科學與工程技術期刊　第四卷　第四期　民國九十七年

Journal of Science and Engineering Technology, Vol. 4, No. 4, pp. 1-6 (2008)

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**

自程式追蹤中萃取指令序列的方法

王欣平　邱瑞山

大葉大學資訊工程系

51591 彰化縣大村鄉山腳路 112 號

摘　要

傳統函式層側描方法所產生的資訊因過於簡略而不敷現代計算機微架構設計使用。雖然以編譯所得的指令序列爲基礎的指令層側描方法可產生較傳統側描方法更爲詳細的資訊，但是這些方法多需搭配高價的軟體套件或硬體平台使用。本文提出一個以軟體模擬執行所得的指令序列爲基礎的指令層側描方法，稱爲 Melting。這個方法除了能產生微架構設計所需的詳細資訊外，也不需使用特殊軟硬體，更符合成本效益。我們提出的方法用於萃取出程式的指令動態追蹤中最常出現的指令序列。所得的指令序列資訊可協助快取記憶體及分支預測等計算機微架構設計參考。本文以 Dijkstra 演算法爲應用範例以說明 Melting 的相關演算法及架構。

關鍵詞：側描，微架構，Dijkstra，指令序列

I. INTRODUCTION

Although a helpful approach for program optimization and system performance tuning, classical function-level profiling has certain limitations. In particular, function-level profiling relies on instrument codes inserted in the functions of the profiled program during compilation. The program's global run-time information is gathered via these inserted instrument codes. The gathered run-time information includes, but is not restricted to, the accumulated CPU time, the number of calls made to each function, and the program's functional call-graph. Programmers and system designers tune their design according to the gathered run-time profiling information. The fact that the profiling is performed at the function level does not necessarily reflect the way that the profiled program runs on a machine, especially in a cross-platform embedded system development in which the program is profiled and executed on different machines. Different machine Instruction Set Architectures (ISAs), libraries and compilation settings of the target machine change the execution behavior to the profiling result performed at the host machine. Modern microarchitecture design requires fine, direct profiling approaches. A promising alternative to this problem is instruction-level or machine-level profiling. An ISA is the gateway between software and hardware [1]. Working with the ISA is thus the closest that the programmer gets to operate the machine hardware. Architecture designers are increasingly turning their interest to the instruction-level profiling, leading to the development of some clever designs over recent years [1]. Suresh et al. summarized some recently available profiling approaches [6].

This paper presents a software approach that profiles a program's run-time instruction traces for frequent instruction sequences. The proposed approach can be classified as a static simulation-based instruction profiler [6]. Specifically, the approach utilizes an instruction-set simulator to generate a run-time instruction trace, and then profiles the trace off-line. It is particularly interested in frequent sequences for several reasons. In summary, since it is frequent that means potential gain from optimizing the sequence is high, following Amdahl's law. Additionally, an instruction sequence carries high quantities of information, making it useful for design and optimization. Section II overviews some related works. Section III describes the proposed approach in detail, and Section V is the conclusion.

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

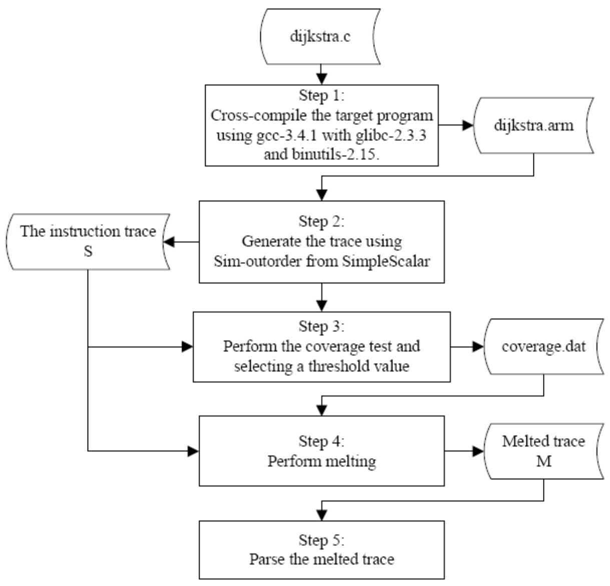
dijkstra.c
Step 1:
Cross-compile the target program using gcc-3.4.1 with glibc-2.3.3 and binutils-2.15.
dijkstra.arm
Step 2:
Generate the trace using Sim-outorder from SimpleScalar
The instruction trace S
Step 3:
Perform the coverage test and selecting a threshold value
coverage.dat
Step 4:
Perform melting
Melted trace M
Step 5:
Parse the melted trace

Fig. 1. Flow chart of the proposed melting approach

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.

Figure 2 shows the histogram of the list from the coverage test. The numbers of occurrences for those instructions after i150 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 i46, as revealed in Figure 2. Consequently, the threshold value 1,400,000 corresponding to instruction i46 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.

