divide-until-trivial task parallel programs.

  - Compiler optimizations for simple loops have been well developed.
    - Loop blocking, unrolling interchange, etc...

→ Develop optimizations for divide-until-trivial tasks.

```
void vecadd(float* a, float* b, int n){
  if(1 <= n && n <= 4096){
    vecadd_seq(a, b, n);
  }else{
    spawn vecadd(a, b, n/2);
    spawn vecadd(a+n/2, b+n/2, n-n/2);
    sync;
  }
}
void vecadd_seq(float* a, float* b, int n){
  // Vectorize the following for-loop,
  // since task keywords implicitly reveal
  // each iteration is independent.
  for(int i = 0; i < n; i++)
    a[i] += b[i];
}
```

Index

0. Short Summary
1. Introduction
2. Static Cut-off and its Limitations
   - Our previous work: static cut-off
   - Limitations
3. Our Proposal: Cut-off with Autotuning
4. Evaluation
5. Conclusion

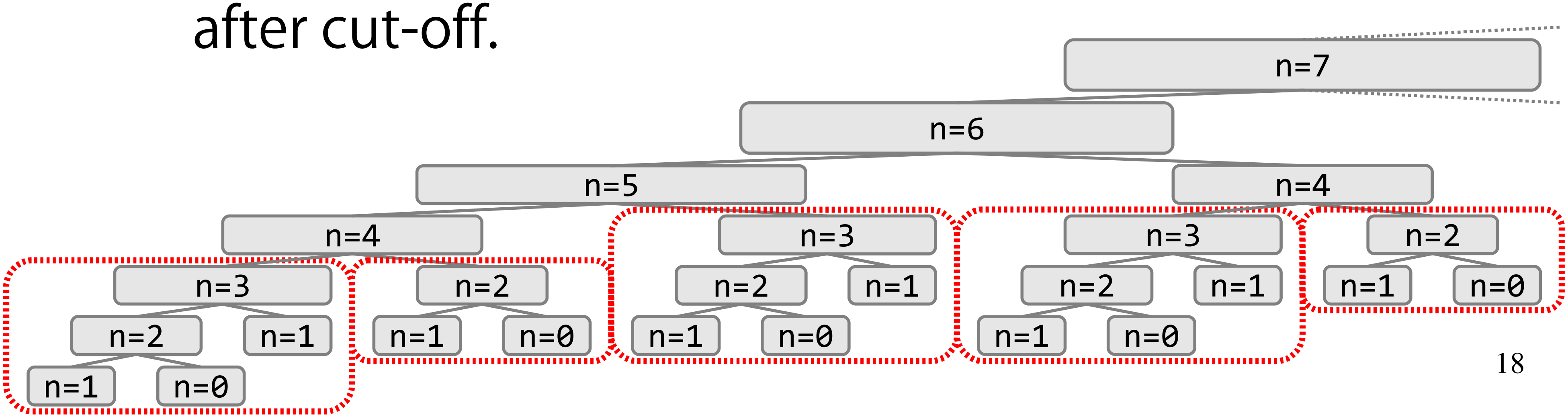
What we've proposed: Static Cut-off

- Static cut-off is an automatic cut-off method including a series of compile-time optimization techniques for task parallel programs.

We proposed it in PACT '16 [*].

- It tries to aggregate tasks near leaves.
  - \+ Low risk of serious loss of parallelism.
  - \+ Chance to apply powerful compiler optimizations after cut-off.

Encircled by ⬚



[*] Iwasaki et al., A Static Cut-off for Task Parallel Programs, PACT '16, 2016.

What we've proposed: Static Cut-off

- Static cut-off is an automatic cut-off method including a series of compile-time optimization techniques for task parallel programs.

We proposed it in PACT '16 [*].

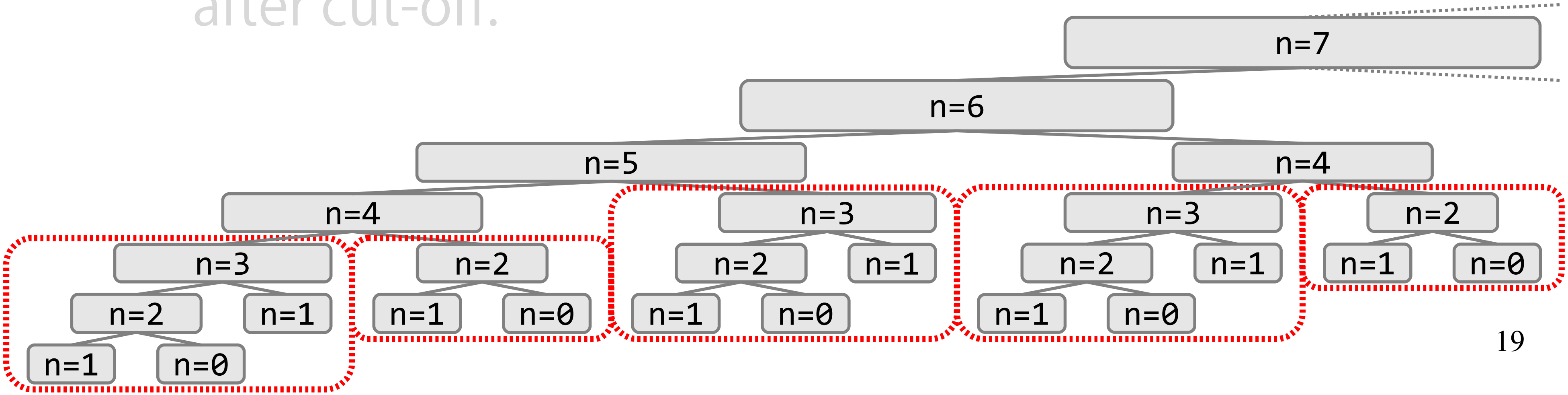
- It tries to aggregate tasks near leaves.

Encircled by

    + Low

Key idea: use a height instead of a depth.

    + Chance to apply powerful compiler optimizations after cut-off.



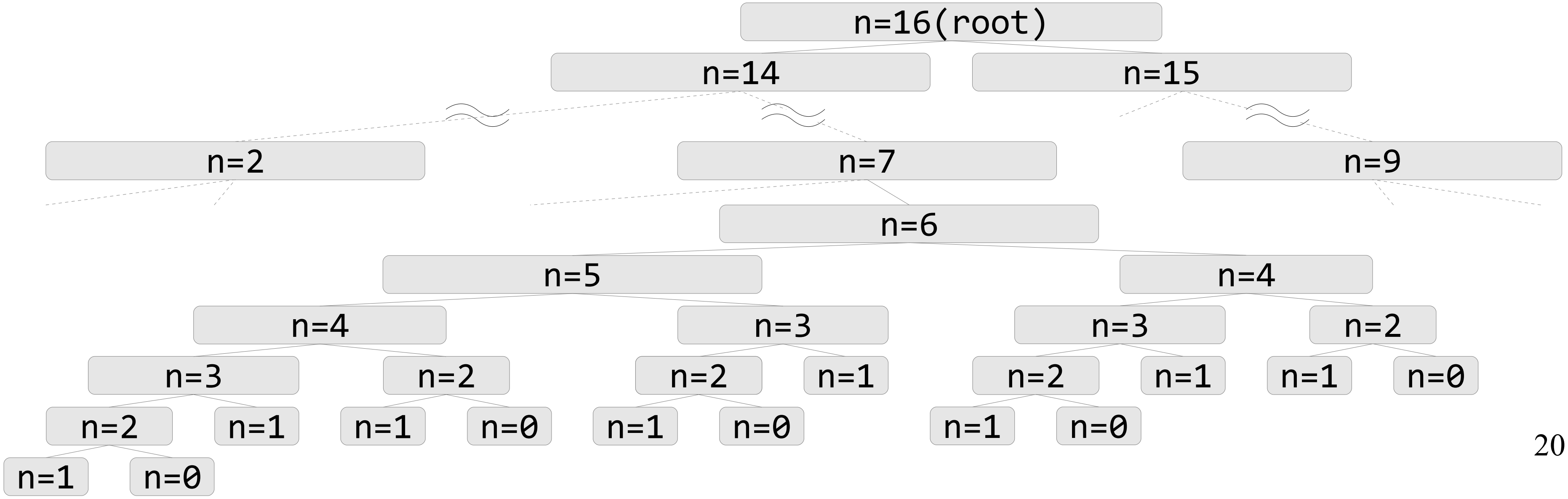
[*] Iwasaki et al., A Static Cut-off for Task Parallel Programs, PACT '16, 2016.

