On-the-fly Detection of Precise Loop Nests across Procedures on a Dynamic Binary Translation System

Yukinori Sato\textsuperscript{(1)}
Yasushi Inoguchi\textsuperscript{(1)}
Tadao Nakamura\textsuperscript{(2)}

\textsuperscript{(1)} JAIST (Japan Advanced Institute of Science and Technology)
\textsuperscript{(2)} Keio University

CF’11 May 4, 2011
1. Introduction

Application programs continuously increase their scale and complexity to add new valuable features

For programmers
• finding available parallelism from such programs becomes major burdens

Programs become large and complex

Still “Loop structures” are
• fundamental to extract available parallelism for highly parallelized multicore systems
• difficult to extract from real apps because some loops are deeply nested across procedures

Currently, finding inter-procedural loops without reading source codes is not well established

To extract loop-level info without sacrificing human resources is critical for future system
The way to extract loop level information

Useful loop-level info includes:
- dynamic control flow, inter-procedural loop nesting, loop trip counts
These are detected at runtime but not detected statically.
So, we need a framework that detects such dynamic information.

To realize such framework, we focus on Dynamic binary translation (DBT)

The advantages of DBT include:
- both flexibility of SW and high-speed capability of HW
- transparent binary instrumentation
- also, DBT can be applied for dynamic binary optimization and parallelization

<table>
<thead>
<tr>
<th>Interpreter</th>
<th>Dynamic Binary Translator</th>
<th>HW scheduler</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flexible, Easy to change and improve</td>
<td>Rigid and inflexible Hard to change and improve</td>
<td>Native execution</td>
</tr>
<tr>
<td>Overhead for non-native execution</td>
<td>Reduces dispatch and execution costs by keeping already transformed insts.</td>
<td>All instructions are supported by their microarchitecture</td>
</tr>
<tr>
<td>Require interpretation overheads for all insts.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
In this paper

- We propose a novel loop detection mechanism
  - extract precise loop structure including inter-procedural nesting
  - detect on the fly without any compiler support using DBT features
  - reduce overheads of loop detection by making precise markers

This contributes our future goal that is to boost parallel processing of various loops

input
Precompiled binary Code

Dynamic binary translator (Pin)

Static Analysis  Runtime Analysis

Our loop monitoring tool

output
- Dynamic behavior of nested loops across procedures

automatic parallelization on DBT system

useful hints for manual optimization and parallelization

JAIST -- Japan Advanced Institute of Science and Technology
Outline of this paper

1. Introduction
2. Existing runtime loop detection
3. Extraction of precise loop structures
4. Experiment
5. Conclusions
2. Existing runtime loop detection

The existing detection assumes a branch with a negative offset always forms a loop.

Loop detection method presented by Tubella and Gonzalez [HPCA98]

A loop region is defined by the area between backward branch and its target. In this example, there are two backward branches. Then, we find two loops [T1, B1] [T2, B2]. We call this “TB method” (Target and Backward branch method).

However, we cannot explain how two overlapped loops are nested clearly.
Baseline loop detection

To shape what is nested loop structure, we follow the definition of "natural loops" which classifies loops into disjoint or nested

- In natural loops,
  - a head instruction of the loop must dominate all of instructions in the loop
- We form pseudo natural loops based on TB method
  - by merging the overlapped region to make the head dominate all insts
  - Then, the region of Loop 1 becomes [T1, B2]
  - We call this "TB-m method" (TB method with merging)

Using TB-m method, nested loop structures can be extracted. So, we use TB-m method as a baseline loop detection method
3. Extracting precise loop structures

Here, we confirm the assumptions of TB-m method

Assumption of TB-m
- all of loops are natural loops (≒ reducible loops)
- instructions between backward branches and their targets form loop region

Actual programs
- there are **irreducible loops** that do not belong to reducible loops
- Branches with negative offsets do not always make loop structure

Loop detection without considering these actual facts leads misunderstandings of loop behavior

Then, we present **precise marker method (pMarker method)** as an efficient loop detection method on DBT system
Precise marker method

The **pMarker method** attempts to detect both of reducible and *irreducible loops* based on *precise control flow analysis*

Here, we explain how the **pMarker method** works:

1. Formulate the loop structures in a procedure using Havlak’s algorithm
2. Generate precise loop markers
3. Insert markers as analysis codes of DBT system
4. Run the binary code with the loop analysis codes
Irreducible loops and Havlak’s algorithm

As the first step, we build CFG of the code, and search dominators, and extract loop structures by Havlak’s algorithm **

** Havlak’s algorithm is an algorithm that identifies precise loop structures including both of reducible and irreducible loops

After the loop formulation, we generate precise loop markers that point out whether the execution is inside loops or not.

