Whole Program Data Dependence Profiling to Unveil Parallel Regions in the Dynamic Execution

Yukinori Sato (JAIST, JST CREST)
Yasushi Inoguchi (JAIST)
Tadao Nakamura (Keio University)

2012 IEEE International Symposium on Workload Characterization
November 5, 2012

Introduction

Current obstacles for improving performance

Parallelizing sequential programs

- A critical process for developing high performance codes
- Trends towards accelerators makes matter worse

App codes become complex

- Lines of code of SPEC2006CPU program are mostly over 10 thousands
- Hard to read all of them
- Issues for data sharing and communication among CPU cores

To parallelize and accelerate programs according to scaling of hardware resources, we must develop a fundamental method that ease the above issues

Tools that can guide and support them by identifying parallelism and dataflow inherent in programs are fairly important

Analyzing data dependencies

"**Data dependence analysis**" is an essential tool for parallelization

- Traditionally static analysis by compilers are dominated
  - Data dependence tests
    - Such as Banerjee test and Omega test
    - Effective only for subscripted array expressions in array references
    - Inadequate for the wide use of pointer expressions
  - Alias profiling
    - Able to detect data dependencies induced by pointers
    - Not able to extract dynamic behaviors decided at run time
    - Also, it cannot disambiguate dependencies finer than memory blocks allocated from the *malloc*

Static data dependence analysis techniques are limited to particular situations!

To extract all of data dependencies in execution, we develop a new framework that can extract ***dynamic data dependencies across whole program execution***.

In this paper

We extract ***dynamic memory dataflow across whole program execution*** in order to unveil parallel regions

- Monitoring data dependencies by dynamic binary translation
  - DBT uses pre-compiled executable binary code as input
  - DBT enables ***transparent analysis*** without compelling programmers to read application program source code.
- Our implementation to minimize profiling overheads
  - Data dependencies are analyzed based on regions of dynamic program context, and represented by ***Loop-Call Context Tree with Memory dataflow (LCCT+M)***
  - We present an efficient method that makes use of ***paging*** of memory access tables
- We visualize the LCCT+M of actual application programs and present how we identify ***loop-, task- and pipeline parallelisms*** using the LCCT+M

To profile dynamic data dependencies

The existing dynamic data dependence profiling is inadequate!

SD^3 [Kim, MICRO2010]

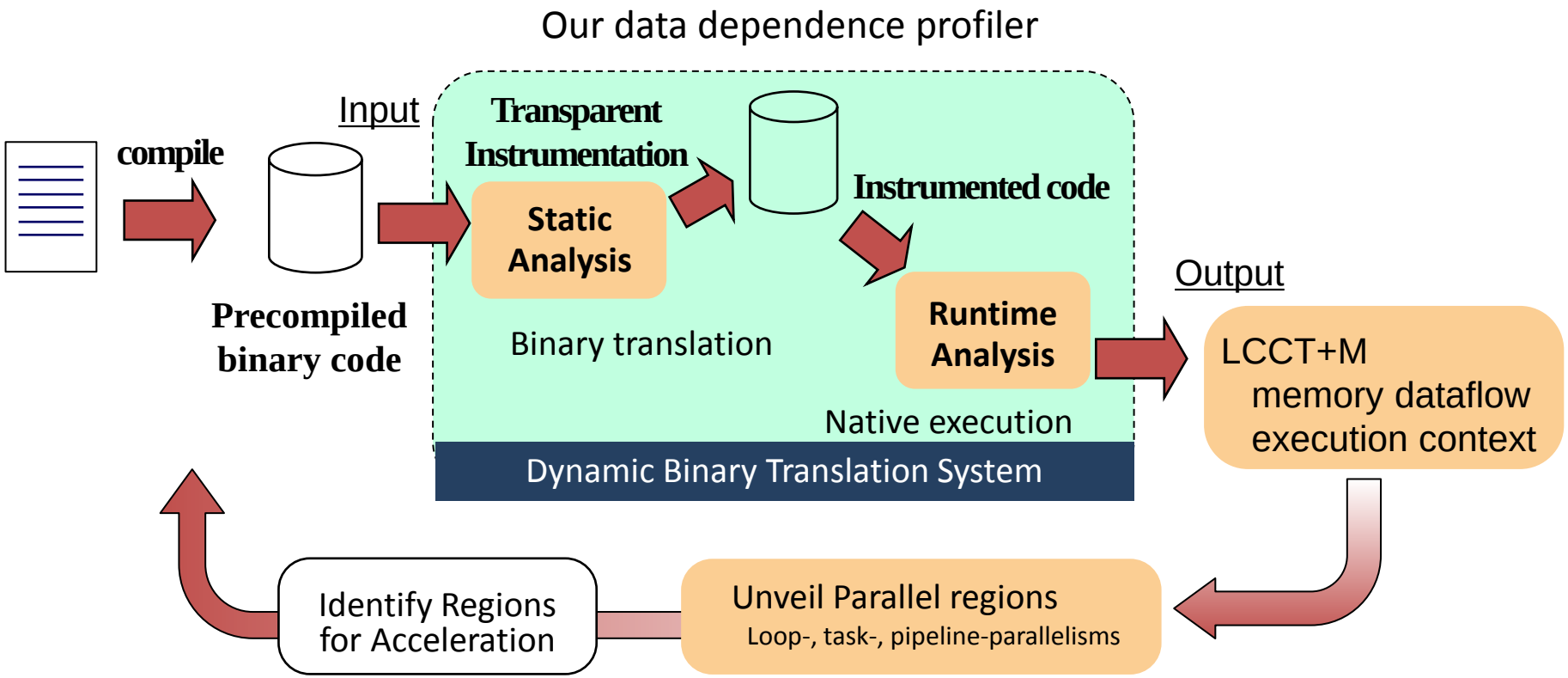
- Focus on loop nest structures, perform stride detection and compression of data dependencies for array access
- So limited to dependencies among loop iterations
- Whole program data dependence analysis including scalar memory accesses was not discussed

