Extracting Frequent-Instruction Sequences from a Program Trace

SHINPING R. WANG and JUI-SHANG CHIU
Department of Computer Science and Information Engineering, Da-Yeh University
No. 112, Shanjiao Rd., Dacun, Changhua, Taiwan 51591, R.O.C.

ABSTRACT
Classical function-level profiling approaches generate coarse-grained information that is inappropriate for micro-architectural designs. Although modern compilation-based instruction-level profiling approaches provide fine-grained information, they are often platform-dependent. This report presents a simulation-based instruction-level profiling approach that provides fine-grained profiling information. Furthermore, the proposed approach is platform-independent, easy to use and cost-effective. The proposed approach profiles frequent-instruction sequences from the run-time trace of a program. The derived sequence can be applied to facilitate micro-architectural designs. Dijkstra was selected as the target program for demonstration.

Key Words: profiling, microarchitecture, Dijkstra, instruction sequence
II. RELATED WORKS

Suresh et al. summarized and categorized some recently available profiling approaches [1]. Some of these approaches require instrument codes or a prior knowledge about the structure of the profiled program. Examples are ATOM, HALT, OPT, and FLAT. Some of these approaches are platform dependent such as HALT, OPT, ProfileMe, Shade, ALTO, Vtune, and CacheProf. A similar approach to this paper is presented by Villarreal et al. [7]. The approach first analysis the profiled program for loop information and constructs a directed acyclic graph (DAG). It then parses the program’s instruction trace against the DAG and from where statistics about loop information is generate.

All previous mentioned approaches generate statistics about program run-time behavior. It then requires human effort to derive target code segment from the generated statistics of the program for optimization. In contrasting, the approach of this paper extracts frequent instruction sequences which are ready for optimization.

The basic concept of the proposed approach derives from WINEPI by Mannila [5]. WINEPI is a pattern-mining algorithm for discovering frequent event episodes from a time series. WINEPI is adopted because it does not require the user to define event episodes specifically for discovery, but instead only requires abstract descriptions of the pattern of the episodes, such as whether it is mining serial or parallel events, and the episode lengths. Pattern mining approaches make the proposed approach easy to use, such that the user no longer requires a prior knowledge of details of the instruction sequence to be traced. The WINEPI is modified and implemented at the initial phase of this research. However, WINEPI is found to be inefficient to our application, and so is replaced with a new algorithm called Melting.

The proposed Melting approach is divided into 5 steps, as depicted in Figure 1. The well-known Dijkstra algorithm is employed to demonstrate the approach.

III. MELTING

1. Step 1: Cross-compile the Target Program

The proposed approach was demonstrated using Dijkstra from MiBench [3] on an ARM machine. Dijkstra was compiled on an i386 host using an ARM cross-compiler built with gcc-3.4.1, glibc-2.3.3 and binutils-2.15. Moreover, to explore all possible means of optimization, compiler optimization was switched off, and the baseline compilation setting was assumed.

2. Step 2: Generate the Traces

The compiled program was simulated by sim-outorder from the Simplesim-arm suite [2]. The sim-outorder provides a ptrace option, which gathers instructions issued alongside the detailed pipeline status at each machine cycle. However, the
proposed approach reduces data size of the program trace by selectively retaining only issued instructions, and suppressing all other information from the ptrace. The entire program trace comprised 117,957,753 instructions, and had a size of 1,641,086,369 bytes. For easy of reference, the program trace is denoted as S.

3. Step 3: Perform the Coverage Test

A coverage test providing statistics about the number of occurrences of each individual instruction inside of S was performed. The test result was an instruction list, which was indexed in descending order by each instruction’s number of occurrences from S. This result can be achieved on a high-end workstation with sufficient memory using the Unix commands sort and uniq -c. A total of 2,263 unique instructions were obtained in Dijkstra. Table 1 shows the top 15 instructions.