Depth/Height of Tasks

- Consider a task tree of fib(16) below.

fib calculates $F_n = \begin{cases} n & \text{if } n < 2 \\ F_{n-1} + F_{n-2} & \text{otherwise} \end{cases}$

```
void fib(int n, int* r){
  if(n < 2){
    *r = n;
  }else{
    int a, b;
    spawn fib(n-1, &a);
    spawn fib(n-2, &b);
    sync;
    *r = a + b;
  }
}
```



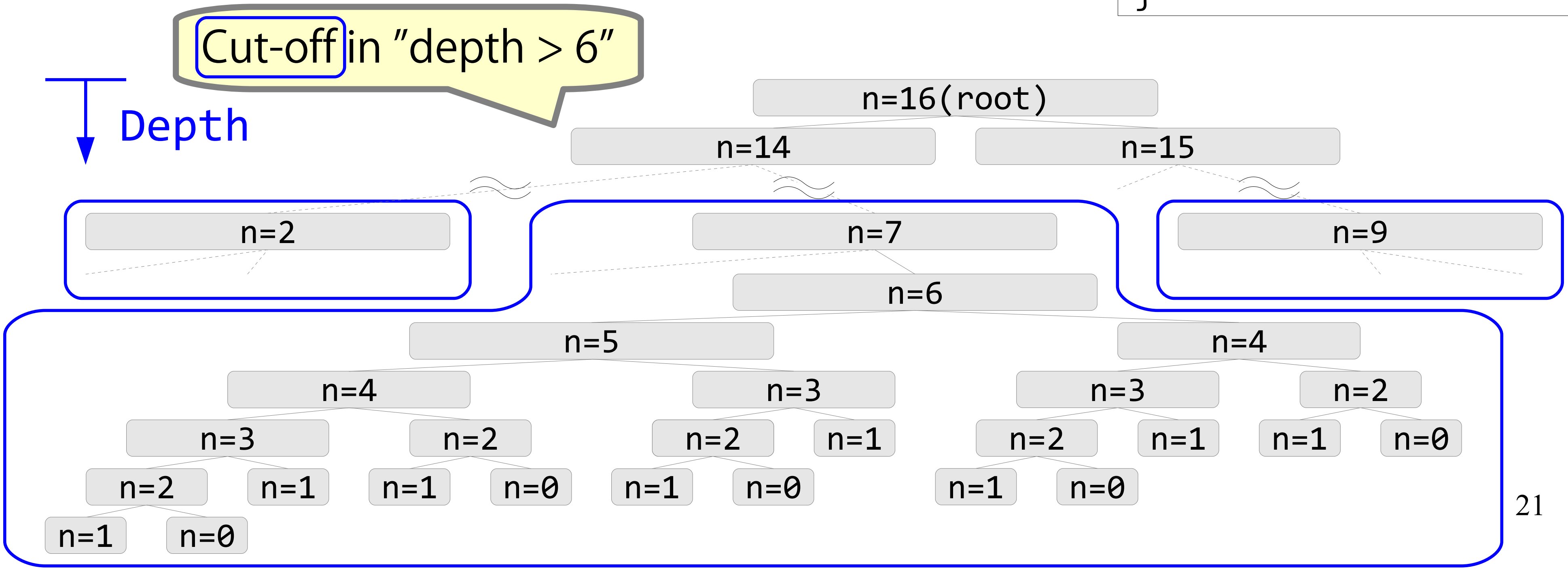
Depth/Height of Tasks

- Consider a task tree of fib(16) below.

fib calculates $F_n = \begin{cases} n & \text{if } n < 2 \\ F_{n-1} + F_{n-2} & \text{otherwise} \end{cases}$

  - Depth is easy to obtain.
    - e.g., increment a variable from the root.

```
void fib(int n, int* r){
  if(n < 2){
    *r = n;
  }else{
    int a, b;
    spawn fib(n-1, &a);
    spawn fib(n-2, &b);
    sync;
    *r = a + b;
  }
}
```



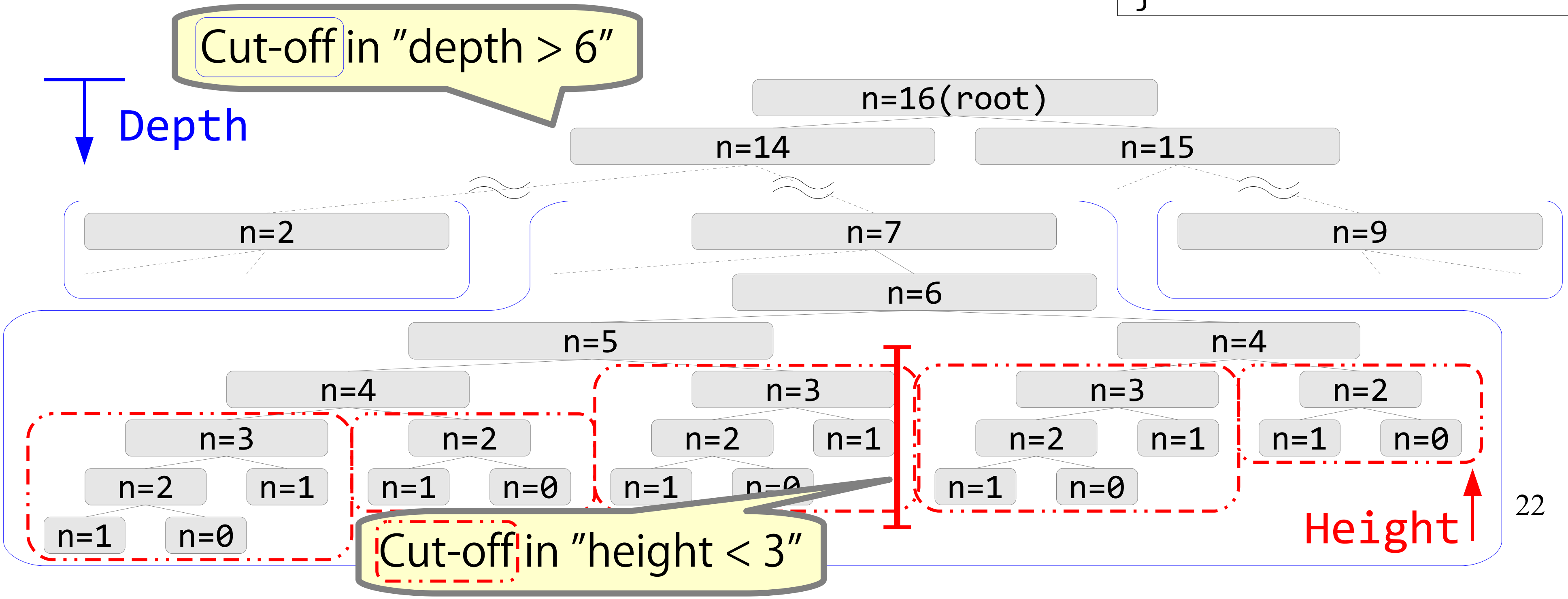
Depth/Height of Tasks

- Consider a task tree of fib(16) below.

  fib calculates $F_n = \begin{cases} n & \text{if } n < 2 \\ F_{n-1} + F_{n-2} & \text{otherwise} \end{cases}$

  - Height is difficult to calculate, but it is suitable for a cut-off condition.

```
void fib(int n, int* r){
  if(n < 2){
    *r = n;
  }else{
    int a, b;
    spawn fib(n-1, &a);
    spawn fib(n-2, &b);
    sync;
    *r = a + b;
  }
}
```



Cut-off in "depth > 6"

Cut-off in "height < 3"

Static Cut-off Flow

1. Try to calculate a height-based cut-off condition.

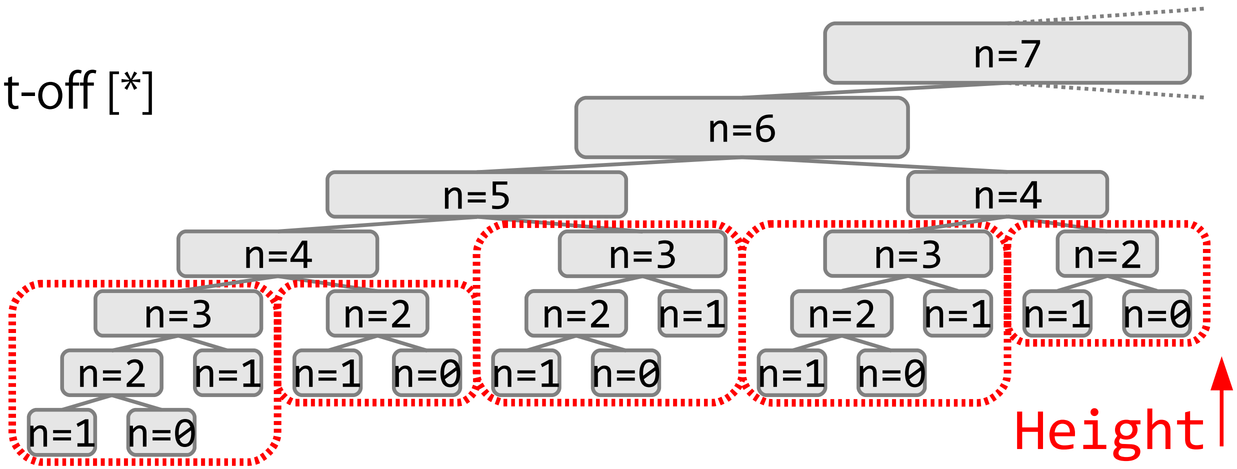
- If the height-based cut-off condition is calculable ...