Pairwise method [Chen, CC2004]

- This compares all addresses of every pair of memory references
- But it needs very expensive pairwise address comparison throughout the program execution

We need to realize ***an accurate and light-weight profiling technique*** that records accurate data dependencies with reasonable time & memory overhead

An overview of our data dependence profiling

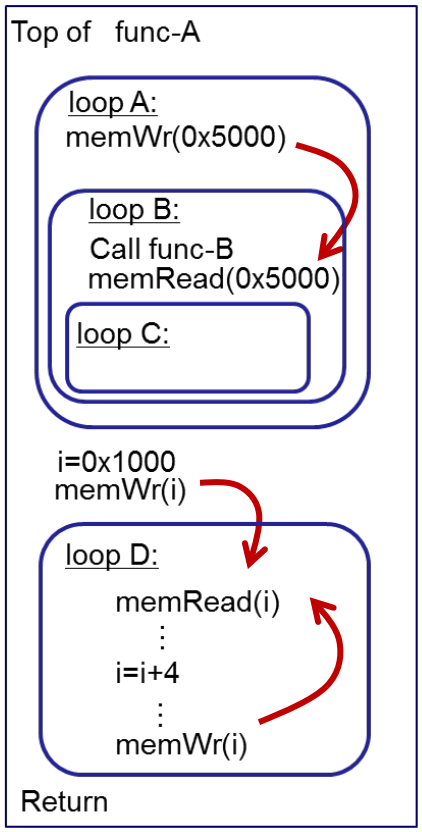


- Using executable binary code as input, we start analysis on DBT system
- At static analysis phase, we insert analysis codes for dynamic analysis
- At runtime analysis phase, we keep track of dynamic data dependencies
  - Together with the dynamic context of loop and call activations
  - Using memory monitoring table with paging

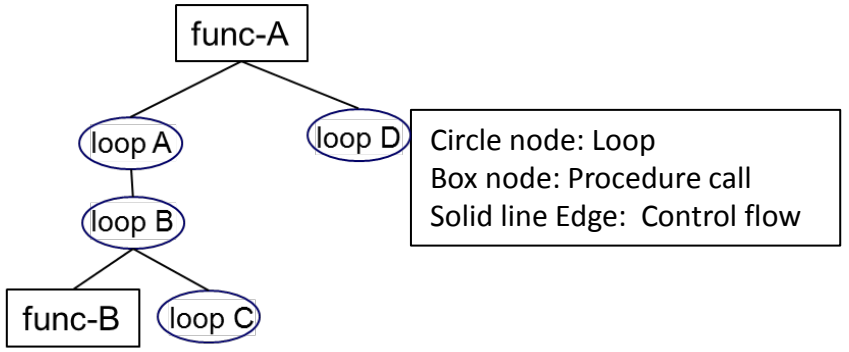
Keys to realize efficient profiling

Representing dataflow using LCCT+M

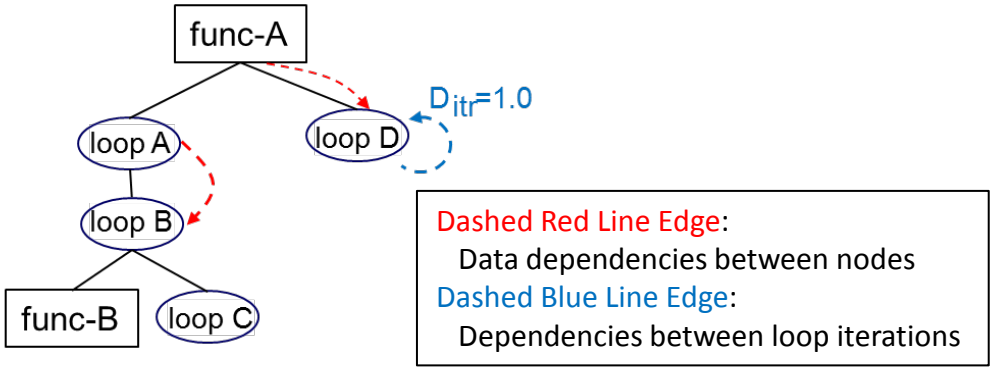
- We propose *Loop-Call Context Tree with Memory dataflow (LCCT+M)* representation
  - This can effectively depict dynamic data dependencies across loop iterations and procedure calls
  - For uncovering various types of parallelisms in loop, task and pipeline fashion



