Design and Implementation of a Customizable Work Stealing Scheduler

Jun Nakashima*1,*2, Sho Nakatani*1, and Kenjiro Taura *1
*1 The University of Tokyo, *2 JSPS Research Fellowship for Young Scientists

Agenda

- Introduction
- Work Stealing Customization Framework
- Evaluation
- Related Work
- Conclusion and Future Work

Background

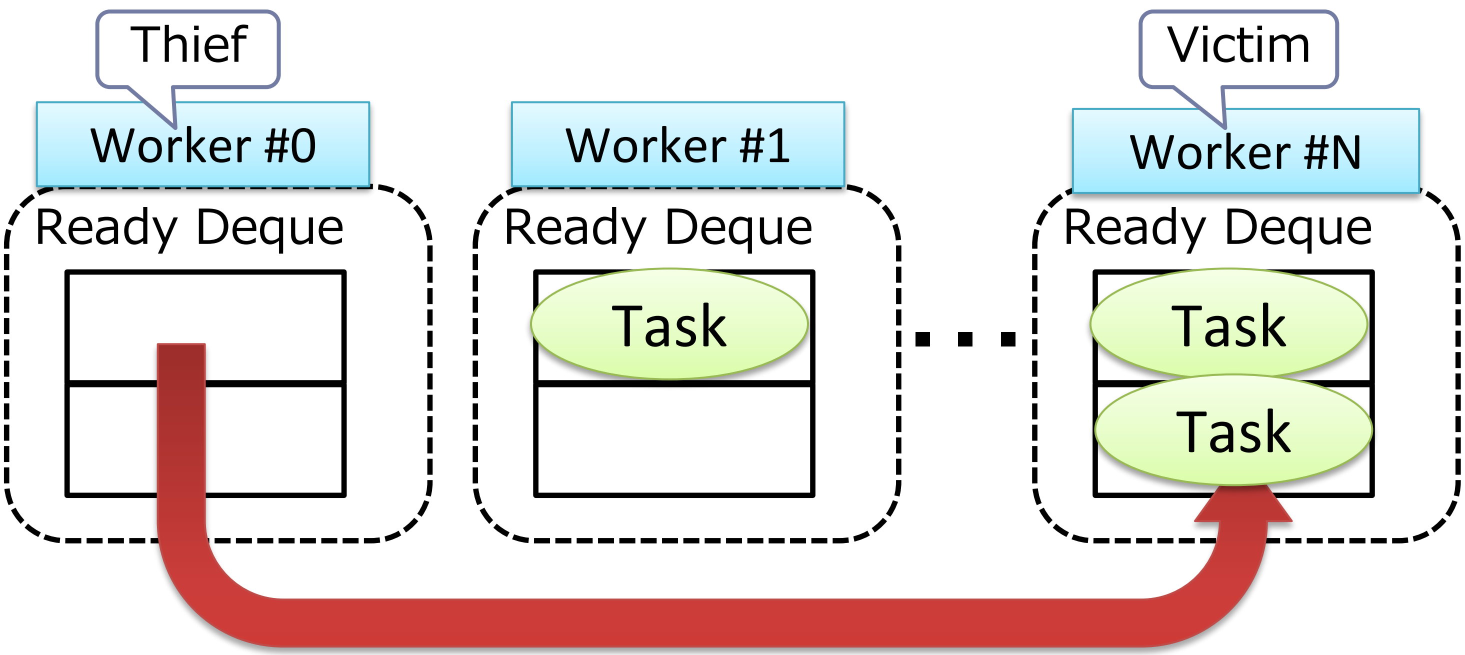
- Productivity is one of the major challenges of parallel programming frameworks
  - Many frameworks and languages proposed

- Many of them provide task parallelism
  - Chapel[Cray], X10[IBM], …
  - Support many forms of parallelism on top of it

Need efficient runtime systems

Work Stealing Scheduler

- A well-known strategy for task parallelism
  - Idle workers steal a task from another (victim)
  - Typically a victim is chosen randomly



Work Stealing Scheduler

- Randomness may cause significant slowdown
- e.g.: A machine with deeper memory hierarchy
  - Considering data placement is essential

- Motivation:
- <u>Work stealing scheduler must become clever</u>
  - Consider hardware and application knowledge