To make loop markers effectively, we briefly explain loop structures

- A loop consists of head, tail, and other instructions
- There is at least one back edge to head, one entry edge that enters the loop region from the outside, and one exit edge that exits from the loop region
Monitoring loops using typical loop characteristics

We attempt to make use of characteristics of loops to monitor precise loop structures with low overheads.

Focus on the following characteristics:
- Control flow transitions caused by the enter edges always occur whenever a loop begins its execution.
- Control flow transitions caused by the exit edges always occur whenever a loop terminates.

**Note that we can monitor irreducible loops using these characteristics.**

Focusing on the entry edges and the exit edges, we can generate simple markers that monitor loops.
Tracking loops across multiple procedures

We make use of **Loop-call context tree** representation

- **Calling context tree (CCT)**
  - can represent *unique* call chains in its underlying CFG
  - however, loops are not considered in the CCT

- **Loop-call context tree (L-CCT)**
  - add loop nodes into CCT
  - can represent sequences of activated loops and procedures

(a) call flow graph  (b) call context tree

(c) Loop-call context tree
Details of L-CCT generation

We use leftmost child right sibling binary tree representation to handle tree representation effectively.
4. Experiment

We implement the pMarker method on Pin tool set, and evaluate it using SPEC CPU benchmark programs.

System configuration
- SGI AltixXE320 composed of two Intel Xeon E5462 CPUs, 8GB memory
- Red Hat Enterprise Linux Server 5.2.

Benchmark programs
- 9 programs from SPEC CPU2006
- 2 program from SPEC CPU2000
- we use ref data set
- we compiled using GNU Compiler Collection 4.1.2 with –O3 option

<table>
<thead>
<tr>
<th>Program</th>
<th>Benchmark</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>429.mcf</td>
<td>INT, 2006</td>
<td>C</td>
</tr>
<tr>
<td>433.milc</td>
<td>FP, 2006</td>
<td>C</td>
</tr>
<tr>
<td>434.zeusm</td>
<td>FP, 2006</td>
<td>fortran</td>
</tr>
<tr>
<td>437.leslie3d</td>
<td>FP, 2006</td>
<td>fortran</td>
</tr>
<tr>
<td>450.soplex</td>
<td>FP, 2006</td>
<td>C++</td>
</tr>
<tr>
<td>462.libquantum</td>
<td>INT, 2006</td>
<td>C</td>
</tr>
<tr>
<td>465.tonto</td>
<td>FP, 2006</td>
<td>fortran</td>
</tr>
<tr>
<td>470.lbm</td>
<td>FP, 2006</td>
<td>C</td>
</tr>
<tr>
<td>482.sphinx3</td>
<td>FP, 2006</td>
<td>C</td>
</tr>
<tr>
<td>191.fma3d</td>
<td>FP, 2000</td>
<td>fortran</td>
</tr>
<tr>
<td>301.apsi</td>
<td>FP, 2000</td>
<td>fortran</td>
</tr>
</tbody>
</table>
Overview of our loop monitoring tool

Dynamic binary translator

Precompiled binary Code

Static Analysis
- Analyzed at image load time
- Build CFG
- Search dominator

Runtime Analysis
- Analyzed at execution time
- Monitor dynamic loop behaviors
- Create L-CCT

Optimization Parallelization
- Loop unrolling
- Software pipelining
- Data prefetch

Output
- Dynamic behavior of nested loops across procedures
- Obtain L-CCT

** Control transition caused by indirect branches are not monitored in this experiment.

We also implement the TB-m method to compare its accuracy & overhead
Our Dynamic Binary Translation system

