Identifying Program Loop Nesting Structures during Execution of Machine Code

Yukinori SATO†‡, Yasushi INOGUCHI†, and Tadao NAKAMURA††, Members

SUMMARY This paper presents a mechanism for detecting dynamic loop and procedure nesting during the actual program execution on-the-fly. This mechanism aims primarily at making better strategies for performance tuning or parallelization. Using a pre-compiled application executable machine code as an input, our mechanism statically generates simple but precise markers that indicate loop entries and loop exits, and dynamically monitors loop nesting that appears during the actual execution together with call context tree. To keep precise loop structures all the time, we monitor the indirect jumps that enter the loop regions and the setjmp/longjmp functions that cause irregular function call transfers. We also present a novel representation called Loop-Call Context Graph that can keep track of inter-procedural loop nests. We implement our mechanism and evaluate it using SPEC CPU2006 benchmark suite. The results confirm that our mechanism can successfully reveal the precise inter-procedural loop nest structures from all of SPEC CPU2006 benchmark executions without any particular compiler support. The results also show that it can reduce runtime loop detection overheads compared with the existing loop profiling method.

key words: dynamic loop nests, loop-call context tree, on-the-fly loop detection

1. Introduction

Traditionally, loops in programs have been fundamental structures in the extraction of parallelism. Specifically, frequently executed loops whose bodies become a time-consuming part of execution are important for highly efficient parallelized processing. Some of the loops are deeply nested, and these are sometimes lying across procedure calls. Since these loop structures contain a wide variety of parallelism such as data-level, task-level and pipeline parallelism, we need to understand these structures from innermost loops to outermost loops throughout a program in order to explore unknown room for parallelization and reveal more opportunities for performance tuning.

On the side of application programs, the scale and complexity of programs are increased continuously, especially in the area of production-level applications [1]. Today, it is common to find over 10,000 lines of source code in such applications as seen in SPEC2006 benchmark programs [2]. Under these circumstances, programmers must understand loop structures by reading them carefully to make plans for performance tuning or optimization. Since finding their existence without reading all lines of the source codes is not an established process, programmers must manually identify loop structures, some of which are nested across multiple procedure calls or file boundaries. Also they need to unveil data dependencies or parallelism across them. These processes require numerous efforts for programmers, and still remain one of hurdles in high performance computing [3].

In this paper, we present a novel technique that identifies precise nests of loops across multiple procedure calls on the fly. Here, the preciseness of loop structures comes from the following two analysis; intra-procedural static nests of loops and inter-procedural dynamic nests of loops.

In order to define a precise loop structure, we first build a control flow graph (CFG) for each procedure call from pre-compiled executable machine code. Then, the obtained CFG is inputted to the Havlak’s algorithm [4]. As a result, we obtain accurate loop structures within a procedure call. While Havlak’s algorithm strictly identifies loop regions from any code with arbitrary control flows, it is limited within a single procedure call.

To detect precise loop nest structures across procedure calls, we need to keep track of loop nest structures during the execution. Then, their precise nests lying across multiple procedure calls are monitored during the actual code execution by the dynamic profiling. This profiling enables to reveal the actual context of program execution based on precise loop nest structures that reflects input data sensitivity or dynamic control flows. Because input data sensitivity and dynamic flow of control, which are decided at run-time, deeply affect the actual execution behaviors, it is important to understand these to realize performance tuning of optimized codes generated by production compilers.

Our loop detection mechanism enables us, without accessing the source code, to precisely imagine how loop nests behave across procedure calls during the actual execution, which loop regions become hot spots and where we should focus for performance optimization and tuning. The primary motivation of this work is to use our analysis for developing better strategies for performance tuning or parallelization, and to feed our results to the wide range of tuning/parallelization schemes performed by programmers, compilers, and binary translation systems.

The most important one that we expected to contribute in terms of performance tuning is loop transformation. Traditionally, loop transformation is performed by skilled programmers or tuners [5]. Even in a typical profile-driven
performance tuning process by programmers, detecting hot spots and understanding their loop structures without reading code carefully are known to be helpful for them [6]. Our precise results that show precise loop nest structures and their weights in the execution help make better strategies for loop transformation.

Also, our precise results can be directly used for static optimization/parallelization by compilers. In a feedback driven optimization process, our precise results can contribute to making better strategies of loop transformation performed by compilers. Furthermore, it can be applied to transparent performance tuning based on a binary translator performed by compilers. Furthermore, it can be applied to loop transformation and optimizations, the loop structures in source code do not always exactly correspond to those in machine code.

In addition to the loop structures that reflect compiler’s optimization results, we need to identify dynamic and inter-procedural loop behaviors based on dynamic profiling technique. Since the translation processes of compilers are based on static analysis, the dynamic control flows, which are eventually decided at runtime cannot be obtained from the static phase while these are known to be important criteria for many optimizations [10]. Examples of such dynamic information are the loop trip count and inter-procedural loop nests.

The loop trip count is the number of iterations of a loop, and is essential for determining loop optimization to be performed [11]. Inter-procedural loop nests are reflected by input-sensitive dynamic control of the execution, and can be obtained during the machine code is executed. Since loop trip counts and loop nest structures appearing in the actual code execution indicate the actual locality of data access which directly affect cache-hit ratios, i.e., the performance of code execution, we should focus on the loops in the code execution in order to utilize this information for making better strategies for performance tuning or effective parallelization.

The precise loop structures during the actual code execution need to be identified because the imprecise detection of loop structures would potentially affect the performance tuning opportunities. Several tools that analyze serial code and provide useful information for performance tuning and parallelization have been introduced in the recent past [6]–[8], [12]. However, their loop detection mechanisms are neither precise nor inter-procedural.

If such tools could be coupled with precise inter-procedural loop nest information, the productivity and capability of these tools for performance tuning could be enhanced. Because the precise loop nest structure is essential information for loop transformation techniques [5], [6], we should profile their weights in the execution accurately to help make better strategies for loop transformation. Also, dynamic data dependence profiling such as LCCT+ M [13] has been developed, and it is stated that precise loop extraction is a key component for accurate dependence profiling.

2. Loop Nesting in Source Code, Machine Code, and the Execution

There are several gaps among source code, machine code, and the code execution when we attempt to identify precise loop structures. In this Section, we briefly explain the reason why structural gaps exist in loop structures extracted by each layer.

