Analysis of Control Flow Patterns in the Execution of SPEC CPU2000 Benchmark Programs

P. J.Joseph T. Matthew Jacob
Department of Computer Science & Automation
Indian Institute of Science.
Bangalore, 560012 INDIA
080-293-2368
peejay,mjt@csa.iisc.ernet.in

Abstract— Trace cache, an important building block in modem wide-issue processors, buffers and reuses dynamic instruction traces. The selection of relevant traces to he buffered is a critical factor in trace cache performance. The relevance of a trace, determined by its repetition count, is closely tied to the control flow behaviour of programs. Hence, we analyse the control flow patterns in the SPEC CPU2000 benchmarks,

We detect the loops in the CPU2000 integer henchmarks and study the loop path properties. The loop paths show wide variation in sizes; sizes ranging from 8 to 100,000 instructions are observed for significant loop paths. In 6 of the 12 benchmarks, loop paths fit within typical L1 cache sizes.

We use the SEQUITUR algorithm to generate reasonably small sets of control flow paths that cover 99% of instruction execution in the benchmarks. These traces cover more than 95% of program execution with different inputs.

1. INTRODUCTION

Modern high-performance processors capable of executing multiple instructiuns per cycle require instruction fetch units with high throughput. Fetching a large number of instructions in a single cycle is complicated by the presence of branches at, on the average, every fifth instruction [4]. Apart from having to predict multiple branch targets further ahead in the instruction stream, the fetch unit must assemble multiple blocks from multiple lines in the instruction cache. The trace cache [11], addresses the multiple block assembly problem by storing block sequences (traces) in execution order within an instruction memory component called trace cache. The design of trace caches has been addressed in several papers [13], [12] and also implemented in modern processors [14].

The selection of the traces to be stored is a critical factor in trace cache performance. The original trace cache design [11] initiates the construction of a trace at every dynamic basic block, and temiinates it after a fixed number of blocks. Later research has tried different trace selection strategies. Trace termination has been based on multiple criteria, such as a limit on the length of a trace, or occurrence of an indirect branch. Trace construction is initiated only at the termination of a trace. Even with such a trace selection strategy, it has been found that 40% of constructed traces are not used again before replacement [12]. The storage of such irrelevant traces decreases the fetch bandwidth provided by the trace cache; irrelevant traces often replace relevant ones.

In this paper, we address the trace selection problem by analyzing control flow patterns in execution of the SPEC CPU2000 benchmark programs [5]. Properties of selected traces like the frequency of repetitive use are closely tied to the behaviour of loops within the program. Repetition of instruction sequences is controlled by loops. Hence, we study the properties of loop execution in the CPU2000 integer benchmarks. We detect loops dynamically and measure the size, and repetition of loop paths during program execution. For 6 of the 12 benchmarks, the loop paths that cover 95% of instruction execution are found to fit within typical L1 cache sizes.

An ideal trace selection strategy would select basic block sequences with highest repetition counts. This set of sequences would be as concise as possible, while covering an adequate fraction of program execution time. We use the SEQUITUR [10] algorithm to generate sets of hot sequences from instruction execution traces of CPU2000 integer programs. Our results demonstrate the feasibility of generating reasonably small sets of traces covering a high fraction of the instructions executed.

The next section presents our work on characterizing loop execution. Section 3 details the mechanism used to extract hot control flow patterns from program execution, and properties of these patterns. Related work is described in section 4.

2. LOOPS

Repetition of control flow in ordinary programs is controlled by loops [7]. Any loop iteration starts at the heginning of a loop body, possibly executes multiple branches and subroutines, and terminates at a backward branch to the heginning of the loop body. Programs dominated by long loop iterations with several branches, especially branches with irregular outcomes, generate traces with longer distances of repetition. We investigate properties of loop paths in the CPU2000 integer

0-7803-7651-X/03/$17.00 ©2003 IEEE

benchmarks. The distribution of instruction execution in the unique loop paths, and the size distribution of these paths are measured.

Loop Detection Algorithm

To study the properties of dynamic loop execution, we require an algorithm that identifies loop entry, exit, and iterations within an instruction execution string. We use a variation of the hardware dynamic loop detection algorithm proposed in [15] for this purpose. The specific details of the implementation is given in [6].

Experimental Framework

We generated traces for the CPU2000 integer benchmarks compiled with default options (-03) on an IBM PowerPC. The training inputs are used for all experiments unless otherwise indicated. The difficultyin processing the huge instruction execution traces forced us to use sampling techniques. The instruction execution was traced at regular intervals such that the basic block distribution of the sampled trace matched closely with the corresponding distribution in the complete execution [6].