  2. Decide a height parameter H.
  3. Apply one of the following:
     - Static task elimination
     - Code-bloat-free inlining
     - Loopification

Show the examples later.

- Otherwise...

  2. Apply the dynamic cut-off [*]



[*] P. Thoman et al. Adaptive granularity control in task parallel programs using multiversioning. Euro-Par '13, 2013

Examples of Static Cut-off

2. Decide a height parameter H.

Use heuristics.

3. Apply one of the following:

- Static task elimination
- Code-bloat-free inlining
- Loopification

```
void vecadd(float* a, float* b, int n){
  if(n == 1){
    *a += *b;
  }else{
    spawn vecadd(a, b, n/2);
    spawn vecadd(a+n/2, b+n/2, n-n/2);
    sync;
  }
}
```

H = 10 in this case.

```
void vecadd(float* a, float* b, int n){
  if(1 <= n && n <= 1024){
    vecadd_seq(a, b, n);
  }else{
    spawn vecadd(a, b, n/2);
    spawn vecadd(a+n/2, b+n/2, n-n/2);
    sync;
  }
}
void vecadd_seq(float* a, float* b, int n){
  ???
}
```

Examples of Static Cut-off

2. Decide a height parameter H.

3. Apply one of the following:

- Static task elimination
- Code-bloat-free inlining
- Loopification

```
void vecadd(float* a, float* b, int n){
  if(n == 1){
    *a += *b;
  }else{
    spawn vecadd(a, b, n/2);
    spawn vecadd(a+n/2, b+n/2, n-n/2);
    sync;
  }
}
```

Just remove spawn & sync
to reduce the overheads.

```
void vecadd(float* a, float* b, int n){
  if(1 <= n && n <= 1024){
    vecadd_seq(a, b ,n);
  }else{
    spawn vecadd(a, b, n/2);
    spawn vecadd(a+n/2, b+n/2, n-n/2);
    sync;
  }
}
void vecadd_seq(float* a, float* b, int n){
  if(n == 1){
    *a += *b;
  }else{
    /*spawn*/vecadd_seq(a, b, n/2);
    /*spawn*/vecadd_seq(a+n/2, b+n/2, n-n/2);
    /*sync;*/
  }
}
```

Examples of Static Cut-off

2. Decide a height parameter H.
3. Apply one of the following:
   - Static task elimination
   - Code-bloat-free inlining
   - Loopification

```
void vecadd(float* a, float* b, int n){
  if(n == 1){
    *a += *b;
  }else{
    spawn vecadd(a, b, n/2);
    spawn vecadd(a+n/2, b+n/2, n-n/2);
    sync;
  }
}
```

Apply inlining to reduce function call overheads w/o exponential code growth.

```
void vecadd(float* a, float* b, int n){
  if(1 <= n && n <= 1024){
    vecadd_seq(a, b ,n);
  }else{
    spawn vecadd(a, b, n/2);
    spawn vecadd(a+n/2, b+n/2, n-n/2);
    sync;
  }
}
void vecadd_seq(float* a, float* b, int n){
  if(n == 1){
    *a += *b;
  }else{
    for(int i = 0; i < 2; i++){
      float *a2, *b2; int n2;
      switch(i){
      case 0:
        a2=a;     b2=b    ; n2=n/2;   break;
      case 1:
        a2=a+n/2; b2=b+n/2; n2=n-n/2; break;
      }
      //Inline 10 times here.
      vecadd_seq(a2,b2,n2);
    }
  }
}
```

Examples of Static Cut-off

2. Decide a height parameter H.

3. Apply one of the following:

- Static task elimination
- Code-bloat-free inlining
- Loopification

```
void vecadd(float* a, float* b, int n){
  if(n == 1){
    *a += *b;
  }else{
    spawn vecadd(a, b, n/2);
    spawn vecadd(a+n/2, b+n/2, n-n/2);
    sync;
  }
}
```

```
void vecadd(float* a, float* b, int n){
  if(1 <= n && n <= 1024){
    vecadd_seq(a, b ,n);
  }else{
    spawn vecadd(a, b, n/2);
    spawn vecadd(a+n/2, b+n/2, n-n/2);
    sync;
  }
}
void vecadd_seq(float* a, float* b, int n){
  for(int i=0; i<n; i++)
    a[i] += b[i];
}
```

Simplify the control flow
and also promote vectorization.

THE UNIVERSITY OF TOKYO

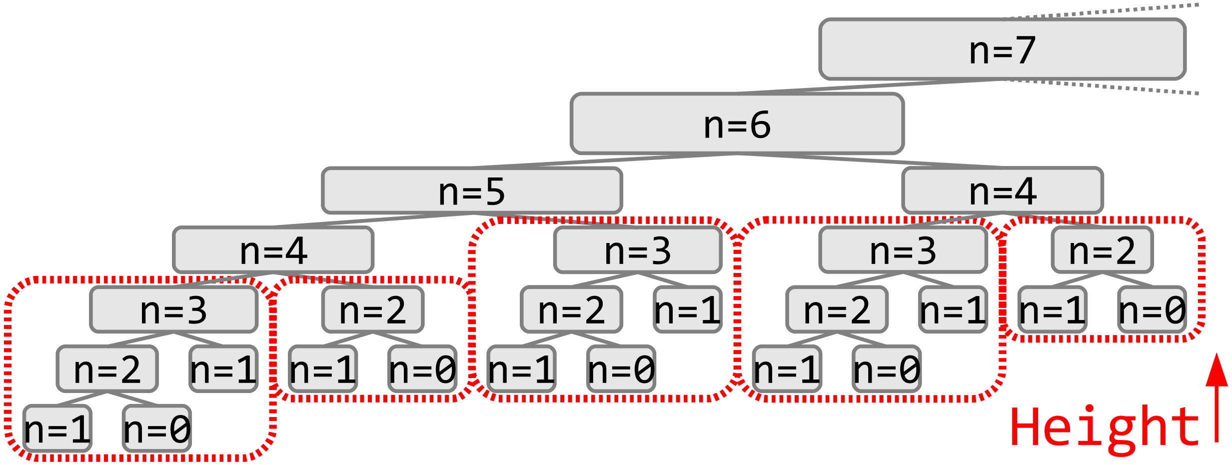
Summary of Static Cut-off

First, try to calculate a height-based cut-off condition.

- If it is calculable, determine H and apply one of them:
  - Static task elimination : reduce tasking overheads.
  - Code-bloat-free inlining : + reduce function call overheads.
  - Loopification : + convert recursion into a loop.

Lower is powerful, but less likely to be applicable.

- Otherwise, apply the dynamic cut-off [*]



[*] P. Thoman et al. Adaptive granularity control in task parallel programs using multiversioning. Euro-Par '13, 2013

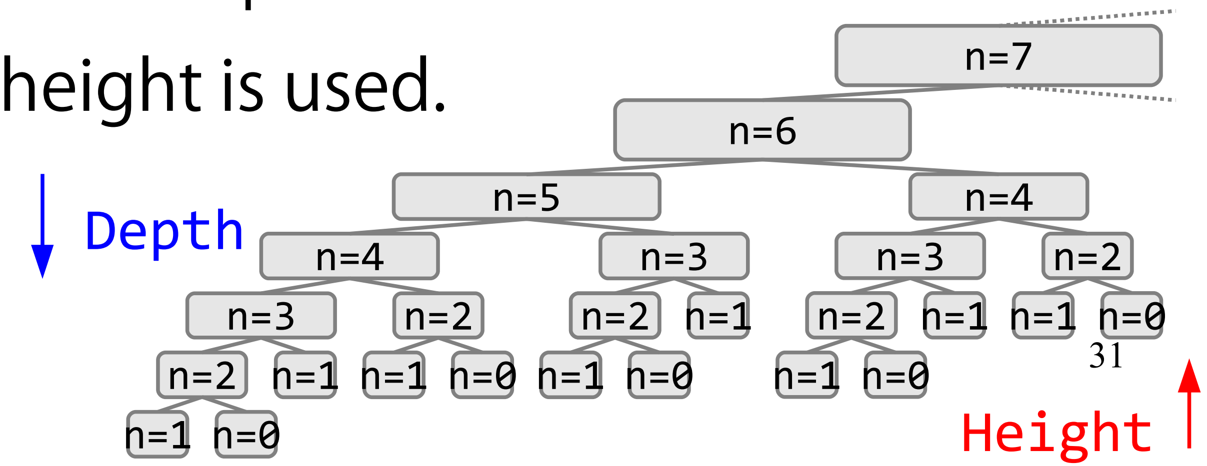
Limitations of Static Cut-off

- The evaluation had shown our static cut-off enhanced performance, yet there are room for further tuning to achieve best performance.
  1. Heuristics-based decision on cut-off threshold does not always return the optimal ones.
  2. Optimization for serialized tasks can be improved more.
     - e.g., combining multiple transformations
  3. Dynamic cut-off is not so efficient.
     - However, our static cut-off cannot be applied to all.