![Fig. 2. Histogram of the coverage.dat](image)

<table>
<thead>
<tr>
<th>index</th>
<th># occurrences</th>
<th>instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>i_1</td>
<td>6348086</td>
<td>ldr r3,[r3,#0]</td>
</tr>
<tr>
<td>i_2</td>
<td>3710080</td>
<td>ldr r3,[r3,#12]</td>
</tr>
<tr>
<td>i_3</td>
<td>3695104</td>
<td>sub r2,r11,#32</td>
</tr>
<tr>
<td>i_4</td>
<td>3586879</td>
<td>str r3,[r2,#0]</td>
</tr>
<tr>
<td>i_5</td>
<td>3399816</td>
<td>ldr r3,[r2,#0]</td>
</tr>
<tr>
<td>i_6</td>
<td>3161704</td>
<td>cmp r2,r3</td>
</tr>
<tr>
<td>i_7</td>
<td>3074970</td>
<td>add r3,r3,r2</td>
</tr>
<tr>
<td>i_8</td>
<td>3044950</td>
<td>mov r2,r3, lsl #2</td>
</tr>
<tr>
<td>i_9</td>
<td>3014237</td>
<td>add r3,r3,#15</td>
</tr>
<tr>
<td>i_10</td>
<td>3014237</td>
<td>mov r3,#9984</td>
</tr>
<tr>
<td>i_11</td>
<td>2524266</td>
<td>cmp r3,#0</td>
</tr>
<tr>
<td>i_12</td>
<td>1933322</td>
<td>ldr r3,[r1,#0]</td>
</tr>
<tr>
<td>i_13</td>
<td>1913007</td>
<td>beq 0xfffffd8</td>
</tr>
<tr>
<td>i_14</td>
<td>1877482</td>
<td>sub r1,r11,#32</td>
</tr>
<tr>
<td>i_15</td>
<td>1847404</td>
<td>b 0xfffffd8</td>
</tr>
</tbody>
</table>

Figure 2 shows the histogram of the list from the coverage test. The numbers of occurrences for those instructions after i_150 were fairly small, and were therefore skipped. A threshold value was selected from the histogram. The choice of threshold value was obvious for the Dijkstra program.

The Dijkstra histogram exhibits an abrupt change around the 46th instruction indexed by i_46, as revealed in Figure 2. Consequently, the threshold value 1,400,000 corresponding to instruction i_46 was adopted in the following melting operation to extract frequent instructions from S. Notably, different threshold value would produce different frequent sequences. Section 4 discusses the choice of threshold value in detail.

4. Step 4: Perform Melting with the Threshold Value

Every instruction from S was matched to the instructions from coverage.dat. A match would indicate that the corresponding instruction from S is frequent, and should be retained in S. If no match is found, then the corresponding instruction is eliminated from S. This simple operation is called melting, and serves two objectives. First, it reduces the
size of S. Second, it breaks down one huge contiguous sequence of instruction S into pieces of much smaller contiguous instruction segments, known as chunks. Chunks contain all frequent instructions, and hence the corresponding frequent instruction sequences. This follows the lemma from WINEPI, that every instruction in a frequent instruction sequence has to be frequent. More significantly, the problem of the searching frequent instruction sequence is downscaled from S to these much smaller chunks, significantly reducing processing time. The melted program trace S was saved to a file named M, which contained 10,699,137 chunks. The smallest chunk was of size 1, and the largest chunk comprised of 3,454 instructions.

5. Step 5: Parse the Melted File M
Frequent instructions collectively appear in chunks, which are of particular interest. Notably, many chunks repeat themselves inside M. To simplify parsing, Unix commands sort and uniq were run on file M to find the chunks. A total of 439 unique chunks were obtained. A closer examination of these chunks reveals that most of the chunks were loops of the same instruction sequence, which WINEPI is inefficient in handling. Consequently, a simple algorithm was developed to parse these chunks where loops arise. The algorithm works on the frequency domain, and finds the positions of occurrence of every individual instruction from the chunks. The precedence order and intervals of occurrences of each individual instruction were derived from this information. Ordered instructions form a sequence if they take place consecutively. Additionally, all the intervals of the sequence share the same interval of occurrences inside the chunk, then the sequence is cyclic. The pseudocode of the algorithm is shown below. In the pseudocode, C denotes the chunk; |C| denotes the size of the chunk, and index j denotes the initial position of an instruction inside the chunk, and is utilized to resolve the instruction's precedence ordering. Index k is utilized to determine an instruction's interval. PS denotes a data structure for storing each instruction and its intervals.

```
for k = j+1 to |C| do begin
  if (i(k) equals i(j));
    INTERVAL = k - j;
  if INTERVAL not in PS.INTERVAL then
    save INTERVAL into PS.INTERVAL;
    COUNT++;
end
end
```

The algorithm was implemented using Perl, and was applied to file M. Table 2 presents the result of a typical chunk from M.