Results

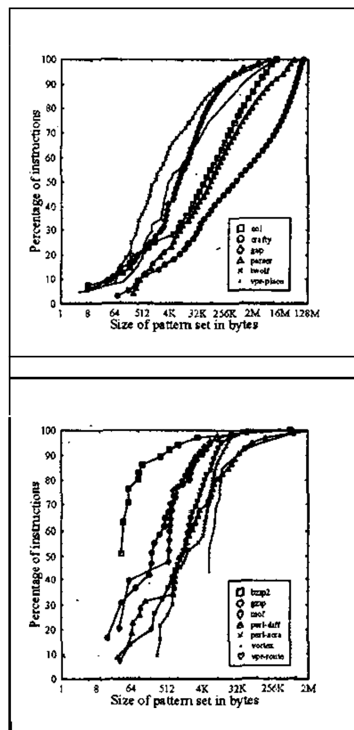

Figure 2. Storage Requirement of Loop Paths.

Figure 1 shows the size distribution of loop paths. The three graphs in the figure plot the cumulative distribution of loop path sizes(in instructions) in the overall instruction execution. The benchmarks have been partitioned across the 3 graphs based on similar behaviour. The classification is based on the size of the hot loop paths covering above 95% instruction execution. In bzip2, gzip, mcf and the routing portion of the vpr benchmark most significant loop paths are less than 300 instructions long. Relatively larger loop paths (upto 4000 instructions in length) contribute to cover 95% instruction execution in cc1, crafiy, gap, parser, and the placement portion of the vpr benchmark. vortex has significant loop paths which are two orders of magnitude larger than in other benchmarks.

Figure 2 shows the sizes of the instruction sequences representing the most hot loop paths in the benchmarks. For the benchmarks plotted in the lower graph, the loop paths covering more than 95% of instruction execution fit within 32 KB of storage, a typical L1 cache size. This observation suggests the selective use of loop based trace selection. The benchmarks plotted on the upper graph have larger instruction footprints; significant paths require more than 256 KB of storage. But, there are potentially large instruction sequences repeating in multiple loop paths of the same static loop. Trace selection strategies which identify and store these sequences separately can reduce the trace storage requirement. We pursue this idea in the next section.

3. Control Flow Patterns

This section presents a mechanism to generate a concise set of basic block sequences which span a majority of program execution. Traditional trace caches use ad hoc mechanisms to select traces. This can create traces with hot as well as rarely executed program regions embedded within them. In addition, the number of traces can he large for large trace sizes, leading to bottlenecks in storing and indexing them. As a step towards a better trace selection strategy, we employ an algorithm to extract repeating patterns in the basic block sequence of program execution. The algorithm is presented in the following subsection. The second subsection studies properties ofthe extractedpatterns. The final subsection shows that this set of hot patterns remains invariant across multiple program inputs for 11 ofthe 12 SPEC2000 integer benchmarks.

Algorithm

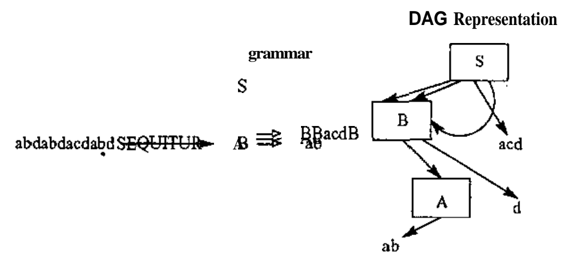

Figure 3. Example of SEQUITUR Grammar Construction.

We use the hierarchical compression algorithm SEQUITUR [10] to identify repetitive structure in a basic block execu-

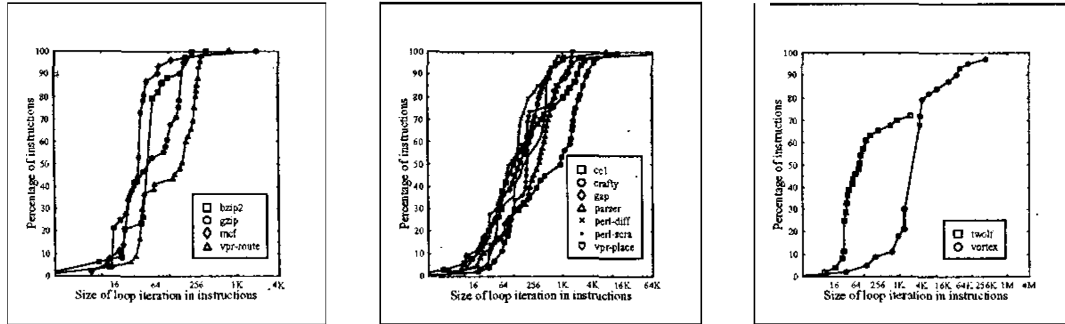

Figure **1.** Distribution of Loop Path Length in Instructions.

tion sequence. We then employ post-processing algorithms to generate hot basic block sequences which span a majority (above 99%) of instruction execution. Nevill-Manning and Witten's SEQUITUR algorithm infers a hierarchical structure from a sequence of discrete symbols. Figure **3** shows the sequence grammar generated by SEQUITUR for the reference sequence ***abdacdubdahd*** and the directed acyclic graph (DAG) representation of the grammar. In our implementation these symbols are the basic block addresses. Generation of hot basic block sequences requires processing of the **SE-**QUITUR generated DAG to produce the frequently repeating symbol sequences. Algorithms have been developed in [8] and [2] to accomplish this purpose. We implement both **of** these algorithms, as well as variations in the context of the basic block execution stream. All the algorithms have a minimum sequence length, maximum sequence length and a hotness criteria as the three parameters.