(a) Loops, calls and memOps in a procedure



(b) Generated L-CCT



(c) Generated LCCT+M

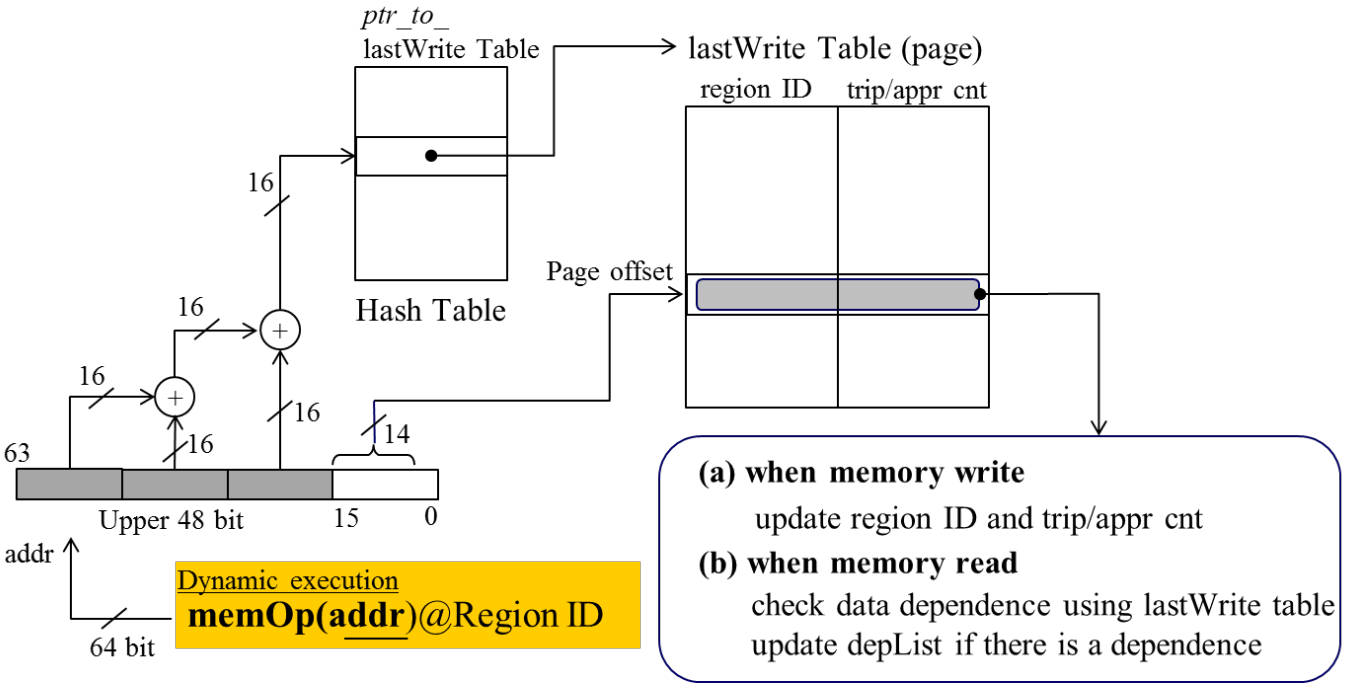
How to keep track of data dependencies

- We focus on only the true dependence in the paper.
  - This is because the essential dependence for parallelization is true dependence
- To maintain who writes the most recent value in each accessed address
  - We build a table structure referred to as a lastWrite table

Here, we propose the ***paging*** of lastWrite tables

- To realize the faster access with smaller memory size
- This concept is similar to pages in virtual memory system, where physical memory space is relocated as a set of fixed sized block (also called a ***page***)

The structure of lastWrite table with paging



- Hash table is used to access lastWrite tables and each of a page includes 8kB address regions
- An element of a lastWrite table is accessed using page offset
- Because a lastWrite table is allocated on-demand only when the memory segment is newly accessed, we believe this can contribute to saving the total amount of memory

Using lastWrite table, we keep track of region-based data dependencies

4. Experiment

We implement our dynamic data dependence profiling on Pin tool set, and evaluate it using NPB and SPEC CPU benchmark programs

System configuration

Appro 1143H servers composed of four AMD Opteron 8435 CPUs, 128GB memory
Red Hat Enterprise Linux Server 5.4.