Index

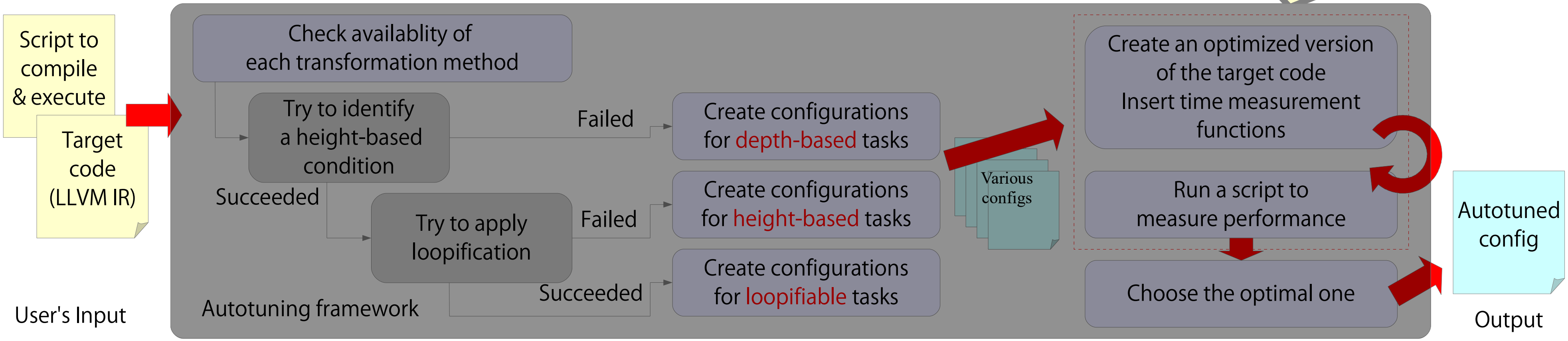
Cut-off with Autotuning

- Decide a cut-off strategy using an autotuning way.
- There are three possible elements for tuning:
  1. Cut-off thresholds (≒ a cut-off condition)
     - Especially for loopification, the cut-off condition has an impact on cache-blocking effect.
  2. Combination of transformations.
     - e.g., inlining & parallel + loopification & serial
  3. Whether depth or height is used.



Autotuning Flow

Black box for now.



- Input: original code + script to compile & run
- Output: autotuned configuration file
  - Our compiler generates an autotuned program with that file.
- We adopt an autotuning strategy similar to that of PetaBricks[*].

[*] J. Ansel et al., PetaBricks: A Language and Compiler for Algorithmic Choice. PLDI '09, 2009.

Related Work

PetaBricks

- PetaBricks[*], proposed by Ansel et al. is an autotuning framework for parallel divide-and-conquer algorithms.
  - It focuses on algorithmic choice.
  - e.g., for sorting, we can combine mergesort, quicksort, insertionsort together, by switching at each “conquer” phase.

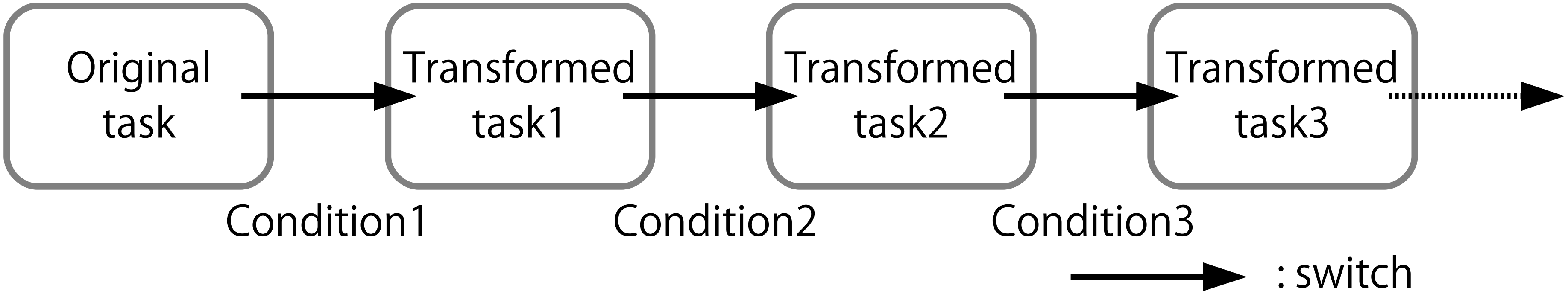
```
transform Sort
from In[n]
to Out[n]
{
  rule MergeSort
  to (Out out) from (In in)
  {
    [...]; // do MergeSort
  }
  rule QuickSort
  to (Out out) from (In in)
  {
    [...]; // do QuickSort
  }
  rule InsertionSort
  to (Out out) from (In in)
  {
    [...]; // do InsertionSort
  }
}
```

- Users need to write multiple versions of the algorithm.

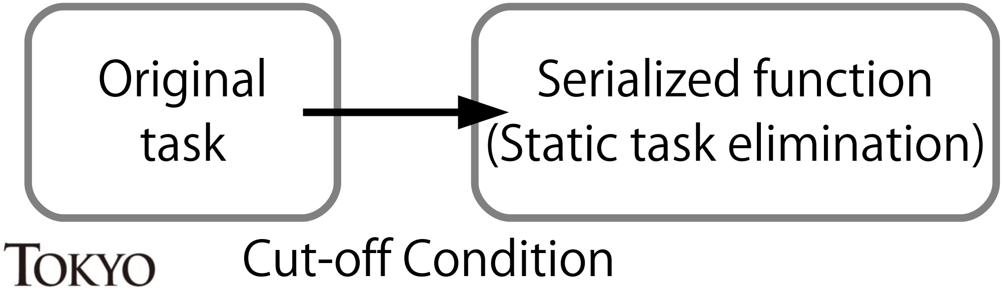
[*] J. Ansel et al., PetaBricks: A Language and Compiler for Algorithmic Choice. PLDI '09, 2009.

Basic Idea: Connecting Tasks

- Similar to the approach of PetaBricks, we optimize cut-off by connecting various tasks with appropriate conditions.



  - The simplest cut-off is represented as follows:



Example: Fibonacci

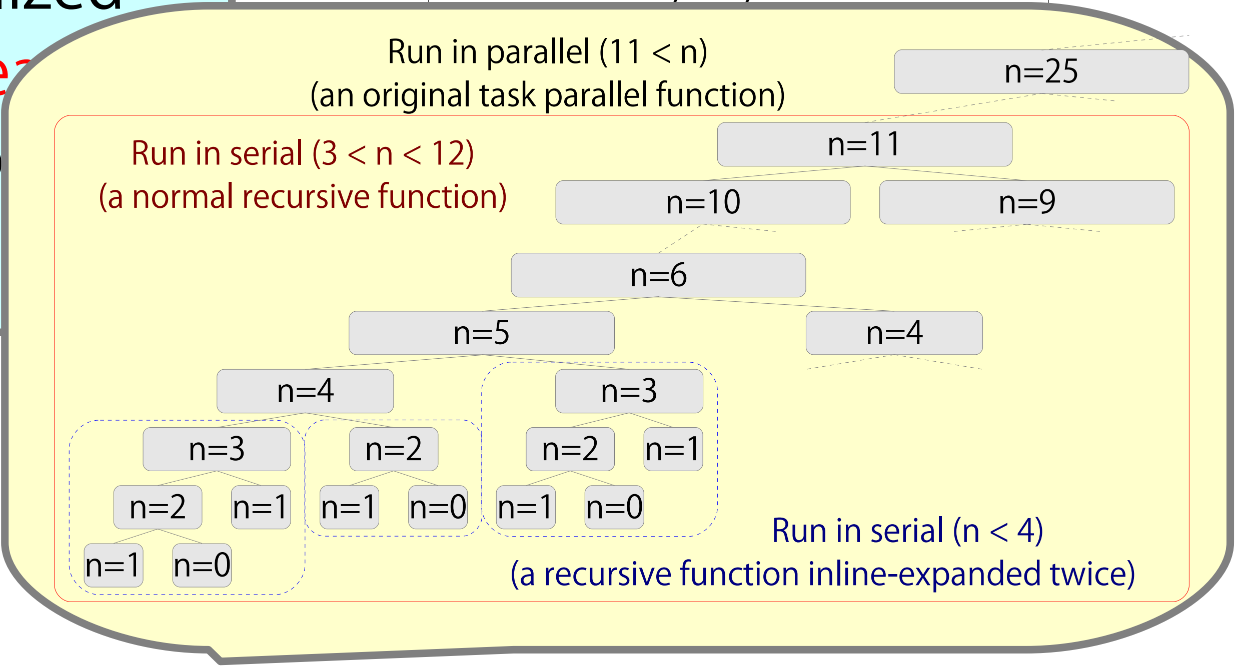
First, run the original task
to ensure parallelism,
then switch to the serialized
to reduce a overhead.
The leaf is inline-expanded
for serial performance.

```
void fib(int n, int* r){
  if(n < 2){
    *r = n;
  }else{
    int a, b;
    spawn fib(n-1, &a);
    spawn fib(n-2, &b);
    sync;
    *r = a + b;
  }
}
```