Our Approach

- Ideal solution: A general strategy that can be used without any effort
  - It remains challenging
  - Difficult to obtain application knowledge
- Our approach: A framework to customize work stealing strategy
  - Enable programmers to optimize the strategy
  - Less ambitious, but more practical

Agenda

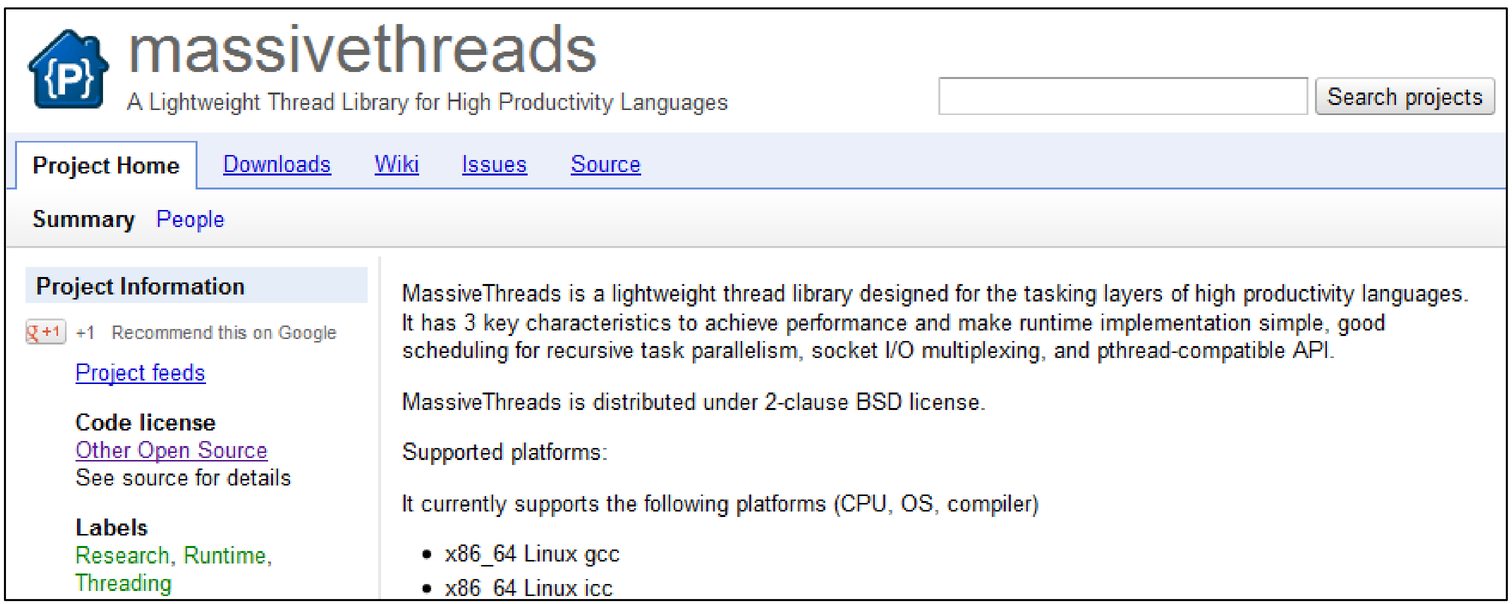
- Introduction
- Work Stealing Customization Framework
- Evaluation
- Related Work
- Conclusion and Future Work

Design Principle

- Purpose of customization
  - Steal tasks being aware of hardware/application
    - e.g. Shared-cache among workers
  - Avoid task steals with negative side-effect
    - e.g. Extra cache misses
- Focus on providing functions to customize a strategy to select a victim of work stealing

Implementation

- Implemented by modifying MassiveThreads
  - A lightweight thread library by our group
  - written in C
    - http://code.google.com/p/massivethreads/



massivethreads

A Lightweight Thread Library for High Productivity Languages

Search projects

Project Home | Downloads | Wiki | Issues | Source

Summary People

Project Information

+1 Recommend this on Google

Project feeds

Code license

Other Open Source

See source for details

Labels

Research, Runtime, Threading

MassiveThreads is a lightweight thread library designed for the tasking layers of high productivity languages. It has 3 key characteristics to achieve performance and make runtime implementation simple, good scheduling for recursive task parallelism, socket I/O multiplexing, and pthread-compatible API.