Benchmark programs

NAS Parallel benchmark 3.3 serial version (NPB3.3) with A class data set
SPEC CPU2006 with ref data set
4 programs from INT, 9 programs from FP
We compiled using GNU Compiler Collection 4.1.2

Conditions

- Analysis includes all of memory access including shared libraries
- Dynamic execution context analysis is performed when the main function is called and terminated when it returns
- We only focus on dependencies via memory access and ignore these among registers

The statistics of the obtained LCCT+M

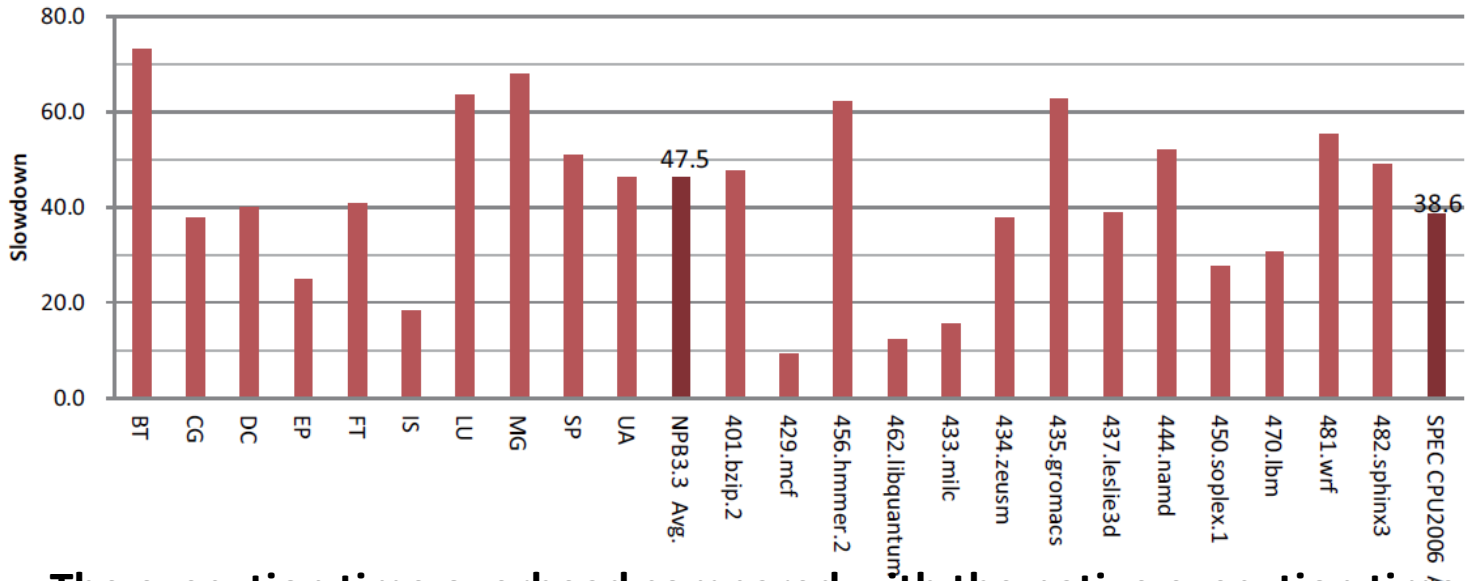
The results of data dependence analysis for SPEC 2006 CPU INT

	401.bzip.2	429.mcf	456.hmmer	462.libquantum
input data	chicken.jpg	-	retro.hmm	-
totalInst	1.80E+11	3.72E+11	2.00E+12	2.24E+12
MemRead	4.88E+10	1.24E+11	1.02E+12	2.22E+11
MemWrite	2.03E+10	3.83E+10	1.80E+11	7.81E+10
RtnNodes	293	356	1660	669
LoopNodes	271	69	130	146
DepEdges	2484	1819	6869	3019

- The amount of memory operation occupies from 30% to 50% of dynamic execution
  - The # of data dependencies between instructions will be proportional to the # of total memory operations
- From the results, we confirm that our data dependence profiling can efficiently handle data dependencies between nodes in whole program executions

Dynamic context of program execution contributes to summarizing the dynamic flow of instruction sequences