Loop structures are originally formulated by programmers in the source code. Then, the source code is fed to a compiler, and translated into the machine code. To optimize and tune the code execution, compilers often perform function inlining, loop inlining, fused loops, and other loop transformations. Due to the fact that the original source-level loop structures are often reorganized by such aggressive transformation and optimizations, the loop structures in source code do not always exactly correspond to those in machine code.

The backward branch, which makes the flow of execution cyclic, is a key feature that forms loop structures. Therefore, many computer architects assume that a branch with a negative offset referred to as a backward branch always forms a loop, and have presented dynamic loop detection...
schemes based on this assumption [14], [15].

Figure 1(a) shows the dynamic loop detection methods presented in [15]. We call this method TB method. There are two backward branch instructions (B1 and B2) and their own target instructions (T1 and T2). Here, we assume that a loop head instruction corresponds to the target of a branch instruction, and a loop tail instruction corresponds to the backward branch instruction which locates at the end of the loop region. Here, a loop region is defined as the area between the backward branch instruction and its target instruction. So, we treat dynamically executed instructions between T1 and B1 as belonging to a loop [T1, B1]. In this example, we find two loops, that is Loop 1 [T1, B1] and Loop 2 [T2, B2], and these two are nested.

3.2 The Baseline Nested Loop Detection

Figure 1(b) shows a situation where two backward branches are overlapping. Here, it is not clear how each of the overlapped loops is nested because these are partly nested but partly disjoint. To clarify the shape of a nested loop structure, we follow the definition of natural loops, that is, a head instruction of the loop must dominate all of instructions in the loop [16]. Since this definition introduces a property in which any possible combinations of loops can be classified into disjoint or nested loops, we can clarify the loop nest structures in any given cases.

Figure 1(b) shows pseudo natural loops that merge the disjoint region of these loops [B1, B2] into Loop 1 in order to follow the definition of natural loops. By this merging of the region, Loop 1 region becomes [T1, B2]. Since there are no flow-in edges outside the loop except for the head, the head of Loop 1 apparently dominates all instructions in this loop. What is more, since the Loop 2 [T2, B2] completely lies inside Loop 1, the nested structure of these loops becomes clear.

Here, we detect pseudo natural loop regions based on the following assumption: In the loop, all instructions between the loop head and the tail are always dominated by the loop head instruction, i.e., all of the entries to the loop are from the loop head instruction. This is equal to the condition for reducible loops. In this method, pseudo natural loops are formulated by merging disjoint regions into outer loops. Afterwards, we can observe loop nests clearly. Because this method reconstructs the loop structures of TB method by merging the disjoint regions, we call it the TB-m method, and use it as the baseline method for conventional loop detection methods.

Moseley et al. proposed the LoopProf in [7] that monitors the repetitive appearances of the dynamic basic blocks. Since Moseley’s LoopProf maintains a stack of dynamically executed BBLs, a loop occurs when a BBL on the stack is executed again. This is actually the same as the one that detects natural loops. Here, we assume that these are similar to the TB-m method, and regard the TB-m method as a reference model in the later Section.

4. Precise Marker Method

As discussed in Sect. 3, the conventional methods assume that backward branch instructions form loop regions and all loops are reducible (natural) loops. However, in actual programs these assumptions are not necessarily true. If there are a significant number of backward branches that do not form cyclic loop structures, or if a large number of irreducible loops in actual programs, the TB-m method will not form precise loop nest structures of the actual execution.

In this Section, we present a precise marker method (pMarker method) as a precise, efficient nested loop detection method. Figure 2 shows an overview of the pMarker method. The light gray area shows our Binary Translation (BT) system where transparent binary instrumentation is performed at runtime, and each dark gray area shows the modules of the analysis. Later in this Section, we explain the details of how the pMarker method works in a dynamic BT system and how we realize it with low overheads.

4.1 Forming Precise Loop Structures

The first phase of our pMarker method is the static analysis phase shown in Fig. 2. First, we load an executable binary file and analyze its control flows and build basic blocks for each procedure call. Then a CFG, where each node corresponds to a basic block and each edge corresponds to the control flow, is statically built. If there exists a node with a cyclic path reachable from every other node in a CFG, that
is a loop. A loop is formally defined by the notation of a strongly connected region. Here, we focus on Havlak’s algorithm which has ability to identify nests of loops based on that definition [4]. Since Havlak’s algorithm can handle both of reducible and irreducible loops, we can identify precise loop regions and their nests from any machine code with arbitrary control flows †.

Here, a reducible loop is defined as a loop with a single entry. Therefore, a unique entry node for the loop dominates all the other nodes of the loop in its CFG. This unique entry node is referred to as the header of the loop. Conversely, an irreducible loop is a loop with multiple entries. In this case, a single entry node does not cover all the entry nodes.

In order to monitor loop structures and loop nests with low runtime overheads, we need to prepare simpler and fewer markers able to indicate precise dynamic loop behavior. We focus on the two important transitions of control flows that determine whether the execution is within a loop: entry edge and exit edge. Figure 3 shows the structure of a typical reducible loop. A loop consists of a head instruction, a tail instruction, and other instructions. In addition, there are at least one back edge to the header, one entry edge that enters the loop region from the outside, and one exit edge that exits from the loop region. Here, the control flow transitions caused by the entry edges always occur whenever a loop begins its execution. Similarly, whenever a loop terminates, we encounter the transitions caused by the exit edges.

Given these facts, we propose that precise loop structures and their nests should be monitored by focusing on the appearances of entry and exit edges. To monitor these, we prepare markers that specify these edges and their corresponding instructions. For irreducible loops, we prepare markers that monitor all of the entry edges to the loop because there are multiple loop entry edges.

To implement the proposed mechanism in a dynamic BT system, we perform static analysis of control flows within a procedure call when the procedure is loaded from the binary image for the first time. After extracting the precise loop structures in a procedure call, we generate loop markers. In order to track the precise loop nest structures together with function calls, we also prepare markers for call and return instructions. Soon after these are generated, analysis codes for monitoring their dynamic behavior are inserted at the location indicated by the markers. These analysis codes are executed together with the application binary code at the runtime analysis phase as shown in Fig. 2. As a result, we obtain the precise loop nest structures during the code execution.