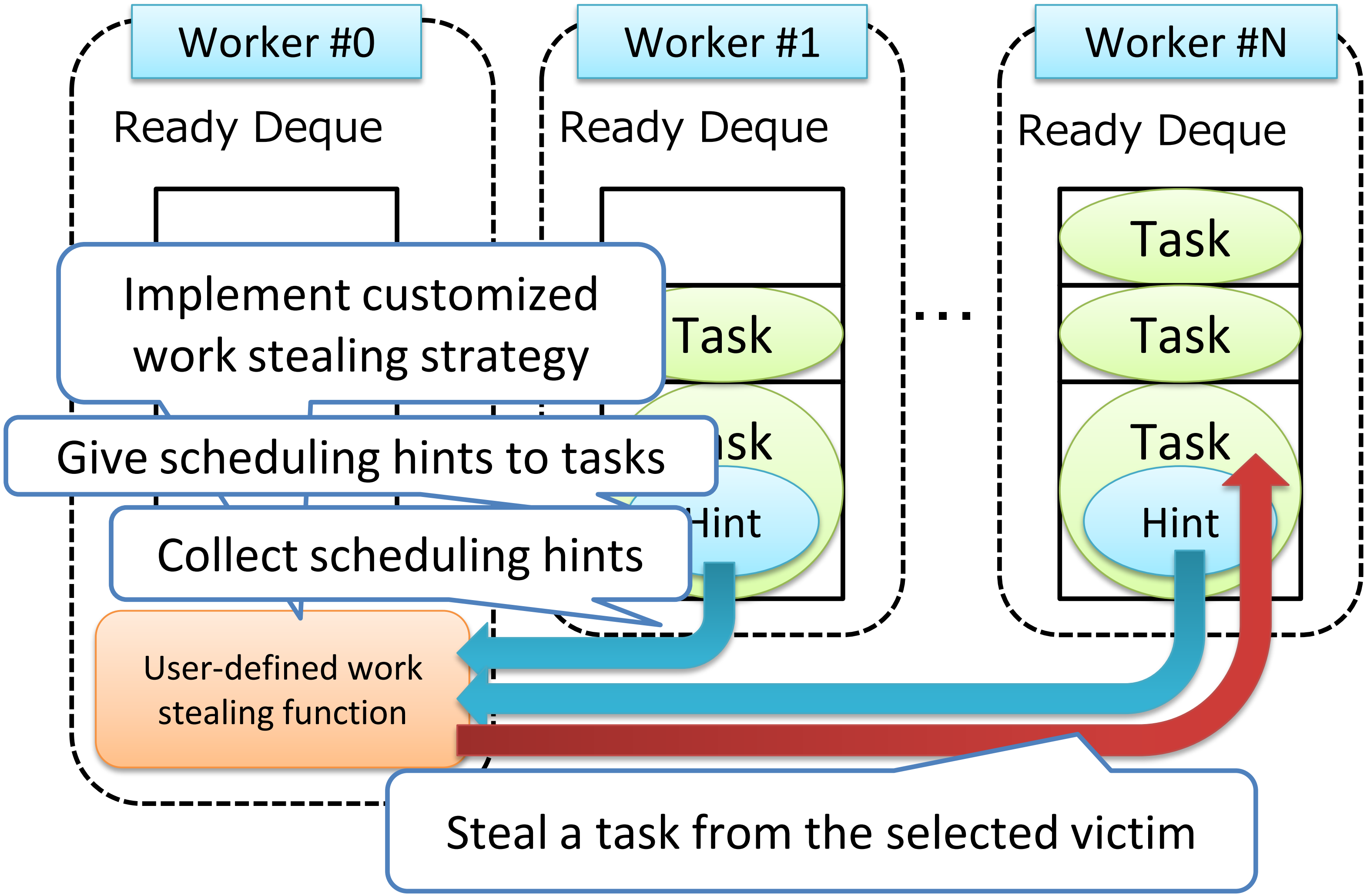
MassiveThreads is distributed under 2-clause BSD license.