The execution time overhead

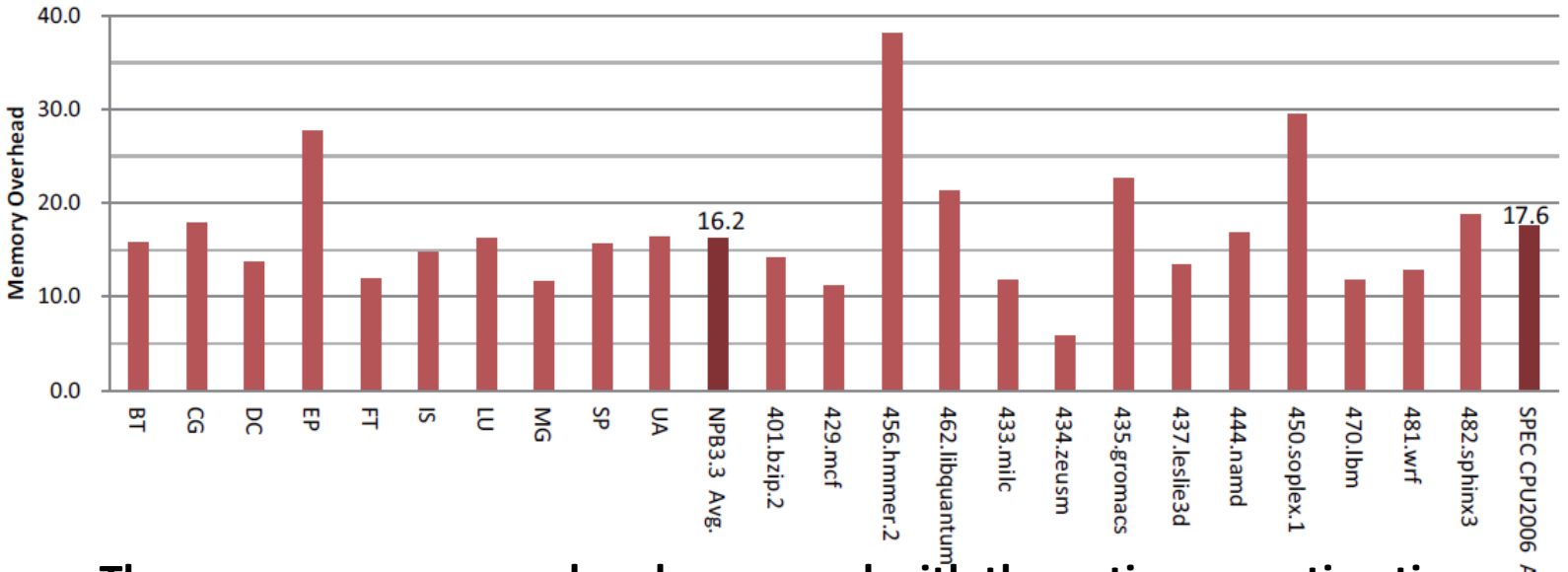


The execution time overhead compared with the native execution time

- The runtime overheads of our profiling are 47.5x and 38.6x in average for NPB3.3 and SPEC CPU2006
- These are dramatically smaller than SD^3 method [Kim, MICRO2010]
  - The runtime overhead of serial version of the SD^3 method is 289x for SPEC CPU2006
  - Since the SD^3 method must compress array memory accesses based on their strides, it will require larger overheads for profiling compared with ours

Fast paging access to lastWrite tables together with LCCT+M contributes to reducing performance overhead of our profiling

The memory space overhead



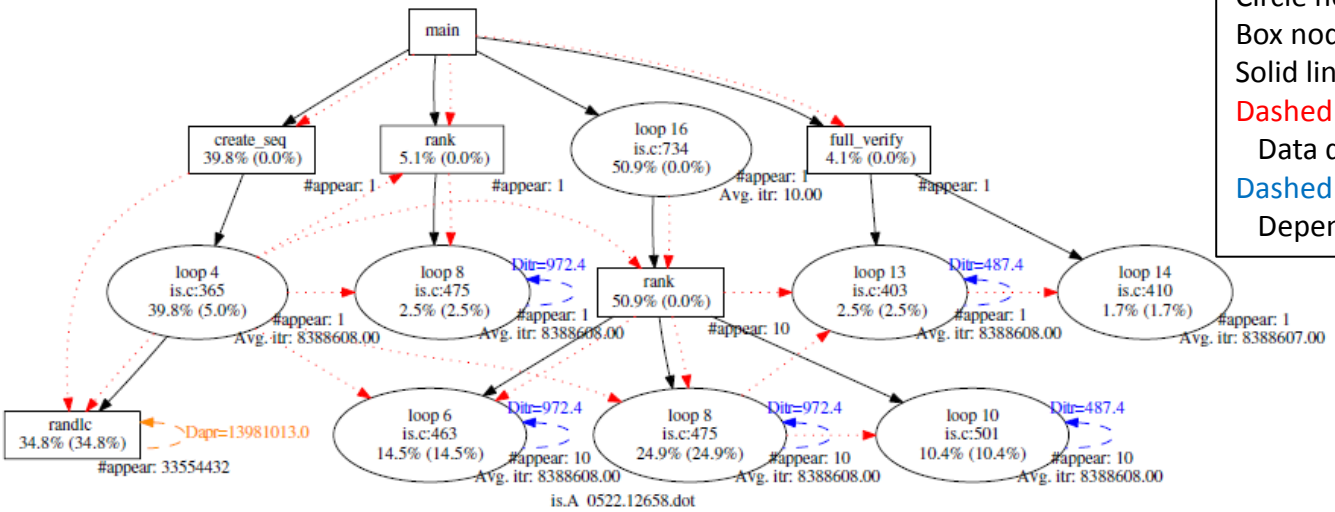
The memory space overhead compared with the native execution time

- Here, we observe the maximum memory size provided by operating system by checking the VmPeak size at **"/proc/<pid>/status"**
- The memory overheads for ours are 16.2x and 17.6x for NPB3.3 and SPEC CPU2006
  - This overhead is comparable to the overhead of serial SD^3 method, 13.4x in SPEC CPU2006
  - Our profiling can detect data dependencies by scalar memory accesses in addition to these by array based data accesses

Our profiling can realized within a reasonable overhead compared with the stride compression based SD^3 method

Validating the obtained LCCT+M

- Visualize the LCCT+M focusing only on hot spots
  - We set the number of executed instructions as a criteria of the threshold



Circle node: Loop
Box node: Procedure call
Solid line Edge: Control flow
Dashed Red line Edge:
Data dependency between nodes
Dashed Blue line Edge:
Dependency btw loop itrerations

The LCCT+M of IS (Thr=1.66%).