We note that previous works, such as Moseley’s [7], did not perfectly handle irreducible loops. Irreducible loops could show up even if these were not considered. In those cases, a single irreducible loop tends to show up as multiple different imprecise loops, which lead to misunderstandings of dynamic nesting of loops and procedure calls. Additionally, forming precise loop structures using the pMarker method enables us to monitor loop trip counts precisely. Using the knowledge that the head instruction is always executed when the loop is executed, the loop trip count can be monitored by counting the appearances of the head of the loop. We note that the existing loop detection methods cannot find loop head instructions as well as loop trip counts precisely because their head instructions often do not form loops in terms of the control flows.

4.2 Loop-Call Context Tree

To monitor dynamic behaviors of loop nests efficiently, we propose making use of the Loop-Call Context Tree (L-CCT) representation. L-CCT can effectively represent the relationship between nested loops and procedure calls as paths of these nodes. This can also be applied to help us to understand how the program is executed within the system.

Before introducing L-CCT, we briefly explain the conventional techniques for representing the behaviors of procedure calls. A call graph is known as the most basic representation for depicting the flow of procedure calls. Figure 4 (a) shows a call graph for multiple procedure calls. However, using call graphs we cannot observe the exact dynamic path of procedure calls when a procedure is called by multiple different procedures. In this example, we can see that two flows from procedure A and D merges at C, and we cannot observe whether the path main-D-A-C appears or not. This is because the procedure C is called by both the procedure A and D.

To represent flow sensitive path associated with se-
Fig. 5  Detail implementation of a L-CCT representation.

sequences of activated procedures, the calling context profiling has been proposed [17]. In the calling context profiling, Calling Context Tree (CCT) representation is utilized to track sequence of procedure calls. Figure 4 (b) shows a CCT of multiple procedure calls. Here, we can see that the relationship between the caller and the callee is always maintained and the flow sensitive paths of procedure calls are clearly depicted in the CCT. From the CCT, we observe that the path main-D-A appears but main-D-A-C does not appear.

To represent nested loops across procedure calls, we extend CCT and refer it to as Loop-Call Context Tree (L-CCT) representation. In the L-CCT representation, loop nodes are added into the original CCT. Figure 4 (c) shows an L-CCT. Here, a loop node represents one loop region, and each of the nested loops across procedure calls are represented properly in the L-CCT representation.

From the L-CCT, we can see that loops in procedure A are executed only when procedure A is called by procedure main. This is because dynamic control flows such as an if-else statement decide whether the Loop 1 in procedure A is activated or not. Since these behaviors depend on the dynamic control flows of the execution, it cannot be determined at the static analysis phase. Also, we can see that the L-CCT can depict inter-procedural loops across procedure A and C. Therefore, we can understand that L-CCT has the ability to depict complex inter-procedural loop nests in the dynamic execution.

In order to handle the L-CCT representation effectively, we use a left child right sibling binary tree representation [18]. One of the advantages of the left child right sibling binary tree is that we can represent any tree structure using a binary tree. This contributes to reducing search time in the tree. Figure 5 shows how we generate a L-CCT using the left child right sibling binary tree representation. Here, each child node corresponds to either an inner loop node or a callee procedure node. Each sibling node corresponds to a disjoint loop node or a procedure node called by the parent node.

In order to monitor dynamic behavior using the L-CCT, we prepare analysis codes for handling them, which are inserted during the static analysis phase. In runtime analysis phase, we execute the application binary code with the analysis code, and record the dynamic behavior of loops and calls by keeping track of the currently executed region on the fly. If the region appearing to be executed is not found in the tree of the L-CCT representation, we add a new node in it. If the region is found, we simply update the pointer that indicates the current node of the L-CCT. By keeping track of all of the program executions, we obtain the L-CCT of the entire program execution. Also, we note that we keep the history of executed loops using a stack of loop regions for verification. Just like a call stack, an element of the loop is pushed when an entry edge is invoked and pushed when an exit edge is invoked.

4.3 To Track the Precise Loop Stack in a Real Program

The pMarker method implemented in [9] does not always keep the loop stack of an execution precisely due to lack of considerations for indirect jumps and setjmp/longjmp functions. These are known to cause complex flows of control in the actual execution. Next, we describe how we enhanced our previous pMarker implementation to keep track of these behaviors.

First, we reconsidered indirect jumps. At the static analysis phase, CFGs are built to identify loop structures. Since all possible target addresses of indirect jumps cannot be obtained in advance of runtime, building accurate CFGs in the static phase is impossible. Nonetheless, these have the ability to enter a loop region, or form a loop.

Figure 6 shows a loop region and a basic block that ends up with an indirect jump instruction. Here, the nodes represent the basic blocks and the edges represent the control flows. The shaded region corresponds to a loop region composed of bbl-1, bbl-2, and bbl-4. It is observed that the basic block bbl-3 that ends up with indirect jumps is not connected to any other basic blocks in the CFG. This is because we could not make the edges incurred by indirect jumps in the static phase. Therefore, a basic block that ends up as an indirect jump is always located outside of the loop region and does not become a backward predecessor of any predefined loop regions.

This is the reason why loops incurred by indirect jumps in the pMarker method implemented in [9] could not be detected properly. If the target of an indirect jump is inside of the loop, an indirect jump becomes an unmarked loop en-
try edge, and the the consistency of the loop stack cannot be maintained. In this example, the bbl-3 is located in the outside of the loop region. If the control flow of the bbl-2 transits to the bbl-3, the edge is regarded as an exit edge in this example. However, there are still the possibility that indirect jumps could enter loop regions. If the target of this indirect jump is the inside of the loop, we have to recognize it and attempt to maintain the loop stack consistency by complementing an enter edge.

In order to resolve this issue, we manipulate the loop stack by dynamically checking whether the targets of indirect jumps are inside loop regions. If the targets are inside loop regions, then we push the stack of loops enough to reach the level of the current loop region. By monitoring the targets of indirect jumps and manipulating them properly we can keep track of loop stack consistency, even if there are indirect jumps†.

Literature on static binary code analysis and rewriting has demonstrated that some of the indirect jumps can be resolved statically through constant propagation and pattern matching [19]. Of course, we could use those static handling methods of indirect jumps during the static phase of the our analysis. However, in this paper, we use a dynamic technique for detecting loop entry edges caused by indirect jumps to productively monitor all of indirect jumps.

Second, we extend the pMarker method in [9] to keep the consistency of a call stack even in complex function call transitions. Complex function call transitions by setjmp/longjmp are known to break the orderly sequence of calls and returns as discussed in [20]. Therefore, several practical concerns must be addressed to keep the consistency of a loop stack within the call stack.