Supported platforms:

It currently supports the following platforms (CPU, OS, compiler)

- x86_64 Linux gcc
- x86_64 Linux icc

Overview



Worker #0
Ready Deque
Worker #1
Ready Deque
Worker #N
Ready Deque
Task
Task
Task
Hint
Task
Task
Hint
Implement customized work stealing strategy
Give scheduling hints to tasks
Collect scheduling hints
User-defined work stealing function
Steal a task from the selected victim

How to Customize

- Two things to do:
  - Modify application to give scheduling hints to tasks
  - Implement user-defined work stealing function

Example Strategy: Depth-Aware

- Try to steal coarse-grained tasks more carefully
  - For divide-and-conquer applications
- Scheduling hint: recursion depth
  - As an indicator of task granularity
- Steal tasks which have the smallest recursion depth

Give Scheduling Hints

- Scheduling hint:
  - A piece of data associated with a task
- Create a task with initial value

```
void user_task (…){
  …
  create_task(user_task,…);
```

Application maintains recursion depth

```
void user_task (int depth,…){
  …
  int newdepth=depth+1;
  create_task_with_hint(user_task,&newdepth,sizeof(int),…);
  …
}
```

Create a task with a scheduling hint

User-defined Work Stealing Function

- Invoked when a worker is idle
- Most operation is allowed
  - Except some functions of runtime system

```
/* User-defined work stealing function definition */
void depth_aware_steal(int id)
{
  task_handle t_stolen;
  /* Here it tries to steal a task */
  return t_stolen;
}

/* At the beginning of an application */
set_steal_function(depth_aware_steal);
```

ID of idle worker

Should return the stolen task

Switch work stealing function

User-defined Work Stealing Function

- Typical implementation:
  1. Select multiple workers as candidates of a victim
  2. Read scheduling hints from available tasks
  3. Select one worker as a victim
  4. Try to steal from the victim
  5. Confirm the stolen task

Step 1. Select Candidates

- Use a function *get_random_workers*
  - return random non-duplicated worker IDs

```
…
int num_of_cadidates = 2;
int candidates[num_of_cadidates];
get_random_workers(candidates,num_of_candidates);
…
```

- Can be written by hand for better selection
  - e.g.: considering memory hierarchy

Step 2. Collect Scheduling Hints

- Use *readydeque_peek* function:
  - Get a copy of scheduling hint of a task to be stolen
- Collect hints from all the candidates

```
…
int depth[num_of_cadidates];
for (i=0;i<num_of_cadidates;i++){
  size_t size=sizeof(int);
  readydeque_peek(candidates[i],&depth[i],&size);
  /* Set depth to -1 if failed to peek */
  if (size!=sizeof(int))depth[i]=-1;
}
…
```

Step 3. Select One Worker as a Victim

- Select a victim based on user-defined strategy

- In depth-aware:
  - Worker that has a task with the smallest depth

```
…
int target=0;
for (i=1;i<num_of_cadidates;i++){
  if (depth[target]<depth[i])target=depth;
}
…
```

Step 4. Try to Steal a Task

- ***readydeque_trysteal*** function: Try to steal from selected victim
- Can specify ***confirm function*** (used in next step)

```
…
task_handle ret;
ret = readydeque_trysteal(target,
                depth_aware_confirm, depth[target]);
…
```

Step 5. Confirm the Stolen Task

- Confirm function:
  - Called when a steal has succeeded
- Cancel the steal if the stolen task is undesirable

```
int depth_aware_confirm(task_handle t,void *param)
{
  int expect_depth=(int)param;
  int *stolen_task_depth=get_hint_ptr(t);
  return (*stolen_task_depth)<=expect_depth;
}


  …
  task_handle ret;
  ret = readydeque_trysteal(target,
                depth_aware_confirm, depth[target]);
  …
```