- Compare the obtained LCCT+M with source code

There are seven hot loops (loop-4, 6, 8, 10, 13, 14, 16)

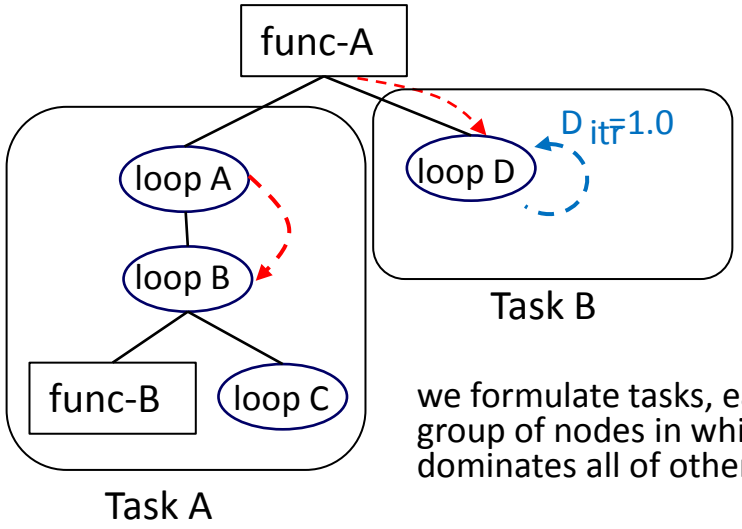
loop-4, 14, 16 are not dependent upon themselves => potential loop-level parallel regions

As obtained results consist with the original source code, we can confirm that ours successfully extract and visualize the run-time memory dataflow

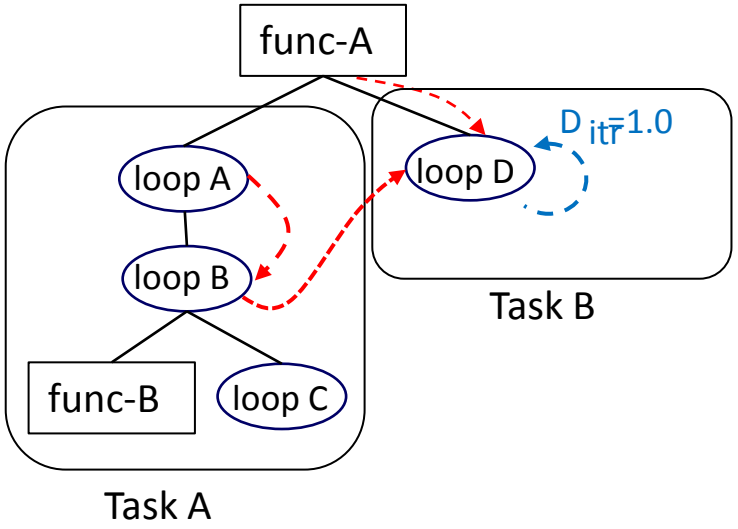
Unveiling various parallel regions

To unveil loop-, task-, pipeline-parallelism,
we check dependencies across regions of LCCT+M

- In LCCT+M, data dependencies are represented by edges between its source node to the sink node.



we formulate tasks, each of which is a group of nodes in which a parent node dominates all of other nodes



Task parallelism

- If there are no dependence, then these are a potentially task-level parallel region

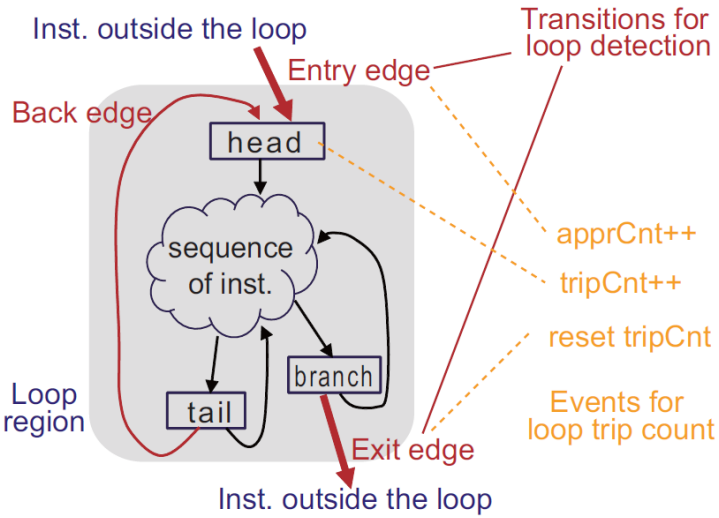
Pipeline parallelism

- If the two tasks are dependent via an edge for one particular direction, these are potentially pipeline-parallel tasks