We handle the irregular manipulations for a call stack caused by setjmp/longjmp functions by allowing them to pop multiple entries to be consistent with the state to which the program is returning. In order to keep a loop stack together with its call stack, we check whether the modified return address of setjmp/longjmp is inside loop regions in the procedure call or not. If it is inside any loop regions, then we push the stack of loops enough to the level of the current loop region, as in the case of indirect jumps. Only when we manipulate loop and call stacks based on the above procedures, we can handle the effects of setjmp/longjmp functions together with their consistency.

We note that irregular calling setjmp functions by dynamically linked shared library and indirect procedure calls are also considered to keep the consistency of the call stack of a sequence of function calls and returns. To handle these, we insert markers for detecting the beginning of procedure calls at the top of the functions.

5. Experimental Results

5.1 Methodology

We implement the pMarker method on Pin tool set [21]. Pin is a well-known dynamic BT system that provides the same ISA translation and is applicable to dynamic binary optimization and parallelization. Figure 2 shows an overview of our runtime loop detection on a dynamic BT system. We implement our mechanism as discussed in Sect. 4.

We also implement the TB-m method on Pin tool set to compare its accuracy and overheads with those of the pMarker method. The TB-m method implemented in this paper is the same as one used in [9].

To validate and evaluate our method, we use all of the programs in SPEC CPU2006 benchmark suite [22]. In addition to CPU2006, we select two programs from SPEC CPU2000 to compare ours with the existing technique such as the loop detection in [7]. We use the GNU Compiler Collection 4.1.2 for x86_64 Redhat Linux as a compiler tool set. The benchmark programs are compiled with the options specified by their default configurations (‘-O2’ in CPU2006 and ‘-O3’ in CPU2000). We also attach the debug options (-g -gdwarf-2)††. Then, we run them using reference data sets (–size=ref). Here, we start the loop analysis when the main function is called, and terminate it soon after the main function returns. Also, we note that the loop analysis is performed only for the executable application binary code but not for the code of shared libraries.

5.2 Static Analysis

At first, we verify and evaluate the output from the static analysis phase of our pMarker method. Table 1 shows a list of benchmark programs, the languages they are written in and the statistics obtained during the static analysis phase. The column of # Calls and # Loops represents the number of procedure calls and loops statically detected from each binary code, respectively. Here, the number of loops contains both the reducible and irreducible loops. The column

††The debug information is used only for obtaining the mapping between the binary code and its source code and it just aims for helping the performance tuning done by programmers. Our precise loop identification can be performed to arbitrary executable binary codes without debug information.
Table 1  Loops detected in static analysis phase.

<table>
<thead>
<tr>
<th>Program</th>
<th>Language</th>
<th># Calls</th>
<th># Loops</th>
<th>iloops</th>
</tr>
</thead>
<tbody>
<tr>
<td>400.perlbench</td>
<td>C</td>
<td>1762</td>
<td>1067</td>
<td>30.1%</td>
</tr>
<tr>
<td>401.bzip2</td>
<td>C</td>
<td>90</td>
<td>181</td>
<td>18.8%</td>
</tr>
<tr>
<td>403.gec</td>
<td>C</td>
<td>4787</td>
<td>4768</td>
<td>33.7%</td>
</tr>
<tr>
<td>429.mcf</td>
<td>C</td>
<td>34</td>
<td>53</td>
<td>41.5%</td>
</tr>
<tr>
<td>2006 CPU</td>
<td>C</td>
<td>2548</td>
<td>1315</td>
<td>31.4%</td>
</tr>
<tr>
<td>456.hmmer</td>
<td>C</td>
<td>509</td>
<td>866</td>
<td>18.1%</td>
</tr>
<tr>
<td>458.sjeng</td>
<td>C</td>
<td>152</td>
<td>240</td>
<td>17.9%</td>
</tr>
<tr>
<td>462.libquantum</td>
<td>C99</td>
<td>106</td>
<td>105</td>
<td>15.2%</td>
</tr>
<tr>
<td>464.h264ref</td>
<td>C</td>
<td>539</td>
<td>1777</td>
<td>13.9%</td>
</tr>
<tr>
<td>471.omnetpp</td>
<td>C++</td>
<td>2556</td>
<td>594</td>
<td>30.3%</td>
</tr>
<tr>
<td>473.astar</td>
<td>C++</td>
<td>108</td>
<td>119</td>
<td>16.0%</td>
</tr>
<tr>
<td>483.xalancbmk</td>
<td>C++</td>
<td>16921</td>
<td>6259</td>
<td>21.0%</td>
</tr>
<tr>
<td>410.bwaves</td>
<td>Fortran</td>
<td>17</td>
<td>89</td>
<td>1.1%</td>
</tr>
<tr>
<td>416.gamess</td>
<td>Fortran</td>
<td>2880</td>
<td>22477</td>
<td>16.9%</td>
</tr>
<tr>
<td>433.milc</td>
<td>C</td>
<td>245</td>
<td>417</td>
<td>20.6%</td>
</tr>
<tr>
<td>434.zesump</td>
<td>Fortran</td>
<td>87</td>
<td>564</td>
<td>19.0%</td>
</tr>
<tr>
<td>435.gromacs</td>
<td>C, Fortran</td>
<td>1251</td>
<td>2037</td>
<td>14.9%</td>
</tr>
<tr>
<td>436.cactusADM</td>
<td>Fortran, C</td>
<td>1322</td>
<td>896</td>
<td>18.0%</td>
</tr>
<tr>
<td>2006 CPU</td>
<td>C</td>
<td>31</td>
<td>450</td>
<td>6.9%</td>
</tr>
<tr>
<td>443.milc</td>
<td>C</td>
<td>112</td>
<td>641</td>
<td>19.7%</td>
</tr>
<tr>
<td>445.dealII</td>
<td>C++</td>
<td>8956</td>
<td>11172</td>
<td>17.4%</td>
</tr>
<tr>
<td>450.soplex</td>
<td>C++</td>
<td>1045</td>
<td>889</td>
<td>23.5%</td>
</tr>
<tr>
<td>455.povray</td>
<td>C++</td>
<td>1752</td>
<td>1299</td>
<td>21.6%</td>
</tr>
<tr>
<td>456.povray</td>
<td>Fortran</td>
<td>1339</td>
<td>4694</td>
<td>17.1%</td>
</tr>
<tr>
<td>459.GemsFDTD</td>
<td>Fortran</td>
<td>108</td>
<td>942</td>
<td>3.3%</td>
</tr>
<tr>
<td>463.tonto</td>
<td>Fortran</td>
<td>4093</td>
<td>10282</td>
<td>4.4%</td>
</tr>
<tr>
<td>470.bwaves</td>
<td>C</td>
<td>29</td>
<td>27</td>
<td>59.3%</td>
</tr>
<tr>
<td>481.wrf</td>
<td>Fortran</td>
<td>2911</td>
<td>8289</td>
<td>6.7%</td>
</tr>
<tr>
<td>482.sphinx3</td>
<td>C</td>
<td>342</td>
<td>607</td>
<td>10.4%</td>
</tr>
</tbody>
</table>