Table 2 presents the outcomes of parsing chunk 349 of M. The chunk comprises nine individual instructions. It begins with instruction {sub r2, r11, #32}, which repeats itself in the intervals of 8 and 3. Specifically, the instruction begins at position 1, and recurs at position 9, which is at a distance of 8 from its previous position 1. It then recurs at position 12, which is a distance of 3 from its previous position of 9, then at position 20, which is a distance of 8 from position 12, and then at position 23, which is a distance of 3 from position 20. Notably, the sum of intervals 8 and 3 gives 11. This indicates the instruction repeats itself twice every 11 instructions. The second instruction is {ldr r3, [r2, #0]}, which also repeats itself at an interval of 11. The third instruction is {ldr r3, [r3, #12]}, which is similar to the first instruction, because it also has two intervals, which are 5 and 6. These two intervals also add up to 11. The fourth instruction is {cmp r3, #0}, which is also at an interval of 11. All the instructions of the chunk have the same interval of 11. Table 2 does not show the two instructions at positions 8 and 9, which are from the third and first instructions of the table respectively. Consequently, the algorithm resolves a sequence comprising of eleven instructions: {sub r2, r11, #32}; {ldr r3, [r2, #0]}; {ldr r3, [r3, #12]}; {cmp r3, #0}; {beq 0x1c}; {sub r1, r11, #32}; {ldr r3,

<table>
<thead>
<tr>
<th>Chunk ID. 349</th>
<th>COUNT</th>
<th>PS.INS</th>
<th>PS.INTERVAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>628</td>
<td>sub r2, r11, #32</td>
<td>8, 3</td>
</tr>
<tr>
<td>2</td>
<td>314</td>
<td>ldr r3, [r2, #0]</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>628</td>
<td>ldr r3, [r3, #12]</td>
<td>5, 6</td>
</tr>
<tr>
<td>4</td>
<td>314</td>
<td>cmp r3, #0</td>
<td>11</td>
</tr>
<tr>
<td>5</td>
<td>314</td>
<td>beq 0x1c</td>
<td>11</td>
</tr>
<tr>
<td>6</td>
<td>314</td>
<td>sub r1, r11, #32</td>
<td>11</td>
</tr>
<tr>
<td>7</td>
<td>314</td>
<td>ldr r3, [r1, #0]</td>
<td>11</td>
</tr>
<tr>
<td>10</td>
<td>314</td>
<td>str r3, [r2, #0]</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>314</td>
<td>b 0x0f0f0f08</td>
<td>11</td>
</tr>
</tbody>
</table>
This lemma states that the number of occurrences of $S$ is bounded above by the sub-sequence with the smallest number of occurrences. In other words, the number of occurrences of $S$ cannot exceed the sub-sequence with the smallest number of occurrences.

Considering the above lemma, the following theorem gives a guide for selecting a suitable threshold value.

**Theorem:** Assume the frequent sequence that under threshold value $T$ is $St$. If the number of occurrences $O(St)$ of $St$ is greater than the threshold value $T$, then it is only possible to find a more frequent sequence $Su$ using a threshold value $U > O(St)$.

**Proof:** Possible threshold value of $U$ is either in the range of (i) $U \leq O(St)$ or (ii) $U > O(St)$. Considering case (i) first, following Lemma 1, any sequences form $U \leq O(St)$ will be bounded by $O(St)$ such that $O(Su) \leq \min \{O(Si) \mid i = 1,2,...,n \text{ and } O(Si) \leq U \leq O(St) \}$. As a result, the number of occurrences of a newly generated sequences $Su$ that having threshold value $U \leq O(St)$ will not exceed the upper-bound $O(St)$. For (ii) that $U > O(St)$, following Lemma 1, $O(Su) \leq \min \{O(Si) \mid i = 1,2,...,m \text{ and } O(Si) > U > O(St) \}$. This says, $O(Su)$ is bounded above by $O(St)$. Since $O(St)$ to all $i$ is greater than $O(St)$, hence it is possible existing an sequence that having greater number of occurrences than $O(St)$ such that $O(Su) > O(St)$.