Unveiling loop-level parallelism

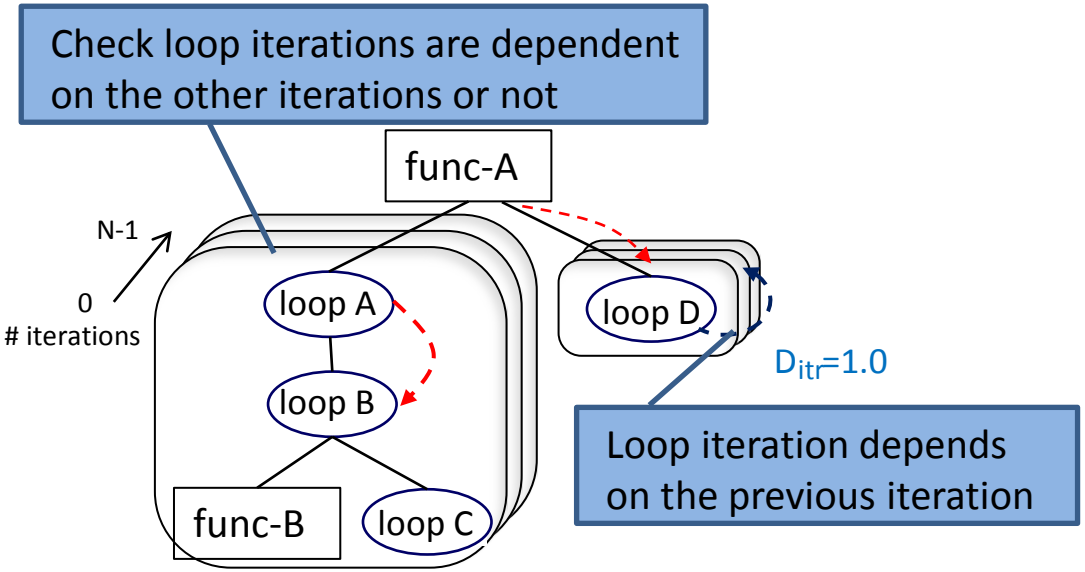
Loop-Level parallelism

- We need to identify data dependencies among loop iterations in addition to among loop regions
- By counting loop trip count, we profile potential loop-level parallel regions