Tabel 1. The table shows the top 15 instructions of the list that generated by the coverage test. Each instruction is indexed by *in* where subscript *n* stands for the order of the instruction inside the list

index	# occurrences	instruction
i_1	6348086	ldr r3,[r3,#0]
i_2	3710080	ldr r3,[r3,#12]
i_3	3695104	sub r2,r11,#32
i_4	3586879	str r3,[r2,#0]
i_5	3399816	ldr r3,[r2,#0]
i_6	3161704	cmp r2,r3
i_7	3074970	add r3,r3,r2
i_8	3044950	mov r2,r3, lsl #2
i_9	3014237	add r3,r3,#15
i_{10}	3014237	mov r3,#9984
i_{11}	2524266	cmp r3,#0
i_{12}	1933322	ldr r3,[r1,#0]
i_{13}	1913007	beq 0x1c
i_{14}	1877482	sub r1,r11,#32
i_{15}	1847404	b 0xffffffd8

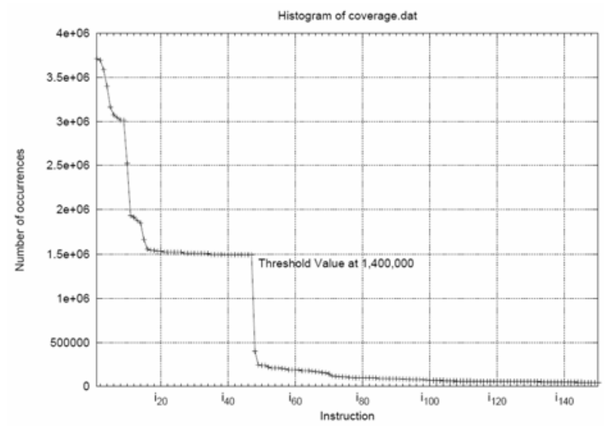

Fig. 2. Histogram of the coverage.dat

In this implementation, all instructions after i46 of the list from the coverage test were removed. The remaining instructions from the list were saved to a file called coverage.dat, and passed to the melting operation as hash indices for rapid processing of the program trace S.

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, if all the instructions 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.

```
INPUT:      M file
OUTPUTS:
            PS.INS, the instruction
            COUNT, number of occurrences of PS.INS in C
            PS.INTERVAL, interval of PS.INS

for all chunk C in M do begin
for j = 1 to |C| do begin
if instruction i(j) not in PS.INS
save i(j) into PS.INS;
            initiate COUNT to 1;
        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
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 2. Outcomes of parsing chunk 349 of M

Chunk ID. 349			
j	*COUNT*	*PS.INS*	*PS.INTERVAL*
1	628	sub r2, r11, #32,	8, 3
2	314	ldr r3, [r2, #0],	11
3	628	ldr r3, [r3, #12],	5, 6
4	314	cmp r3, #0,	11
5	314	beq 0x1c,	11
6	314	sub r1, r11, #32,	11
7	314	ldr r3, [r1, #0],	11
10	314	str r3, [r2, #0],	11
11	314	b 0xffffffd8,	11

[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 above only takes 11 of them, and thus constitutes 13.25% of enqueue(). Furthermore, the instructions of the frequent sequence above only take 1.51% of the entire Dijkstra program. This analysis reveals that a total 17.23% of run-time instruction counts come from only 1.51% of the source, indicting an extreme instance of the "90-10 rule".

Function-level profiling using gprof also reveals that enqueue() is the most frequently called function of Dijkstra. It is called 14975 times, and spends 20% of the overall execution time.

The time complexity of melting operation from step 4 of this implementation was O(n|c|), where n denotes the size of program trace, and |c| denotes the size of coverage.dat. Unlike in WINEPI, the melting operation in the proposed approach is performed in a single pass, while the operation of discovering the candidate sequence against the program trace in the WINEPEI requires many passes to complete.

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 {S1, S2, .., Si,.., Sn}, then the number of occurrences O(S) of S is bounded by O(S) ≦ min {O(Si) | i = 1,2,..,n}.

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 ≦O(St) or (ii) U > O(St). Considering case (i) first, following Lemma 1, any sequences form U ≦ O(St) will be bounded by O(St) such that O(Su) ≦ min {O(Si) | i = 1,2,..,n and O(Si) ≦ U ≦ O(St) }. As a result, the number of occurrences of a newly generated sequences Su that having threshold value U ≦ O(St) will not exceed the upper-bound O(St). For (ii) that U > O(St), following Lemma 1, O(Su) ≦ min {O(Si) | i = 1,2,..,m and O(Si) > U > O(St)}. This says, O(Su) is bounded above by O(Si). Since O(Si) 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-occurence 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 threshold value 1,400,000.

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

1. Anderson, J. M., L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger and W. E. Weihl (1997) Continuous profiling: Where have all the cycles gone? *ACM Transactions on Computer Systems* (TOCS), 15(4), 357-390.
2. Burger, D. and T. M. Austin (1997) The SimpleScalar tool set, version 2.0. *Computer Architecture News*, 25, 13-25.
3. Guthaus, M., J. Ringenberg, D. Ernst, T. Austin, T. Mudge and R. Brown (2001) MiBench: A free, commercially representative embedded benchmark suite. Proceedings of the 4th Annual IEEE International Workshop on Workload Characterization (WWC-4’01), Austin, TX.
4. Hennessy, J. and M. Heinrich (1996) Hardware/software codesign of processors: Concepts and examples. In: *NATO/ASI Series E: Applied Sciences*, 310, 29-44. G. De Micheli and M. S. Dordecht Eds. Kluwer Academic Publishers, Dordrecht, Netherlands.
5. Mannila, H., H. Toivonen and I. Verkamo (1995) Discovery of frequent episodes in event sequences. Proceedings of the 1st International Conference on Knowledge Discovery and Data Mining, Montreal, Canada.
6. Suresh, D., W. Najjar, F. Vahid, J. Villarreal and G. Stitt (2003) Profiling tools for hardware/software partitioning of embedded applications. Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems, San Diego, CA.
7. Villarreal, J., R. Lysecky, S. Cotterell and F. Vahid (2001) A study on the loop behavior of embedded programs: Technical Report UCR-CSE-01-03, Department of Computer Science and Engineering, University of California, Riverside.

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