Agenda

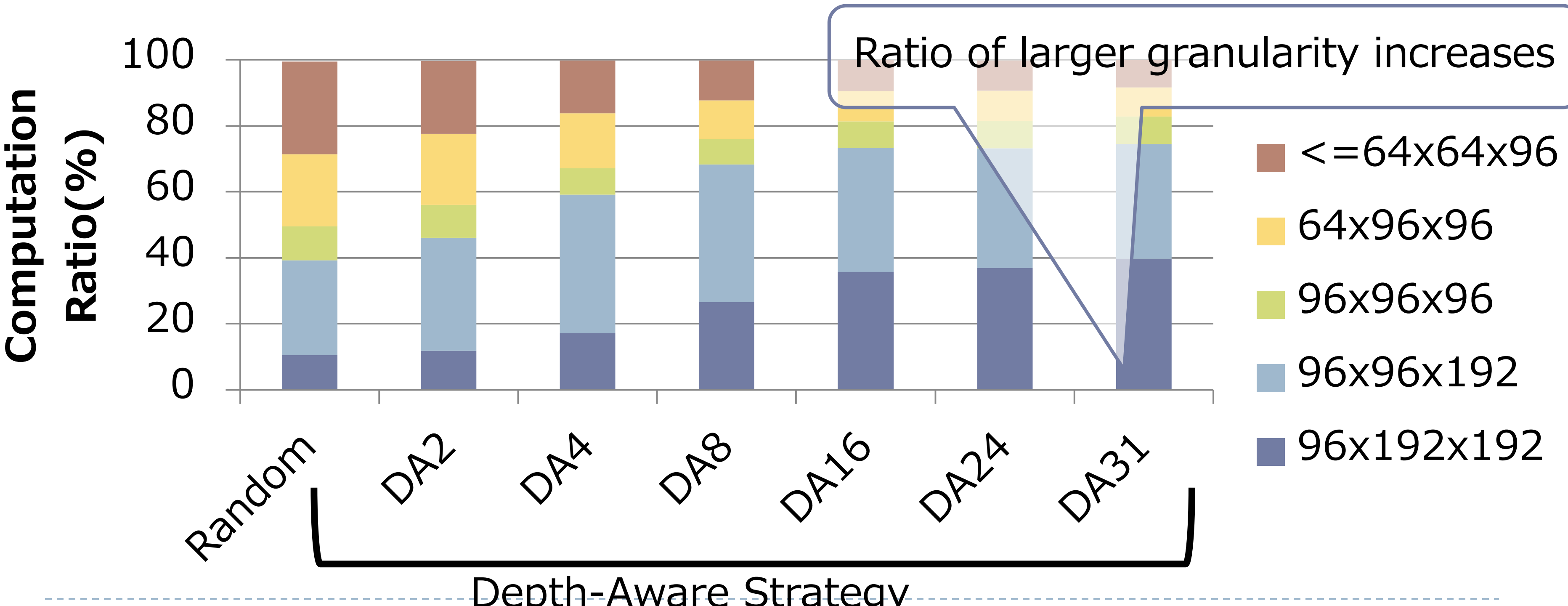
- Introduction
- Work Stealing Customization Framework
- Evaluation
- Related Work
- Conclusion and Future Work

Evaluation

- Implemented two scheduling strategies
  - Depth-aware
  - Affinity-aware
- Evaluated on a machine with 32 cores
  - Quad-Core Opteron 8354 (2.2 GHz) × 8 Sockets
  - Caches
    - L1D: 64 KB/Core, L2: 512 KB/Core, L3: 2 MB/Socket
  - NUMA Policy :Interleave