CPU 191.fma3d | Fortran | 471 | 1690 | 11.5% |
| 2000 301.apsi | Fortran | 108 | 316 | 13.0% |

of iloops represents the ratio of irreducible loops compared with the total number of loops. From the results, it is observed that a lot of irreducible loops are generated even in structured programs written by widely used languages and their compiler.

Then, we compare the structures of the loops extracted during the static analysis phase with original ones in source programs. Since all of source programs in the SPEC CPU benchmark suite are too large to compare, we pick up several portions of their source codes and compare them with our output. Figure 7 (a) shows the source-level loop structures in subroutine solve of 191.fma3d. We can see that there are 21 loops in this subroutine, and a loop induced by the goto statement is the outer loop of the several DO loops. Figure 7 (b) shows the detected loops in our static analysis phase. Here, irreducible loops are represented as iloop, self loops as sloop, and the number in the loop nodes correspond to the ID numbers of basic blocks.

From the results, it is observed that the structure of loops detected in the static analysis phase is consistent with that in the original source code. However, there might be situations in which loop structures in source code do not correspond exactly to those in binary code due to aggressive loop transformations and function inlining. Even in those cases, we can map the structures without inconsistency of semantics as discussed in [8]. Also, we note that the loop structures in the machine code is one of the important factors in determining the data access locality and performance.

Next, we compare loop regions in the source code with loop regions detected by the pMarker and the TB-m method.

5.3 Verifying Runtime Loop Nests and Statistics of L-CCTs

Next, we verify and evaluate the output from the runtime...
analysis phase of our pMarker method. Here, we execute the instrumented binary code generated at the static phase and dynamically keep track of the currently executed region together with loop and calling contexts. In order to confirm whether we can monitor the loop behaviors on-the-fly by observing the loop entry and exit edges, we verify the consistency of the loop stack and the L-CCT. For verification, we check whether the currently executed region is always consistent with the current node of the loop stack and the L-CCT.

From the results of the verifications that run all the benchmark programs with their reference data sets, we confirmed that the on-the-fly loop nesting can be monitored successfully without any inconsistency of loop stacks and trees of loop-call contexts. Therefore, the pMarker method is productive enough to analyze realistic programs used in production or industry level applications such as SPEC CPU2006.

Table 2 shows the actual statistics detected by the pMarker method. Here, # Dy.Call and # Dy.Loop represent the number of function call nodes and loop nodes of the L-CCT dynamically detected during the actual execution. Dy.iloop represents the percentage of irreducible loops within the dynamic loop nodes. The # HotLoop represents the number of hot irreducible/reducible loops of the execution. Here, hot loops, which highlight hot spots of execution in terms of loops, are defined as loops that occupy more than 1% of the total executed instructions.

From the results, we observe that it is hard to predict the dynamic behaviors of the execution only by the statistics of the static analysis. In some of the programs, the number of loop or call nodes that appeared dynamically is dramatically increased from that in the static analysis. In some other programs, the number of nodes remained relatively the same. We also find that the frequency of appearance of irreducible loops during the execution differs for each program.

Therefore, the dynamic behavior has to be monitored in addition to the static analysis of the code to obtain the accurate loop behavior.

The results also show that all the numbers of hot loops are less than 100 in all the program executions used in this evaluation. This implies not a quite large number of hot loop nodes appear in the actual dynamic executions and we can understand the outline of dynamic execution by focusing on the hot loops. It is also observed that some of loops in the hot regions are irreducible loops. These results imply that we should not ignore the behaviors of irreducible loops in the process of obtaining accurate loop behaviors.

Next, in order to depict input data sensitivity, we present the differences among input sets within the same programs of SPEC CPU2006. Table 3 shows the dynamic information obtained from programs with multiple reference workload sets. Here, we represent the ID number of the
inputted workload by the number after the program name, where the ID is equal to the original order of the invocation described in the benchmark suite†. From the results, we can observe that some of programs are sensitive to their input set. For example, in 403.gcc and 445.gobmk, there is a wide range of variations of the number of loop nodes across input sets††. The 401.bzip shows different characteristics such that the variation of loop nodes is very small across input sets. These facts imply that our pMarker method has ability to measure the input sensitivity of program executions.

5.4 The Effects of Indirect Jumps, Setjmp/Longjmp

As discussed in Sect. 4.3, we extended the pMarker method implemented in [9] to handle indirect jumps that enter loop regions and setjmp/longjmp functions that cause irregular function call transitions. Here, we verify the effects of these extensions. Table 4 shows the number of indirect jumps and loop entry edges (loopIn) caused by the indirect jumps. The other benchmark programs that do not appear here have no indirect jumps that cause loop entry edges. The column labelled Ratio represents the percentage of indirect jumps that directly jump into the loop regions compared with the total number of indirect jumps that appeared during the execution.

We observe that the indirect jump instructions are executed within the actual programs and some of them occasionally jump to loop regions. Therefore, we can find that for such programs the consistency of loop stacks cannot be maintained without considering indirect jumps toward loop regions.

††We note that Table 2 shows the first reference workload of a program when the benchmark program has several reference workloads.

††If we visualize their L-CCTs of different input data sets, we can obtain the graphical view of different characteristics of them such as the variation of loop nodes and key procedure calls that invoke many loops. Due to space limitations, these L-CCTs are not represented in this paper.

Table 3 Input data sensitivity appeared in different input sets.