Example: Fibonacci

First, run the original task
to ensure parallelism,
then switch to the serialized
to reduce a overhe
Leaf tasks are inline-exp
for serial performa

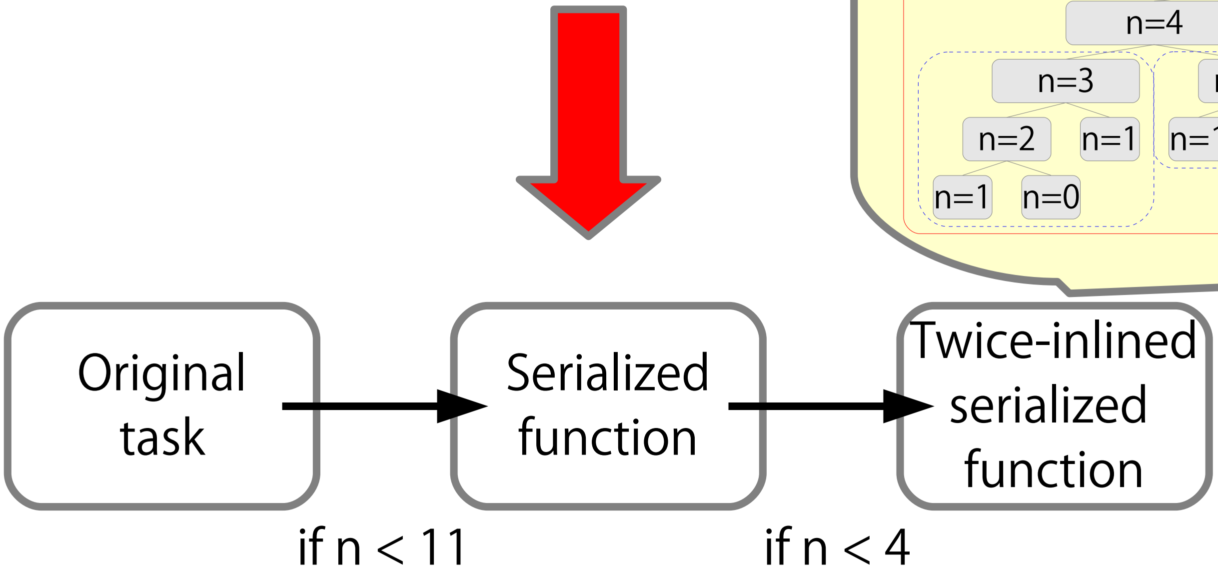
```
void fib(int n, int* r){
  if(n < 2){
    *r = n;
  }else{
    int a, b;
```



Run in parallel (11 < n)
(an original task parallel function)

Run in serial (3 < n < 12)
(a normal recursive function)

Run in serial (n < 4)
(a recursive function inline-expanded twice)

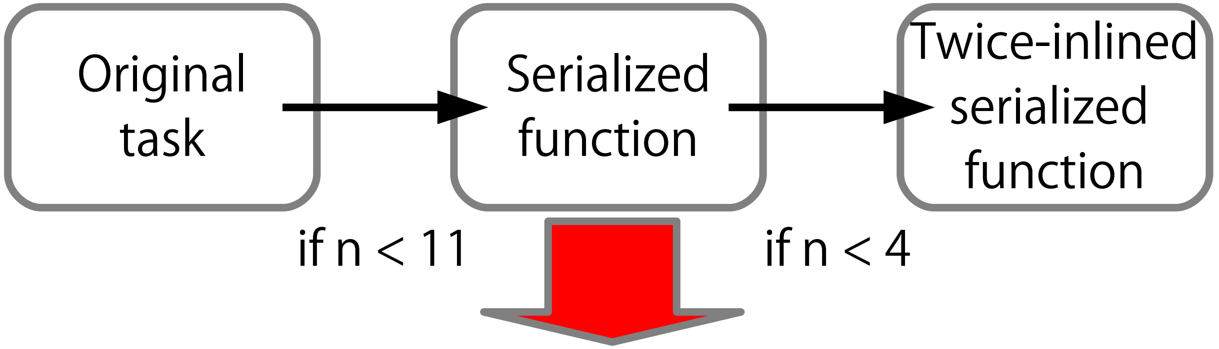


Original task → Serialized function → Twice-inlined serialized function

if n < 11

if n < 4

Example: Final Code



Original task

Serialized function

Twice-inlined serialized function

if n < 11

if n < 4

```
void fib(int n, int* r){
  if(n < 11){
    fib2(n, r);
  }else{
    int a, b;
    spawn fib(n-1, &a);
    spawn fib(n-2, &b);
    sync;
    *r = a + b;
  }
}
```

```
void fib2(int n, int* r){
  if(n < 4){
    fib3(n, r);
  }else{
    int a, b;
    fib2(n-1, &a);
    fib2(n-2, &b);
    *r = a + b;
  }
}
```

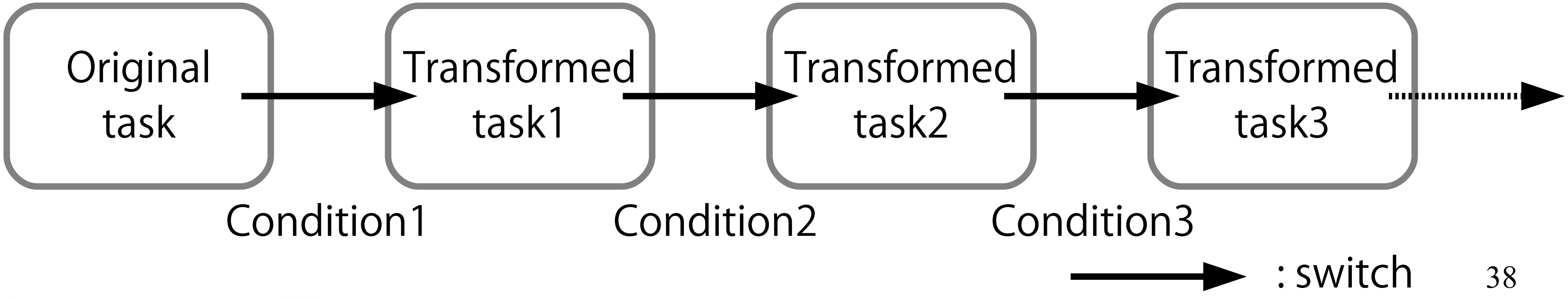
```
void fib3(int n, int* r){
  if(n < 2){
    *r = n;
  }else
    [inlined twice];
}
```

Search Space for Autotuning

- There are two tuning parameters:
  1. Switching conditions
  2. Optimizations for each task (task)
     - + Optimization parameters (e.g., # of times of inlining)

Sometimes not parallelized
e.g., serialized task

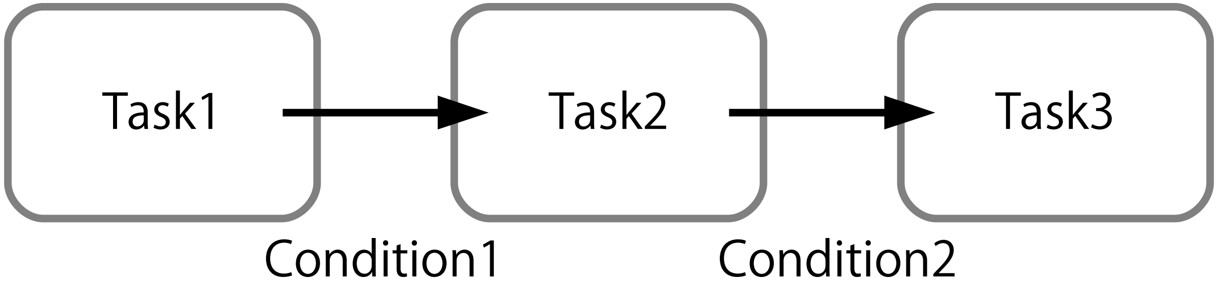
- The number of patterns are potentially countless.



Basic Cut-off Strategy

To limit the search space.