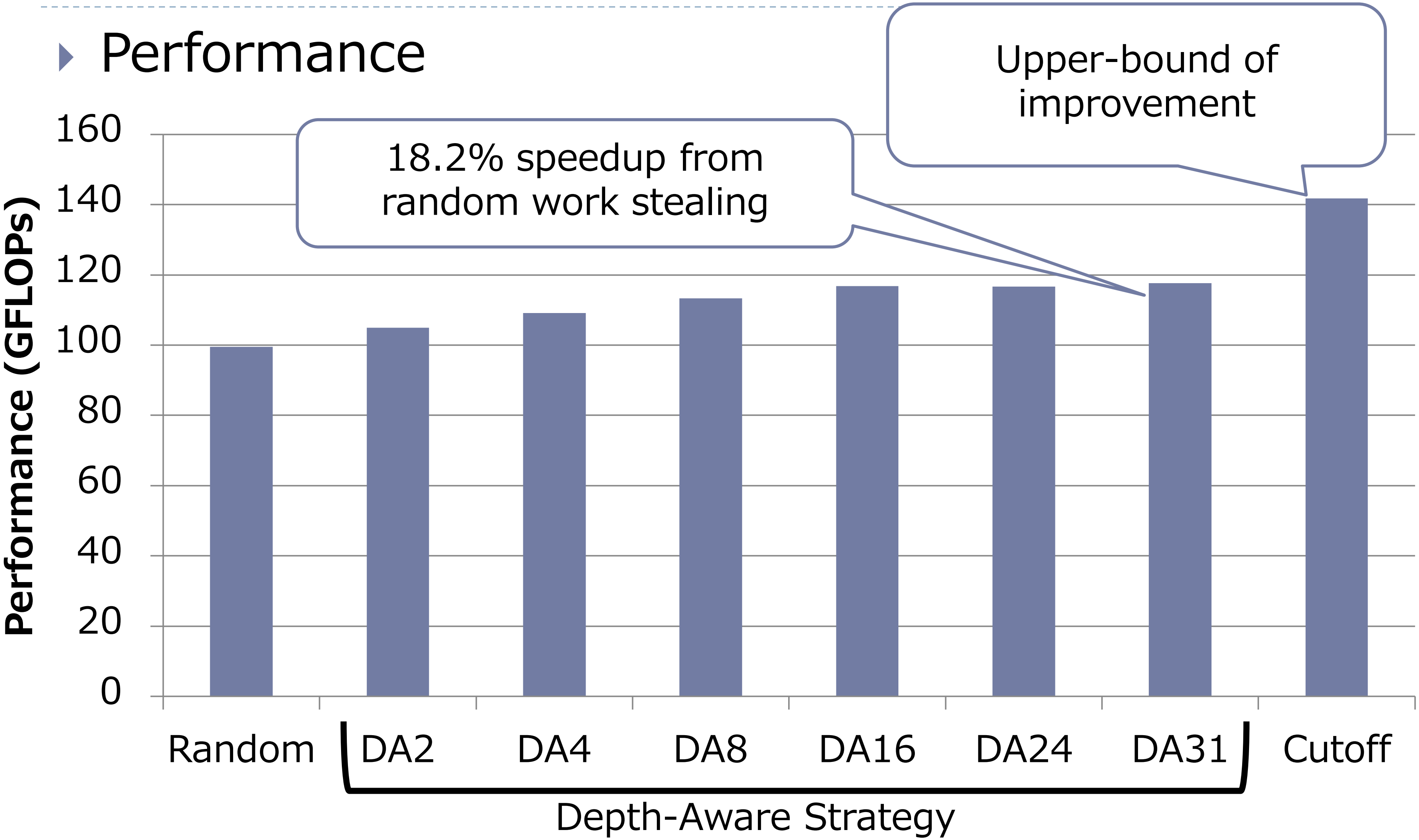
Depth-Aware Evaluation Result

- App: Matrix Multiply using divide-and-conquer
  - Performance gets better if granularity gets larger
  - Size: 768x768 SP
- Granularity of Computation



Depth-Aware Evaluation Result

- Performance



Affinity-Aware Strategy

- Give a task an affinity as array of integers
  - How the task desires to be stolen from each worker
- Try to execute a task with the largest affinity
- Variants:
  - Best-effort: Steal even if the affinity is zero
  - Strict: Ignore tasks with no affinity