We use Pin tool set as a DBT system

Results of static analysis phase

It is observed that there are a lot of irreducible loops even in structured program written by current widely used languages

<table>
<thead>
<tr>
<th>Programs</th>
<th>Func. calls</th>
<th>Loops</th>
<th>Irr. loops</th>
</tr>
</thead>
<tbody>
<tr>
<td>429.mcf</td>
<td>34</td>
<td>53</td>
<td>22</td>
</tr>
<tr>
<td>433.milc</td>
<td>245</td>
<td>417</td>
<td>86</td>
</tr>
<tr>
<td>434.zeusm</td>
<td>87</td>
<td>564</td>
<td>110</td>
</tr>
<tr>
<td>437.leslie3d</td>
<td>31</td>
<td>450</td>
<td>31</td>
</tr>
<tr>
<td>450.soplex</td>
<td>1045</td>
<td>889</td>
<td>212</td>
</tr>
<tr>
<td>462.libquantum</td>
<td>106</td>
<td>105</td>
<td>16</td>
</tr>
<tr>
<td>465.tonto</td>
<td>4093</td>
<td>10274</td>
<td>464</td>
</tr>
<tr>
<td>470.lbm</td>
<td>29</td>
<td>27</td>
<td>16</td>
</tr>
<tr>
<td>482.sphinx3</td>
<td>342</td>
<td>607</td>
<td>63</td>
</tr>
<tr>
<td>191.fma3d</td>
<td>471</td>
<td>1690</td>
<td>195</td>
</tr>
<tr>
<td>301.apsi</td>
<td>108</td>
<td>318</td>
<td>47</td>
</tr>
</tbody>
</table>
Loops in source and binary codes

We compare loops in source and binary codes of subroutine solve_ (fma3d)

(a) Loops in source codes and their structures
(b) Detected loops by staticAna

We observe that loop structures detected in the staticAna phase is just consistent with that in the original ones yet there are irreducible loops.
Source code and its CFG in solve_ (191.fma3d)

We compare loop regions in source and detected CFG by staticAna

(a) Nested loops in source code
There are two nested loops and 3 if statements

(b) Detected CFG by static analysis
There are four backward branches in CFG

# of Loops
Four pseudo natural loops in TB-m while only two loops in pMarker & src code

Loop regions
An assumption of loop region in TB-m method is incorrect (ex. bbl-128)

Therefore, TB-m is less accurate than pMarker method
Dynamic loop analysis

We checked whether the simple markers are enough to monitor loop nest behavior.

From the results, we made sure that we can **successfully** monitor the on-the-fly loop nest using pMarker method.
Dynamic hot loop behavior

To highlight the typical behavior of the execution, we attempt to identify hot spots of execution by counting the # of insts in each region.

Here, we assume a region that occupies more than 1% of the total executed instruction as a hot region.

<table>
<thead>
<tr>
<th>Program</th>
<th># of loop nodes</th>
<th># of hot loop nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>pMarker</td>
<td>TB-m</td>
</tr>
<tr>
<td>429.mcf</td>
<td>220</td>
<td>633996</td>
</tr>
<tr>
<td>433.milc</td>
<td>709</td>
<td>1639</td>
</tr>
<tr>
<td>434.zeusm</td>
<td>668</td>
<td>1259</td>
</tr>
<tr>
<td>437.leslie3d</td>
<td>337</td>
<td>517</td>
</tr>
<tr>
<td>450.soplex</td>
<td>392</td>
<td>1619</td>
</tr>
<tr>
<td>462.libquantum</td>
<td>148</td>
<td>626</td>
</tr>
<tr>
<td>465.tonto</td>
<td>12211</td>
<td>54250</td>
</tr>
<tr>
<td>470.lbm</td>
<td>13</td>
<td>37</td>
</tr>
<tr>
<td>482.sphinx3</td>
<td>611</td>
<td>19756</td>
</tr>
<tr>
<td>191.fma3d</td>
<td>257</td>
<td>1063</td>
</tr>
<tr>
<td>301.apsi</td>
<td>944</td>
<td>1414</td>
</tr>
</tbody>
</table>