<table>
<thead>
<tr>
<th>Program</th>
<th># Dy.Call</th>
<th># Dy.Dylop</th>
<th>Dy.loop [%]</th>
<th># HoIoop (loop / Loop)</th>
</tr>
</thead>
<tbody>
<tr>
<td>401.bzip2.2</td>
<td>296</td>
<td>271</td>
<td>15.9%</td>
<td>20/45</td>
</tr>
<tr>
<td>401.bzip2.3</td>
<td>297</td>
<td>297</td>
<td>16.3%</td>
<td>28/43</td>
</tr>
<tr>
<td>401.bzip2.4</td>
<td>296</td>
<td>244</td>
<td>15.6%</td>
<td>12/40</td>
</tr>
<tr>
<td>401.bzip2.5</td>
<td>296</td>
<td>243</td>
<td>15.6%</td>
<td>12/37</td>
</tr>
<tr>
<td>401.bzip2.6</td>
<td>297</td>
<td>273</td>
<td>15.8%</td>
<td>15/43</td>
</tr>
<tr>
<td>403.gcc.2</td>
<td>2.21E+05</td>
<td>6.81E+04</td>
<td>29.4%</td>
<td>62/88</td>
</tr>
<tr>
<td>403.gcc.3</td>
<td>2.27E+05</td>
<td>7.13E+04</td>
<td>36.8%</td>
<td>40/65</td>
</tr>
<tr>
<td>403.gcc.4</td>
<td>1.64E+05</td>
<td>4.83E+04</td>
<td>32.0%</td>
<td>28/49</td>
</tr>
<tr>
<td>403.gcc.5</td>
<td>1.73E+05</td>
<td>4.80E+04</td>
<td>32.4%</td>
<td>31/56</td>
</tr>
<tr>
<td>403.gcc.6</td>
<td>1.92E+05</td>
<td>5.38E+04</td>
<td>31.7%</td>
<td>32/59</td>
</tr>
<tr>
<td>403.gcc.7</td>
<td>2.64E+05</td>
<td>5.97E+04</td>
<td>32.2%</td>
<td>25/46</td>
</tr>
<tr>
<td>403.gcc.8</td>
<td>1.35E+05</td>
<td>3.77E+04</td>
<td>34.3%</td>
<td>36/61</td>
</tr>
<tr>
<td>403.gcc.9</td>
<td>2.02E+05</td>
<td>5.94E+04</td>
<td>30.9%</td>
<td>65/87</td>
</tr>
<tr>
<td>445.gobmk.2</td>
<td>5.02E+06</td>
<td>4.55E+06</td>
<td>24.9%</td>
<td>47/79</td>
</tr>
<tr>
<td>445.gobmk.3</td>
<td>3.41E+06</td>
<td>3.07E+06</td>
<td>25.0%</td>
<td>51/78</td>
</tr>
<tr>
<td>445.gobmk.4</td>
<td>3.76E+06</td>
<td>3.40E+06</td>
<td>25.0%</td>
<td>49/73</td>
</tr>
<tr>
<td>445.gobmk.5</td>
<td>2.94E+06</td>
<td>2.65E+06</td>
<td>24.7%</td>
<td>28/58</td>
</tr>
<tr>
<td>464.h264ref.2</td>
<td>3661</td>
<td>3421</td>
<td>13.1%</td>
<td>23/81</td>
</tr>
<tr>
<td>464.h264ref.3</td>
<td>3745</td>
<td>3633</td>
<td>13.6%</td>
<td>23/81</td>
</tr>
</tbody>
</table>

Also, we observe whether setjmp/longjmp functions appeared in the actual program execution. From the result, it can be observed that these appear during the execution of 400.perlibench, 416.gamess, 453.povray, 459.GemsFDTD and 471.omnetpp. For these programs, we have to consider longjumps to maintain the consistency of call stacks. We note that since these irregular manipulations of call stacks sometimes break the consistency of loop stacks, we have to adjust the loop stack based on the target of longjmp.

As a result of considerations of the indirect jumps and the setjmp/longjmp functions, we can successfully detect all the control transitions that incur loop entry/exit edges. We can confirm that we could maintain the consistency between loop and call stacks, and trees of these contexts for all of SPEC CPU2006 benchmark program executions. Because we were not able to keep track of a stack of loop nests in several program executions by the implementation of [9], we realized that the consideration of indirect jumps and setjmp/longjmp functions is critical for monitoring dynamic loop nest structures precisely.

5.5 Comparison to the Other Loop Detection Techniques

Here, we compare our pMarker method with the TB-m method and the loop detection method in [7]. For the comparison, we select nine programs from SPEC CPU2006 and two programs from CPU2000, as is evaluated in [9]. Table 5 shows the number of loop nodes in the L-CCT detected by the pMarker and the TB-m method. From the results, we can see that the number of loop nodes detected by TB-m is greater than that by pMarker for all the benchmark programs. This is because the nodes detected by the TB-m method contain imprecise loop nodes, as described in Sect. 5.2. Moreover, an irreducible loop tends to be detected as multiple different loops if we consider only reducible loops. In the worst case, a single irreducible loop could be identified N times where N is the number of entry points that map to the same or nearby source location. Therefore, the dynamic loop information obtained by the TB-m contains a large number of inaccurate loop nodes.

To validate the L-CCTs generated by pMarker and TB-m method, we compare their detected hot regions of 191.fma3d. Figure 9 and Figure 10 show the hot regions generated by the pMarker and the TB-m method, respectively. Here, the circles represent loops, the boxes represent procedure calls, and circles with double lines represent irre-
ducible loops. The numbers inside the loop nodes are the ID numbers for the loops. In each node, after the percentage of the total instructions accumulated over all of its successors, the percentage of the instructions executed in the node itself is represented inside the parenthesis. We note that loop trip counts and appearance counts are also presented in the L-CCT generated by the pMarker method as shown in Fig. 9.

From the results, we can find that the depth of the loop nests of the TB-m tends to be deeper than those of the pMarker due to the effects of irreducible loops. We can also observe the inconsistency in the subroutine solve, which is also examined in Sect. 5.2. As shown in Fig. 9, the loops of the solve detected by the pMarker are nested loops composed of one outer loop and three independent inner loops. This structure is clearly a subset of the statically detected loops; its original source code structure is shown in Fig. 7. On the other hand, as shown in Fig. 10, the loops of the solve detected by the TB-m method are triple-nested loops. This loop structure is not consistent with the loops in static analysis of binary codes as shown in Fig. 7.