Monitor head instructions & count tripCnt

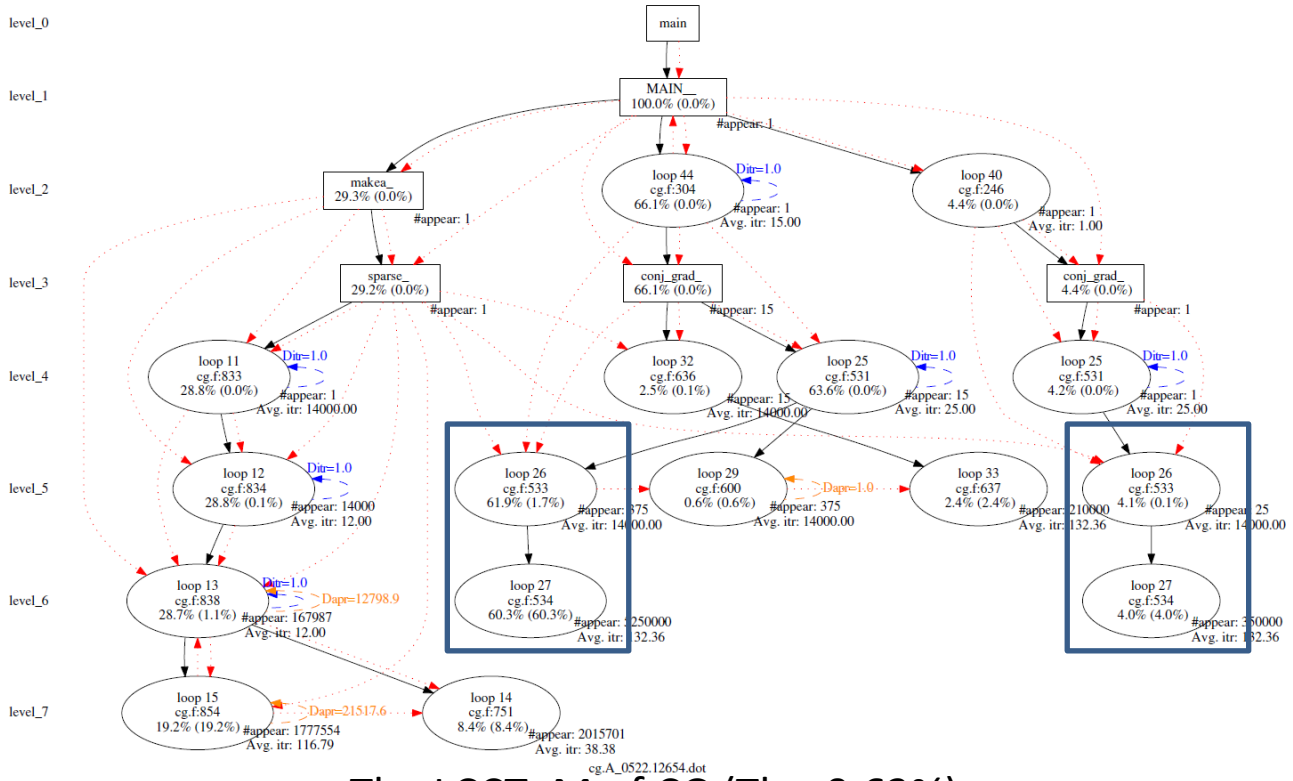
(a) How to detect loop iterations



Check dependencies among loop iterations

(b) Data dependencies among loop iterations

Identifying loop-level parallelisms

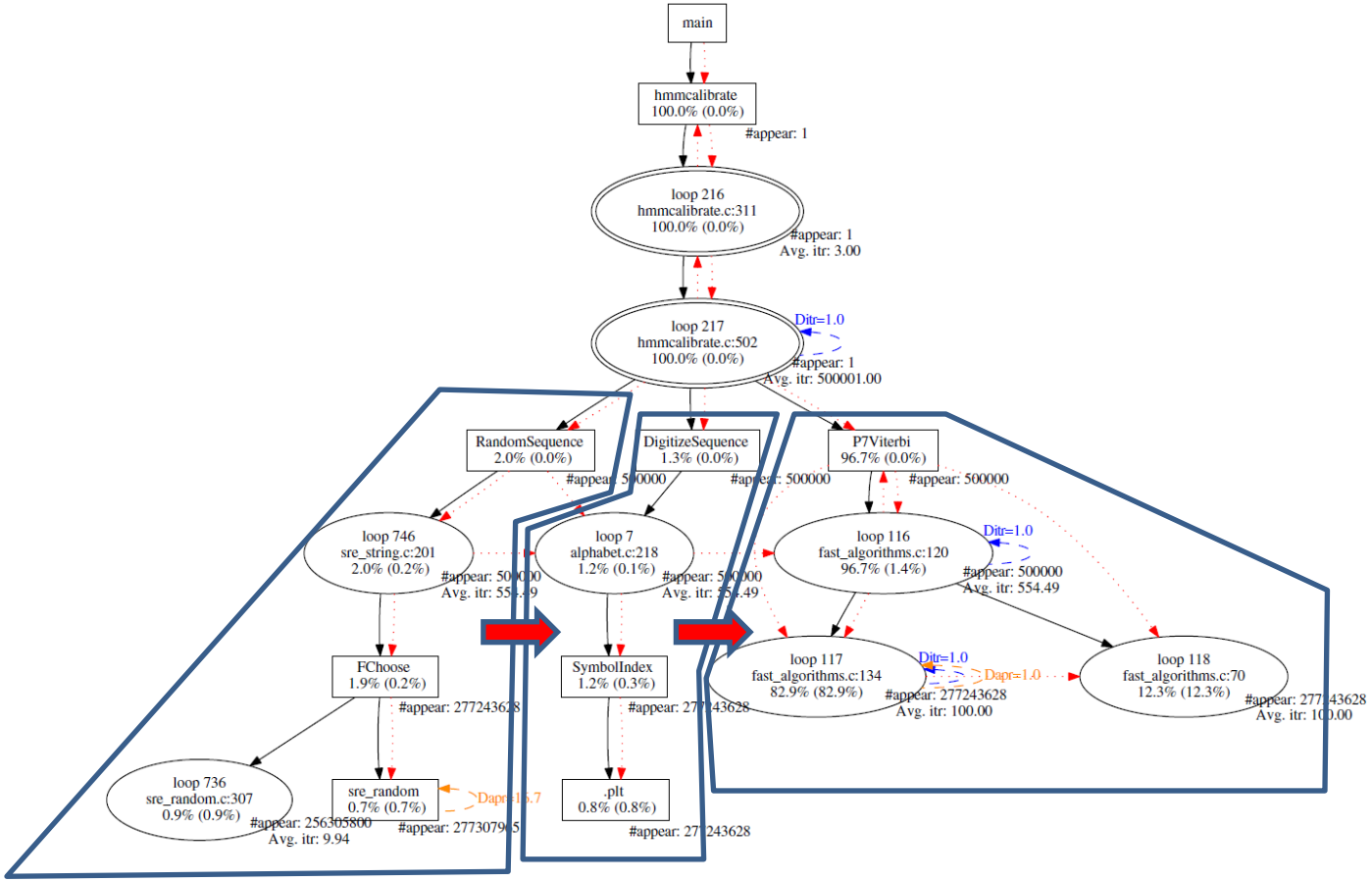


The LCCT+M of CG (Thr=0.63%)

- In loop nests of (loop-26, 27) and (loop-32, 33), each nodes is independent from the others

We find that we can profile potentially parallel loops from nested loops

For identifying task-, and pipeline-parallelisms



The LCCT+M of 456.hmmer [input data=retro.hmm] (Thr=0.74%).

We find that we can form tasks and profile potentially pipeline-parallel tasks

Conclusions

- We have presented a dynamic data dependence profiling technique
  - For analyzing various types of parallelisms across the whole program execution
  - Transparent profiling on a dynamic binary translator
  - A locality-aware table structure with paging
  - LCCT+M (Loop-Call Context Tree with Memory dataflow)
- We have implemented our mechanism and evaluated it using NPB and SPEC CPU 2006 benchmark suite.
  - From the results, we have confirmed that we can successfully extract data dependencies within reasonable time and space overheads
  - We also have visualized the LCCT+M using its hot spot and confirmed that we can identify potential loop-, task-, and pipeline-parallelisms from arbitrary binary codes
- For future work,
  - we are planning to apply our mechanism to speculative loop parallelization and runtime loop transformation for further productive utilization of parallel computing system