The number of hot loop nodes detected in L-CCT

It is observed that the # of hot loop nodes is decreased dramatically compared with the # loop nodes.
Next, we examine how many irreducible loops are executed at runtime

<table>
<thead>
<tr>
<th>Program</th>
<th>Irr. loops</th>
<th>Hot irr. loops</th>
</tr>
</thead>
<tbody>
<tr>
<td>429.mcf</td>
<td>177</td>
<td>14</td>
</tr>
<tr>
<td>433.milc</td>
<td>83</td>
<td>15</td>
</tr>
<tr>
<td>434.zeusm</td>
<td>180</td>
<td>3</td>
</tr>
<tr>
<td>437.leslie3d</td>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>450.soplex</td>
<td>97</td>
<td>4</td>
</tr>
<tr>
<td>462.libquantum</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>465.tonto</td>
<td>1897</td>
<td>34</td>
</tr>
<tr>
<td>470.lbm</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>482.sphinx3</td>
<td>129</td>
<td>7</td>
</tr>
<tr>
<td>191.fma3d</td>
<td>69</td>
<td>7</td>
</tr>
<tr>
<td>301.apsi</td>
<td>94</td>
<td>33</td>
</tr>
</tbody>
</table>

The # of irreducible loops detected by pMarker method

We observe that there are a lot of irreducible loops in actual dynamic execution and some of them are appeared even in the hot regions
Hot regions of the L-CCT

Here, we visualize the L-CCT representation of fma3d using graphviz

These structures are consistent with the original source codes.

There are no triple-nested loops in the original source codes of solve.

pMarker method

Across the L-CCT, it can be found that the depth of loop nests of TB-m tends to be deeper than that by pMarker.

TB-m method

From these results, we can see that the TB-m method are less accurate than the pMarker method.

JAIST -- Japan Advanced Institute of Science and Technology
**Overheads for analysis on DBT**

We evaluate the overhead of loop analysis incurred by the DBT system.

<table>
<thead>
<tr>
<th>Program</th>
<th>Native</th>
<th>pMarker</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>total</td>
<td>staticAna</td>
</tr>
<tr>
<td>429.mcf</td>
<td>708</td>
<td>2641</td>
</tr>
<tr>
<td>433.milc</td>
<td>916</td>
<td>2766</td>
</tr>
<tr>
<td>434.zeusm</td>
<td>834</td>
<td>1425</td>
</tr>
<tr>
<td>437.leslie3d</td>
<td>842</td>
<td>2278</td>
</tr>
<tr>
<td>450.soplex</td>
<td>312</td>
<td>1246</td>
</tr>
<tr>
<td>462.libquantum</td>
<td>1191</td>
<td>4853</td>
</tr>
<tr>
<td>465.tonto</td>
<td>727</td>
<td>8040</td>
</tr>
<tr>
<td>470.lbm</td>
<td>1699</td>
<td>2287</td>
</tr>
<tr>
<td>482.sphinx3</td>
<td>962</td>
<td>5675</td>
</tr>
<tr>
<td>191.fma3d</td>
<td>145</td>
<td>416</td>
</tr>
<tr>
<td>301.apsi</td>
<td>151</td>
<td>602</td>
</tr>
<tr>
<td>Average</td>
<td>771.6</td>
<td>2929.9</td>
</tr>
</tbody>
</table>

Native execution time and analysis time in seconds.

It is observed that
- The time required for static analysis is several order of magnitude smaller than that for whole the analyses.
- Compared with native execution, pMarker is 4 times slower
- The total overhead of pMarker is 3.6 times faster than TB-m
Conclusions

- We have presented pMarker (precise marker) method that dynamically detects precise loop structures on DBT.
  - That enable us to monitor the inter-procedural dynamic behavior of loops during execution.
- We have proposed L-CCT (loop-call context tree) representation to depict the exact paths of loops and procedure calls.
- We have implemented our method using Pin tool set (one of DBT systems).
  - Evaluated using SPEC CPU 2006/2000 benchmark suite
  - Demonstrated that our loop detection successfully monitor precise loop structures with low overheads
- Future work
  - planning to apply our precise loop detection to run-time data dependence analysis