1. Use height rather than depth if possible.
2. # of task versions is at most 3.



Task1 → Task2 → Task3

Condition1

Condition2

- An original task: no optimization is applied

  → fine-grained & parallel
- A middle task: optimization may be applied

  → fine~coarse-grained & serial
- A leaf task: optimization may be applied

  → coarse-grained & serial

Three Typical Patterns

- We defined three typical patterns to limit the search space.
  - Pattern 1: depth-based cut-off
    - Target examples: tree traversals
  - Pattern 2: height-based cut-off without loopification
    - Target examples: fibonacci, nqueens
  - Pattern 3: height-based cut-off with loopification
    - Target examples: vector addition, matrix multiplication

Three Typical Patterns

- We defined three typical patterns to limit the search space.
  - Pattern 1: depth-based cut-off
    - Target examples: tree traversals
  - Pattern 2: height-based cut-off without loopification
    - Target examples: fibonacci, nqueens
  - Pattern 3: height-based cut-off with loopification
    - Target examples: vector addition, matrix multiplication

Three Typical Patterns

- We defined three typical patterns to limit the search space.
  - Pattern 1: depth-based cut-off
    - Target examples: tree traversals
  - Pattern 2: height-based cut-off without loopification
    - Target examples: fibonacci, nqueens
  - Pattern 3: height-based cut-off with loopification
    - Target examples: vector addition, matrix multiplication

Three Typical Patterns

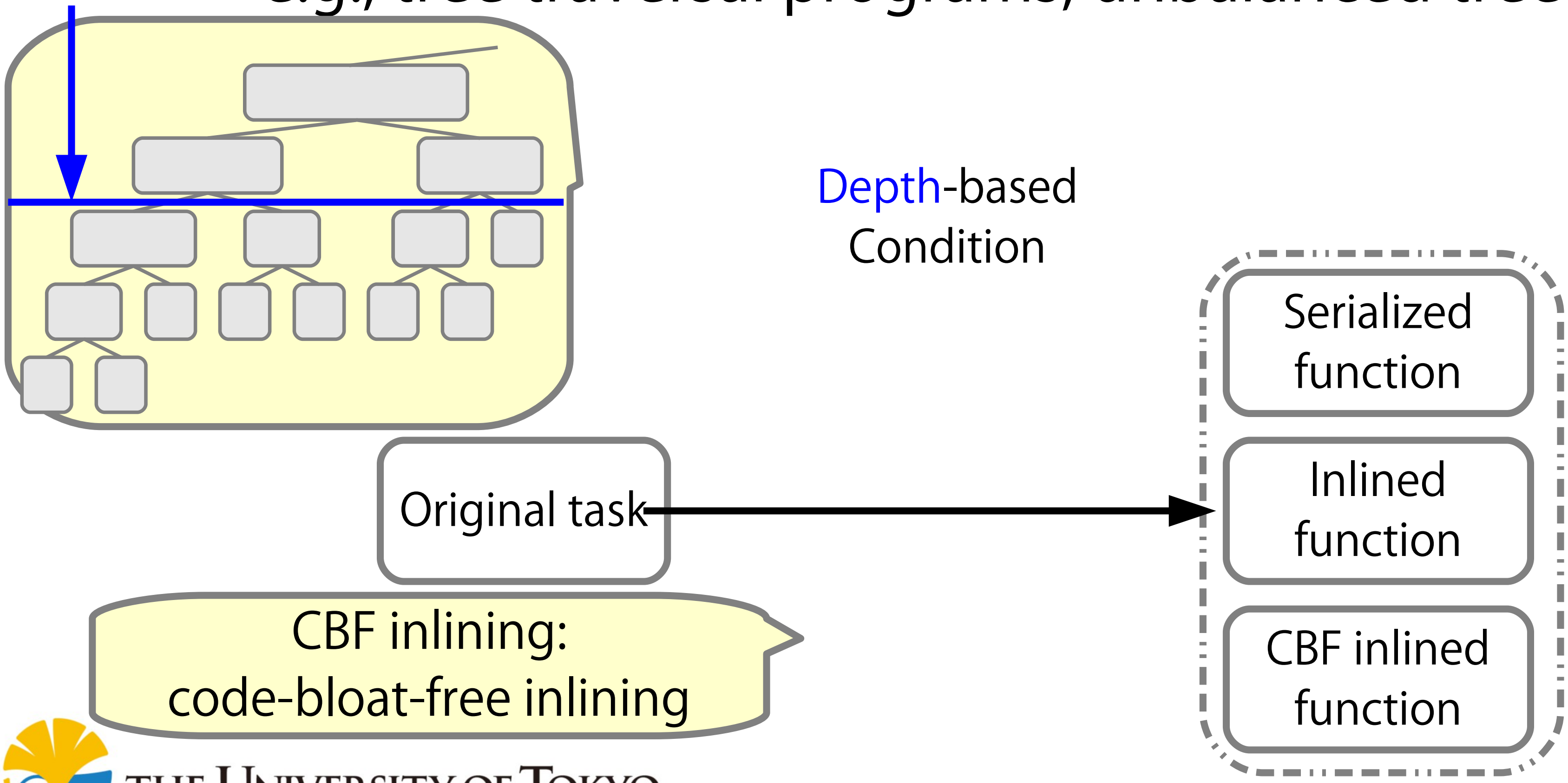
- We defined three typical patterns to limit the search space.
  - Pattern 1: depth-based cut-off
    - Target examples: tree traversals
  - Pattern 2: height-based cut-off without loopification
    - Target examples: fibonacci, nqueens
  - Pattern 3: height-based cut-off with loopification
    - Target examples: vector addition, matrix multiplication

Three Typical Patterns

- We defined three typical patterns to limit the search space.
  - Pattern 1: depth-based cut-off
    - Target examples: tree traversals
  - Pattern 2: height-based cut-off without loopification
    - Target examples: fibonacci, nqueens
  - Pattern 3: height-based cut-off with loopification
    - Target examples: vector addition, matrix multiplication

Pattern 1: Depth-based Cut-off

- It is designed for tasks to which it is difficult to apply static cut-off.
  - e.g., tree traversal programs, unbalanced tree search



Depth-based Condition

Original task

CBF inlining: code-bloat-free inlining

Serialized function

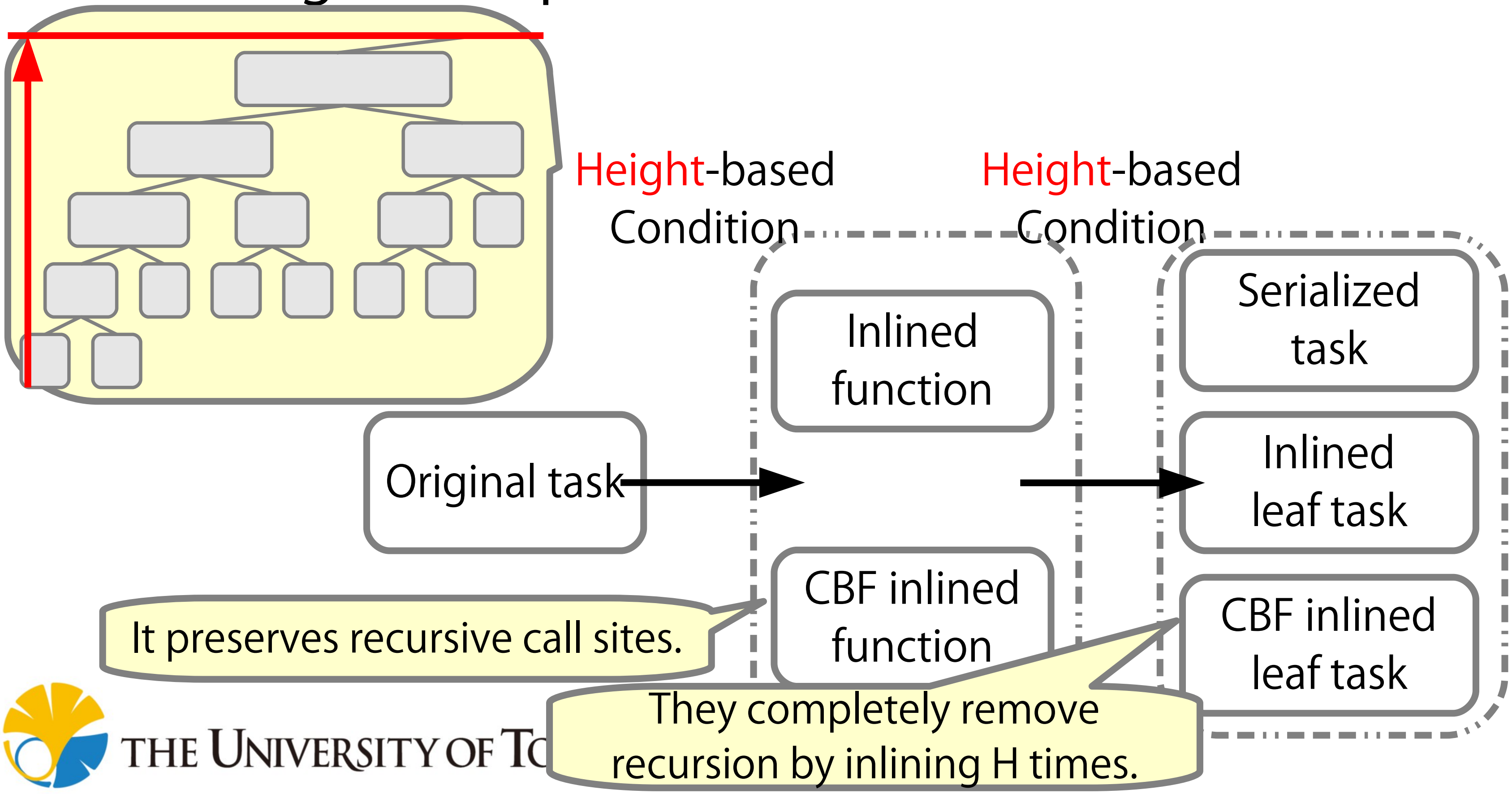
Inlined function

CBF inlined function

Pattern 2: Height-based Cut-off

without loopification

- It is designed for tasks to which static cut-off is applicable, but loopification is not.
  - e.g., fib, nqueens