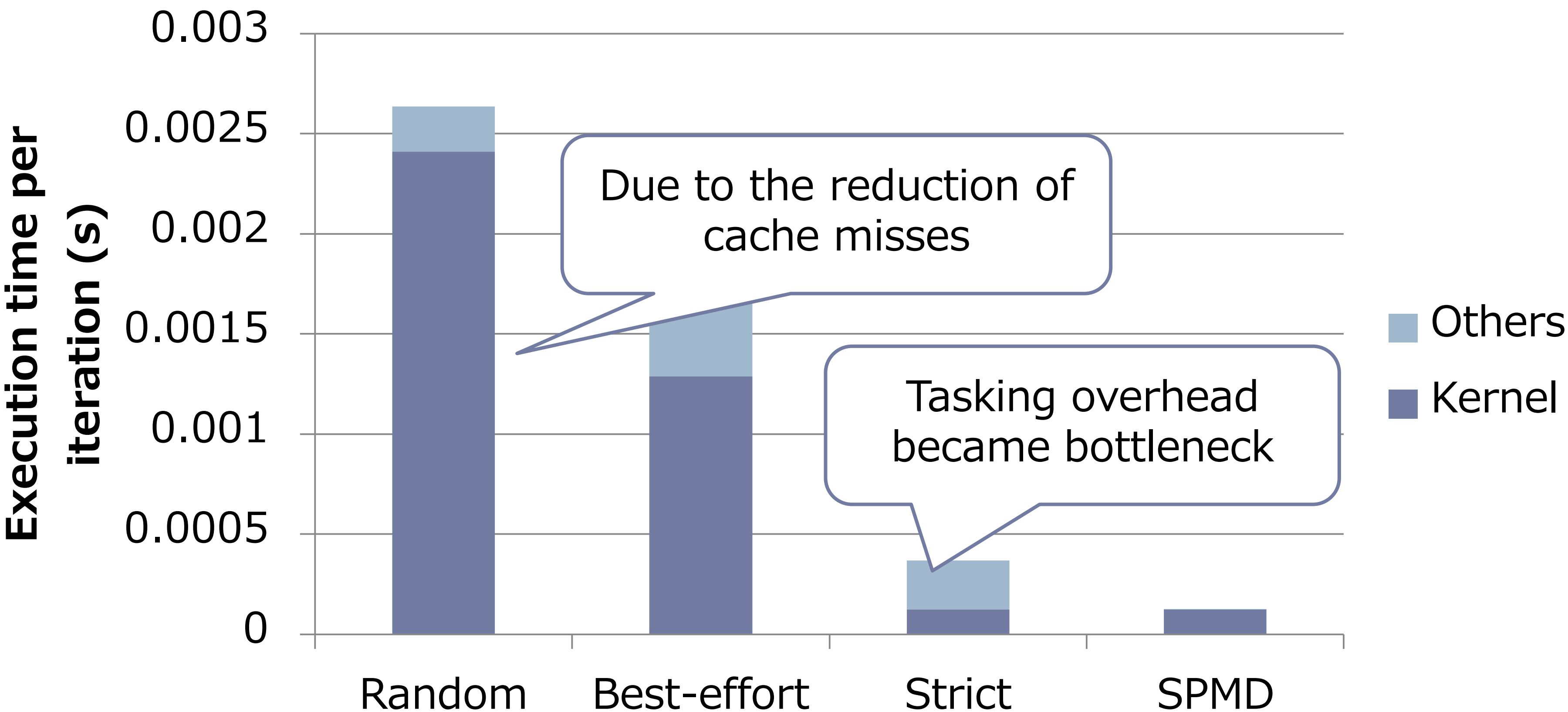
Affinity-Aware Strategy

- Benchmark: Repeats STREAM TRIAD
  - Parallelized using divide-and-conquer (256 tasks)
  - Array size: 8MB * 3 = 24MB
  - 768KB/core (fits L2 and L3 cache)

- Need to utilize previously cached data
- Give a task an affinity with a worker of last iteration
  - # of candidates=31

Affinity-Aware Evaluation Result

- Execution time per iteration



Agenda

- Introduction
- Work Stealing Customization Framework
- Evaluation
- Related Work
- Conclusion and Future Work

Related Work

- CATS[Chen,2012]
  - Online profiling and DAG partitioning
- Qthreads[Oliver,2012]
  - Share one task queue among intra-socket cores
- Work-stealing with Configurable Scheduling Strategies[Wimmer,2013]
  - # of tasks to steal, execution order,…

What's new in Our Work?

- Our proposed framework is <u>flexible</u>
- Enable programmers to customize a victim selection strategy directly
- Tradeoff:
  - ○ Performance can be much improved
  - × Additional effort for customization

Conclusion

- Proposed a framework to customize work stealing strategy
  - Focus on how to decide a victim of work stealing
- Example customization strategies worked as expected

Future Work

- Improve framework design
  - Look for good tradeoff between performance and programmers' effort
- Further evaluation:
  - Unbalanced application
    - Adaptive Mesh Refinement
  - On distributed memory environment

Thank you for listening!

Takeout

- We propose a framework to customize work stealing strategy
- Give scheduling hints to tasks
- User-defined work stealing function
  1. Select candidates of a victim
  2. Read scheduling hints
  3. Select one worker
  4. Try to steal
  5. Confirm
- MassiveThreads:
  - http://code.google.com/p/massivethreads/
- Contact me:
  - nakashima@eidos.ic.i.u-tokyo.ac.jp