The theorem states that sequences that occur more often than the most frequent sequence under a certain threshold can only be obtained by selecting a threshold value greater than the largest number of occurrences. Consider the frequent sequence from Table 2 as an example. The sequence is found with a threshold value of 1,400,000, and its number of occurrences is 1,847,302. Following the theorem, a more frequent sequence can be found only with a threshold value greater than 1,847,302. Hence, the melting operation was performed with thresholds of 1,850,000 and 2,000,000. Experimental results indicate that these larger threshold values break the frequent sequence from Section III into shorter fragments, which merge with other scatter sequences into sequences of higher occurrence. However, the increased number of occurrences is at expense of lower length-occurrence product. In particular, the length-occurrence product of these newly derived sequences is much lower than the frequent sequence from Section III, which has a length-occurrence product of 20,320,322 instructions, and contributes 17.23% of the program’s run-time instruction counts. Consequently, these newer frequent sequences are considered to be less significant than the sequence from Section 3 that using

```
[r1, #0]; {ldr r3, [r3, #12]}; {sub r2, r11, #32}; {str r3, [r2, #0]}; {b 0xffffffd8}. Since all instructions of the sequence share the same common interval 11, the sequence repeats itself cyclically 314 times in the chunk. Notably, all the instructions from the above frequent instruction sequence arise in the top 15 entries of Table 1.

Further analysis indicates that 367 out of 439 chunks are loops of the above length 11 sequence. The sequence takes place totally 1,847,302 times, and the length-occurrence product gives a total of 20,320,322 instructions. The total number of instruction counts of $S$ is 117,957,753, revealing that the above frequent sequence contributes 17.23% of the program’s run-time instruction counts.

To demonstrate that the above sequence is a frequent sequence, the object code of Dijkstra was disassembled to trace the frequent sequence back to function enqueue() of Dijkstra. The whole disassembled object code contains 726 instructions, and function enqueue() contains 83 instructions. While enqueue() comprises 83 instructions, the frequent sequence back to function enqueue() of Dijkstra. The whole disassembled object code contains 726 instructions, and function enqueue() contains 83 instructions. While enqueue() comprises 83 instructions, the frequent sequence back to function enqueue() of Dijkstra.

To demonstrate that the above sequence is a frequent sequence, the object code of Dijkstra was disassembled to trace the frequent sequence back to function enqueue() of Dijkstra. The whole disassembled object code contains 726 instructions, and function enqueue() contains 83 instructions. While enqueue() comprises 83 instructions, the frequent sequence back to function enqueue() of Dijkstra.

### IV. THE RELATIONSHIP BETWEEN FREQUENT SEQUENCES AND THRESHOLD VALUE

This section investigates the relationships between threshold values and their corresponding frequent sequences. Different threshold values yield different frequent sequences. The following discussion elucidates this phenomenon, and provides guidance for selecting a suitable threshold value.

**Lemma 1:** If a sequence $S$ consists of sub-sequence $\{S_1, S_2, ..., S_i, ..., S_n\}$, then the number of occurrences $O(S)$ of $S$ is bounded by $O(S) \leq \min \{O(S_i) \mid i = 1,2,...,n\}$.
## V. CONCLUSION

The proposed approach acts as a static simulation-based instruction profiler, and provides fine-grained instruction-level profiling in contrast to the classical function-level profiler. Since this approach is software-based, it is flexible, platform-independent and cost-effective.

Although proposed approach is demonstrated using Dijkstra, it also works on other programs. The approach has also been successfully tested on jpeg from MiBench. Nonetheless, this approach is designed for small-scale applications running on a dedicated system such as a deeply embedded system. A large program could be difficult to profile, because of the size of traces involved. However, the problem can be overcome by using a hierarchical profiling approach. Specifically, a function-level profiling is first performed to determine the most frequent function, and only the most frequent function is traced. The proposed approach can then be adopted to extract the frequent instruction sequence of the function.

The derived frequent sequence can be employed to facilitate designs of customized instruction synthesis, hardware/software partition and instruction cache. The proposed method is also promising for other applications. By modifying the parser from Step 5 of the Melting approach, not only the instruction traces, but others pipeline run-time status such as register utilization, program branch behavior, and memory references, can be profiled. The profiled information can in turn be employed in compiler tuning, branch prediction and data cache design.

### REFERENCES


Received: Mar. 8, 2008 Revised: Apr. 9, 2008 Accepted: May 26, 2008