Height-based Condition

Height-based Condition

Original task

Inlined function

CBF inlined function

Serialized task

Inlined leaf task

CBF inlined leaf task

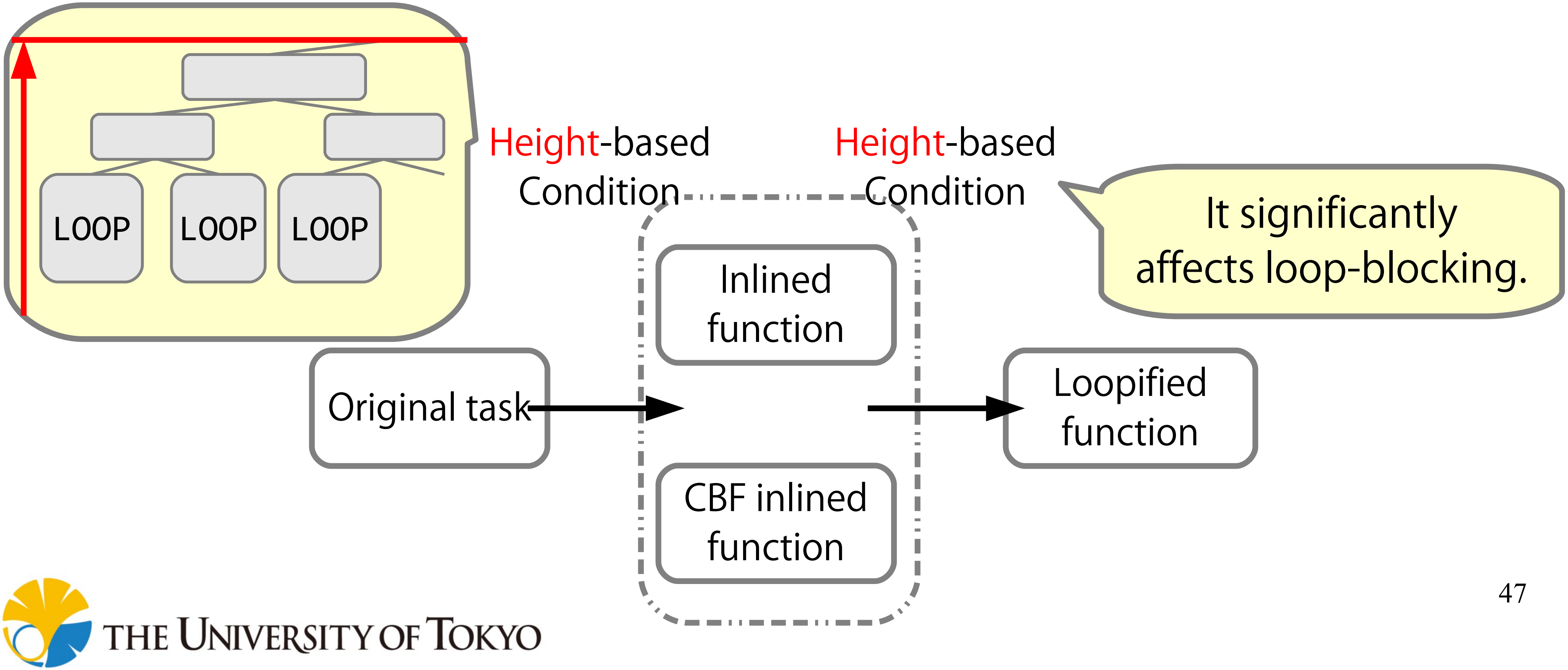
It preserves recursive call sites.

They completely remove recursion by inlining H times.

Pattern 3: Height-based Cut-off

with loopification

- It is designed for loopifiable tasks.
  - e.g., vecadd. matmul, heat2d



It significantly affects loop-blocking.

Avoid Loss of Parallelism

- More parallelism is better if the performance is the same in terms of dynamic load balancing.
- Our autotuning adapt the switching condition preserving most parallelism, which can accomplish 99% of the optimal performance measured.
  - In this example, we choose n<2000 even if n<10000 performs slightly better.

```
void vecadd(float* a, float* b, int n){
  if(n == 1){
    *a += *b;
  }else{
    spawn vecadd(a, b, n/2);
    spawn vecadd(a+n/2, b+n/2, n-n/2);
    sync;
  }
}
```

Cut-off n<20000: 11[s]

Cut-off n<10000: 10[s]

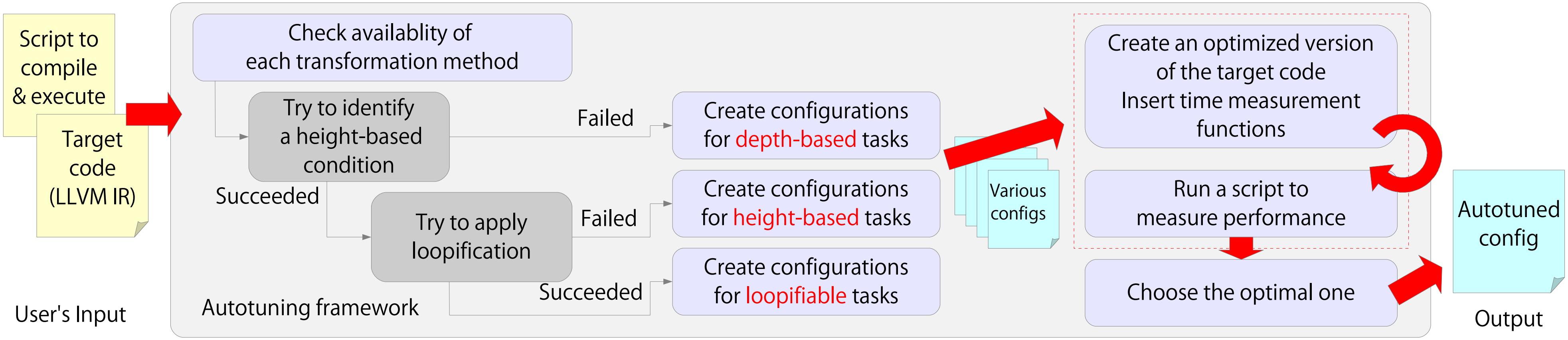
Cut-off n<2000: 10.1[s]

Not fastest, though.

Cut-off n<1000: 12.0[s]

Cut-off n<100: 14.0[s]

Autotuning: Summary



- Our autotuning searches for the best combination of differently transformed tasks.
  - It contains a cut-off concept.
- It employs three patterns to limit the search space.
  - Depth-based one
  - Height-based ones (w/ & w/o loopification)

Index

Implementation & Environment

- We implemented it as an optimization pass on LLVM 3.6.0.

  Modified MassiveThreads[*1], a lightweight work-stealing based task parallel system adopting the child-first scheduling policy[*2].

- An autotuning driver is written in Python.
- Experiments were done on dual sockets of Intel Xeon E5-2699 v3 (Haswell) processors (36 cores in total).
  - Use `numactl --interleave=all` to balance physical memory across sockets.