It can also be observed that the points where the subroutine internal-forces is called are different between the pMarker and the TB-m method. The point detected by the pMarker method is within the body of a single loop in subroutine solve whereas that by TB-m is within the innermost loop of triply-nested loops. Because the subroutine call is located in the line 583 in the original source code which locates outside the DO loops and inside the loop caused by the goto statement (line 314:614) in Fig. 7 (a), the precise point that calls the subroutine internal-forces is in the body of a single loop as depicted in Fig. 9. From these results, we can observe that the TB-m method is less accurate than the pMarker method.

Moseley et al. also showed the loop-call graph of 191.fma3d in [7]. In the loop-call graph shown in Fig. 7 of their paper, there are six disjoint loops in the subroutine
solve. This structure is different from that of our pMarker method and the original source code. Also, compared with our pMarker method and the original source code, the outer loop caused by the goto statement and the point where the subroutine internal_forces is called is not detected precisely. These imply that because they did not consider the irreducible loops, some of the loops may not have been detected correctly.

5.6 Overheads for Runtime Analysis on a Dynamic BT

Next, we evaluate the overhead of the runtime loop detection incurred by the dynamic BT system. Here, we run our on-the-fly detection on SGI AltixXE320 composed of two Intel Xeon E5462 CPUs, 8GB memory, Red Hat Enterprise Linux Server 5.2, which are the same conditions as for the evaluation in [9].

---

Fig. 10 Hot regions of the L-CCT generated by the TB-m method. (191.fma3d)
The runtime overhead of pMarker and TB-m methods compared with the native time.

Figure 11 shows the runtime overhead of the pMarker method and the TB-m method compared with the native time. Here, we measure the total time required for whole the analysis including the time required in its static analysis phase. Here too, the implementation of the TB-m method is the same as that of [9].

From the results, we find that, on average, the pMarker method can detect precise loop structures with 5.0 times slowdown compared with native execution. We can also find that the runtime overhead of the pMarker method is smaller than that of the TB-m method. On average, the pMarker method is 2.8 times faster than the TB-m method. It is considered that on-the-fly detection with simple markers using an L-CCT representation contributes to these results. One of the advantages of pMarker is that it reduces the cost of searching the nodes to be executed in the L-CCT. To check the node to be executed in the next step is new or already in L-CCT is one of the primary sources of overheads in the loop detection. Since the pMarker method can search the nodes from the children of the currently executed node in the L-CCT representation, we can dramatically reduce the space of searches. On the other hand, the TB-m method must search based on their instruction addresses of loop regions. Therefore, the search of loop regions using indexes of their addresses slows down the profiling.

Here, we note that the original implementation of pMarker method shown in [9] could precisely detect the loop structures with 4.0 times the slowdown on average. Therefore, the consideration for the indirect jump and the setjmp/longjmp handling causes additional overheads compared with the previous implementation without these considerations.

In [7], they showed their loop profiling is on average 22 times slower than the native execution in SPEC CPU 2000 benchmark programs. We consider that the slowdown would be caused by the searches in the stack of instruction trace done by findBPI(). Compared with [7], our pMarker method is considered to be fast, and its search simple.

5.7 Applicability to Dynamic Tuning or Parallelization

Although the proposed pMarker method can monitor dynamic loop nesting faster than other methods, the slowdown compared with native execution is not small especially when executing large HPC applications and it might become unrealistic to profile full scale data sets of them. However, in the case we use the obtained information as hints fed to programmers or compilers, the use case is exactly the same as a profiler of a particular run aiming for making better strategies for performance tuning or parallelization. Then, we can reduce the time for the profiling if we use a smaller data set as is done in the development processes of HPC applications. We note that if the dynamic behaviors depend on the characteristics or the size of input data sets, then the resulting L-CCT might be different from the original one as discussed in Sect. 5.3, but we could adjust the tradeoff between the profiling time and the accuracy.

On the other hand, our mechanism enables to obtain the actual behavior of the code execution that reflects input data and dynamic control flow via dynamic binary transformation mechanism during the code execution [23]. This means that we can obtain performance gain via dynamic tuning or dynamic parallelization without the feedback to the source codes.

 Especially in the case of transparent transformation, the overheads should be avoided to achieve speedup from the original execution. Next, we discuss the two possible directions to amortize or handle this overhead, i.e., the use of the concept of a two-phase dynamic optimization system and the research and development of a light-weight loop detection mechanism by co-design of software and hardware.

The first direction is to use our loop detection only for the first phase of the two-phase dynamic optimization system. As typical dynamic optimization systems use a two-phase system and explore regions to be optimized in the first phase, the overhead will be incurred only for the first phase if we turn off the detection in the second phase. In this case, performance gain through runtime performance tuning or optimization can be expected. Also, the concept of two-phase dynamic optimization system with speculative execution based on checkpointing enables on-demand code customization for the input data set.

The other direction is to improve our detection mechanism by enhancing or making use of the microarchitectural features of CPUs as a co-designed implementation of software and hardware such as [24]. If we implement a co-designed loop detection mechanism using hardware features, we will be able to reduce the overhead dramatically. When a hardware based detection is co-designed, a detection mechanism that consumes less time on a dynamic BT
system is similarly expected to reduce overheads. In addition, based on our accurate detection mechanism, we will be able to investigate how much accuracy of the loop detection is required to realize productive dynamic optimization and parallelization system in future.

6. Related Work

As far as we know, this is the first paper that quantitatively reports the differences between a strict loop definition based on strongly connected regions and a pseudo loop definition based on branches with negative offsets [7], [14], [15]. As is seen from the case in [13] where a precise loop nest information presented in this paper was used when the LCCT+M representation was generated, we believe that our precise loop detection opens a new vista for assisting optimization and parallelization.

The structure of loop nests is one of the important attributes that illustrate how to partition a program into parallel portions and how to map them into multicore systems. There are several papers that attempt to detect loops and apply them to hints for tuning or parallelization. The LoopProf presented in [7] identifies loop structures using binary codes applied to manual parallelization. The primary contributions of this paper over [7] are: (1) using Havlak’s algorithm to extract irreducible loops, (2) inserting simple but precise markers and decreasing overhead dramatically and (3) using L-CCT instead of loop-call graph. They also presented the LoopSampler that profiles loop with sampling and evaluate the accuracy of it while we attempt to emphasize the accuracy of loop profiling.