THE UNIVERSITY OF TOKYO

[*1] MassiveThreads https://github.com/massivethreads/massivethreads

[*2] Mohr et al., Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs, LFP '90, 1990

Benchmarks

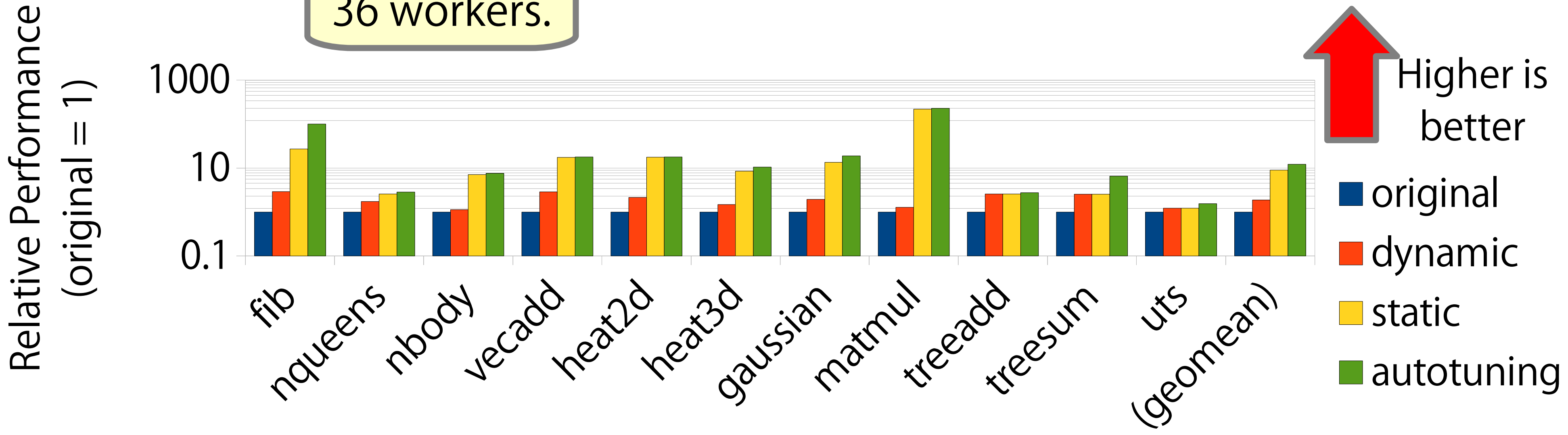
- 11 benchmarks were prepared for evaluation.
  - All are divide-until-trivial task parallel programs.

- fib
- nqueens
- nbody
- vecadd
- heat2d
- heat3d
- gaussian
- matmul
- treeadd
- treesum
- uts

	Dynamic Cut-off	Autotuning Pattern
fib	✔	2. Height-based without loopification
nqueens	✔	2. Height-based without loopification
nbody	✔	2. Height-based without loopification
vecadd	✔	3. Height-based with loopification
heat2d	✔	3. Height-based with loopification
heat3d	✔	3. Height-based with loopification
gaussian	✔	3. Height-based with loopification
matmul	✔	3. Height-based with loopification
treeadd	✔	1. Depth-based
treesum	✔	1. Depth-based
uts	✔	1. Depth-based

Static cut-off is not applicable to them.

Multi-threaded Performance



36 workers.

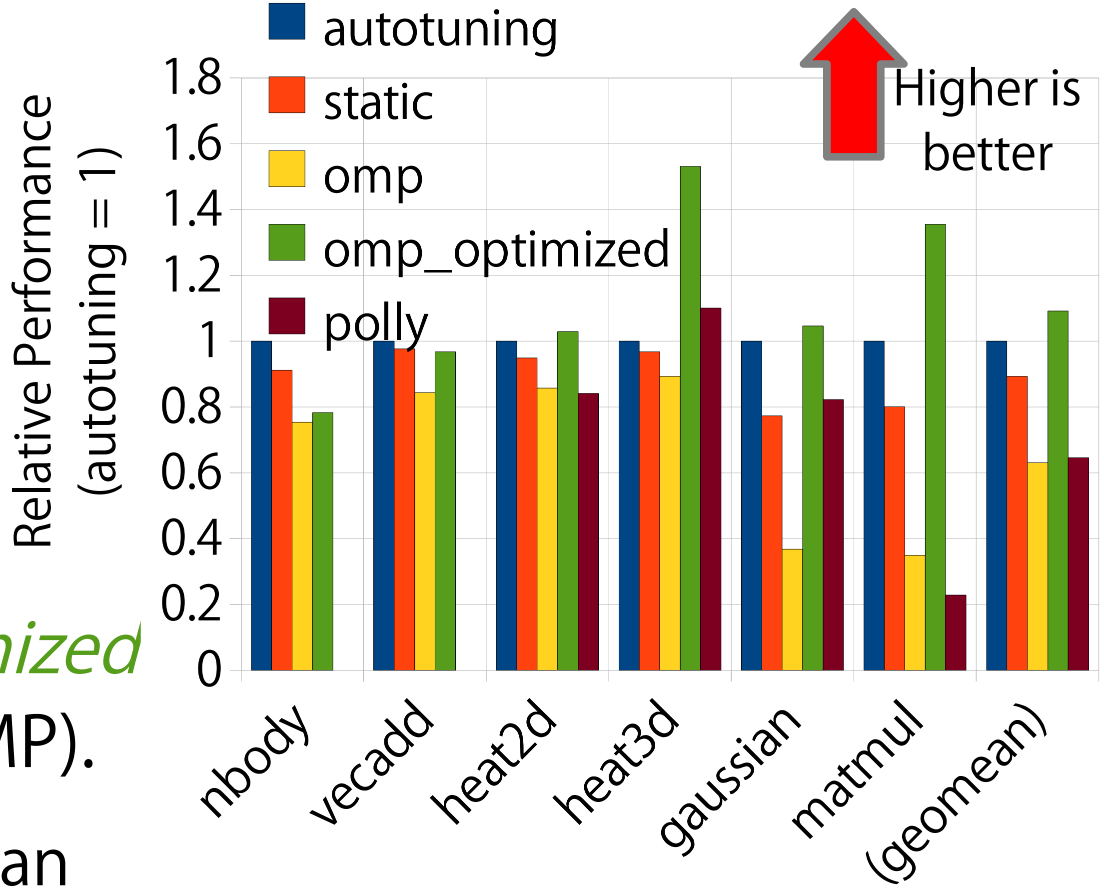
- Optimization including *dynamic* (dynamic cut-off[*]) improved performance over *original* (no cut-off)
- *autotuning* (proposal) was faster than *dynamic* and *static* (static cut-off) overall.

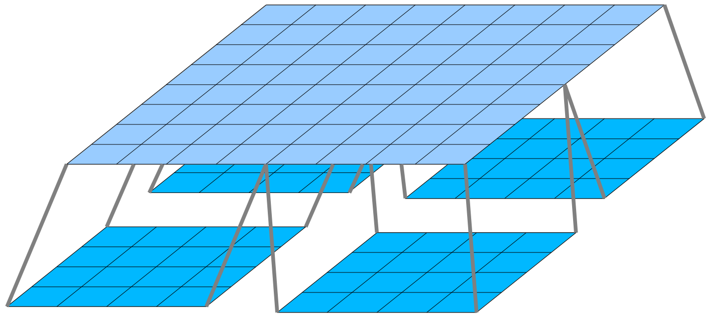
---

[*] P. Thoman et al. Adaptive granularity control in task parallel programs using multiversioning. Euro-Par '13, 2013

vs. Loop Parallel Programs

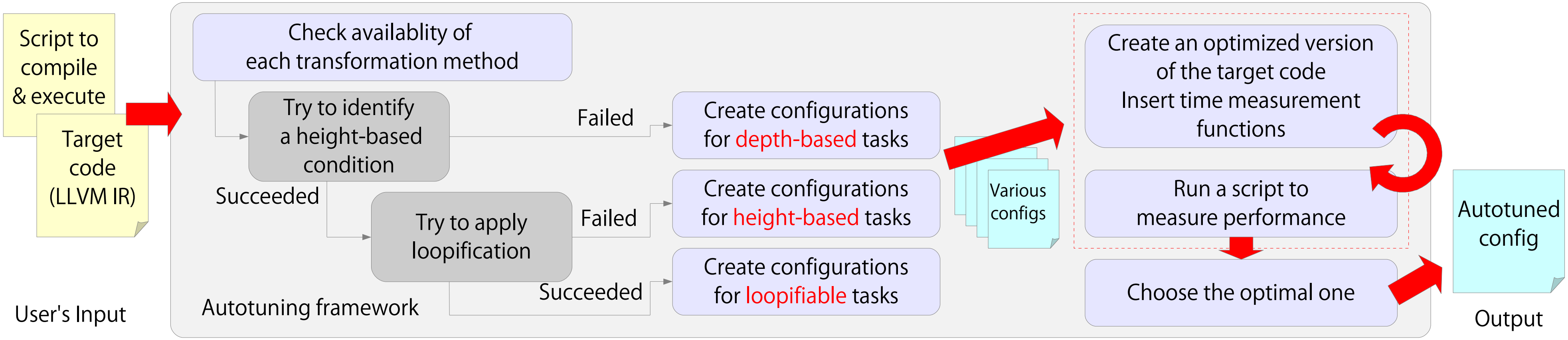
- *autotuning* (proposed autotuned one) was
  - comparable to *polly* (Polly) and *omp* (OpenMP)
  - defeated by *omp_optimized* (hand-optimized OpenMP).
    - Hand-tuned OpenMP can employ flexible cache-blocking.
    - div-and-conq divides the axis only by a constant integer.





Index

Conclusion



- We developed an autotuning framework for divide-until-trivial task parallel programs.
- It achieved significant speedup over the original naïve task parallel programs.