The HPCToolkit presented by Tallent et al. [8] performs binary analysis that recovers its static structure of loop nests using Havlak’s algorithm and maps it into the debug information. While theirs aim at tuning of programs performed by post-mortem, two-pass analysis, the proposed pMarker method is based on one-pass analysis and can obtain dynamic information on-the-fly. Also, HPCToolkit and ours are significantly different in the way how the loop nests are extracted. In [8], they perform static loop analysis and dynamic CCT generation separately, and later they combine these independent results together. This is why they cannot monitor on-the-fly loop nesting structure and loop trip count while ours can.

As discussed in [8], there are structural gaps between the original source code and its corresponding machine code. Since machine code is generated from source code by compilers, compilers basically have ability to map them accurately. However, the debug information provided by compilers is not always true [32]. To solve this problem, they also discussed how to recover its program structure and how to reconstruct a mapping back to its source code [8]. They showed that they can accurately correlate highly optimized binaries with procedures and loops to the original source codes within a procedure call. Therefore, we assume that we can map the binary code structure to that of original source code, and the detail evaluation of the gap is beyond scope of this paper.

MAQAO is a static analysis tool of assembly code produced by compilers coupled with dynamic information such as hardware performance counters [25], [26]. MAQAO can profile metrics such as the number of iterations of the loop body (trip count) and the number of instructions per iteration. However, it does not analyze the dynamic structures of loop nests, and loop trip counts are obtained by monitoring the registers containing the loop counters through value profiling.

Rul et al. present the program representation of procedure calls, loop bodies and general code fragments, and use it for automatic program partitioning [27]. Wang et al. present a loop selection algorithm and its performance estimation [28]. Lau et al. used the call-loop graph to detect the phase change of code execution [29]. However, the authors of these papers do not discuss the details of how they obtained the results on profiling. Also, Beyls et al. visualize loop nest structures in the form of nested loop forests [30], but the scope of loop nests is limited to a single procedure call because it is based on static analysis.

In several papers, the loop-call graph is utilized to represent nested loop structures [7], [29]. Like the difference between call graph and CCT, the loop-call graph cannot depict exact paths of nested loops and procedure calls due to merging of the flow. Therefore, they cannot precisely monitor the number of nested loops. Using L-CCT enables us to monitor loops that are nested across the whole execution, whether particular invocations of loops behave differently in terms of the number of iterations, the data dependence patterns and so on [28]. In [9], we compared the differences between the L-CCT and the loop-call graph (which is referred to as the L-CFG) in terms of the number of nodes in the execution.

K-scope is a Fortran source code analyzer aiming at guiding performance tuning of the K computer, the current Japan’s biggest supercomputer [3]. While K-scope focuses on the multi-level nested loops, the obtained loop structures are source code level. Also, it is limited to the scope of static analysis and accepts only Fortran language.

Chen et al. extract the loop entries and exits by hands and monitor them using special instructions for annotations called sloop and eloop [31]. While their mechanism assumes hardware support, our pMarker method is performed on dynamic BT aiming for the runtime optimization or speculative execution.

7. Conclusions

In this paper, we have presented the pMarker (precise marker) method that dynamically detects precise loop structures with their inter-procedural nests using a dynamic binary translator. To generate precise loop structures, we have extracted reducible and irreducible loops by static analysis when the code is loaded for the first time. Then, we create simple markers for instrumentation and monitor inter-procedural dynamic behaviors of loops during execution.
To track the precise loop stacks for maintaining dynamic loop nests, we have implemented mechanisms that handle indirect jumps and setjmp/longjmp functions. We have also proposed the use of a L-CCT (Loop-Call Context Tree) representation to depict the exact paths of loops and procedure calls.

We have implemented our method in Pin tool set, and evaluated it using SPEC CPU benchmark suite. We have demonstrated that our loop detection mechanism successfully keeps track of the dynamic loop behaviors of all the program executions on the SPEC CPU2006 suite without any manual directive insertions and particular supports by compilers. We have described why the conventional dynamic loop detection methods incorrectly identify loops and showed the imprecise loop nests that appeared in their methods. In addition, we have shown that our pMarker method can detect inter-procedural loop nest structures on-the-fly with 5.0 times slowdown to the native execution on average, and can reduce overheads compared with the conventional loop detection methods. These results indicate that our tool is precise and productive enough to be used in actual programs.

For future work, we need to discuss how many of the loop regions can actually be optimized or parallelized quantitatively under realistic assumptions. Since, unlike the tools based on post-mortem analysis, our tool can obtain dynamic information on-the-fly, the proposed system could be applied to dynamic tuning/optimization and dynamic automatic parallelization. However, a gap still exists between having an L-CCT and performing dynamic optimization/parallelization. We foresee this work becoming an integral part of the puzzle for dynamic optimization and dynamic automatic optimization/parallelization in future.

References


Yukinori Sato received the BS degree, and the MS and Ph.D. degree on Information Sciences from Tohoku University in 2001, 2003, and 2006 respectively. From 2006 to 2007 he was engaged in embedded processor system design at FineArch Inc. and also became a joint research member at Tohoku University. Since 2007, Dr. Sato is with JAIST (Japan Advanced Institute of Science and Technology) as an assistant professor and currently engaged in research and development of HPC system architecture and software at Research Center for Advanced Computing Infrastructure. His research interests include system architectures and compilers for high-performance computing systems and reconfigurable computing.

Yasushi Inoguchi received his B.E. degree from Department of Mechanical Engineering, Tohoku University in 1991, and received MS degree and Ph.D. from Japan Advanced Institute of Science and Technology (JAIST) in 1994 and 1997, respectively. He is currently a Professor of Research Center for Advanced Computing Infrastructure at JAIST. He was a research fellow of the Japan Society for the promotion of Science from 1994 to 1997. He is also a researcher of PRESTO program of Japan Science and Technology Agency from 2002 to 2006. His research interest has been mainly concerned with parallel computer architecture, interconnection networks, GRID architecture, and high performance computing on parallel machines.

SATO et al.: IDENTIFYING PROGRAM LOOP NESTING STRUCTURES DURING EXECUTION OF MACHINE CODE


Tadao Nakamura received the IEEE Computer Society Taylor L. Booth Award in 2004. He has been Advisory Committee Chair of the IEEE COOL Chips conference series fully sponsored by the IEEE Computer Society. He is a Life Fellow of the IEEE.