The results reported in this section are based on control flow patterns generated using Larus' algorithm [8]. We observe from the results a maximum instruction span of 70-90% of instruction execution in all the benchmarks when the algorithm is employed. We employ additional techniques to produce a set ofpatterns which span a large fraction (above 99%) of program execution. A set of patterns **is** generated using Larus' algorithm. Instances of these patterns in the complete basic block sequence are replaced by single symbols and a new sequence obtained. The algorithm **is** re-executed on the new sequence to obtain an additional set of patterns. This procedure is repeated twice to obtain three sets of patterns. It **is** observed **that** the three sets of patterns in combination span above 99% of instruction execution for all the benchmarks.

Properties of Extracted Patterns

We generate hot basic block sequences of length **4** for all the CPU2000 integer benchmarks. Larus' algorithm is employed to extract basic block patterns with a minimum and maximum length of **4.** Three sets of patterns are generated, as described in the previous section, for all the benchmarks

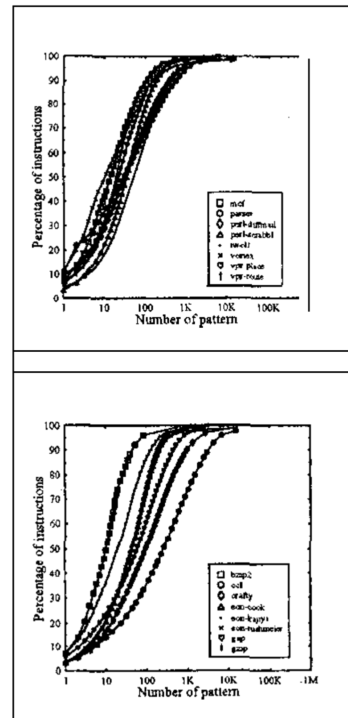

Figure **4.** Distribution of SEQUITUR Patterns of Length **4.**

Figure **4** plots the cumulative distribution of the SEQUITUR generated patterns in the overall instruction execution. The X-axis plots the number of patterns in the set, sorted in decreasing order of hotness. The number of significant patterns (covering more than 95% of instruction execution) is **less** than 1000 for all benchmarks except crafty and cc1. In Figure 5, the storage required for these patterns is studied. The storage

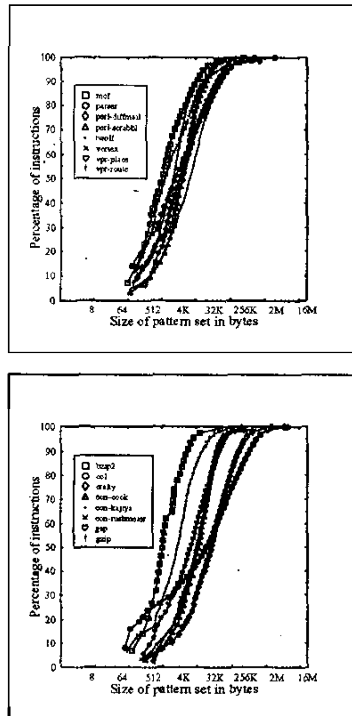

Figure 5. Storage Distribution of SEQUITUR Patterns of Length *4.*

for a pattern incorporates the storage for all the instructions directly embedded in the sequence. If a symbol in the pattern corresponds to another pattern in the pattern set, a storage requirement of **4** bytes is assumed. The X-axis of the two graphs is the cumulative storage of the patterns sorted in decreasing order of hotness. The size of the significant pattern set is less than 256 Kilobytes for all benchmarks except crafty.

If these patterns remain invariant across multiple data inputs, they can be generated using information from a profiling execution, stored with the executable, and used to drive instruction fetch during subsequent program executions. The issue of invariance across multiple data inputs is addressed in the next section.

Pattern Set Variation Across Data Inputs

We studied the instruction coverage of the SEQUITUR generated patterns across different data inputs. The graphs in Figure *6* show the instruction execution span of the patterns (length **4)** in the training input against the span of the same pattern sets in portions of the reference execution with different inputs. The pattems are sorted in decreasing order of hotness **in** the execution with training input. The variations of the plot from linearity indicate that the hotness of the patterns differ in the reference execution. But, the Same set of patterns cover a majority of execution. This is made more explicit in the zoomed version of the graph on the right side. **A** more complete set of results are presented in [6]. The observed property holds true for 11 of the 12 benchmarks.

4. RELATED WORK

There has been extensive research on trace caches and it **has** been implemented in Intel Pentium 4[14]. In [12], Rosner et al find that ad hoc trace selection strategies create a large number of traces, 70% of which are used less than **4** times before replacement. They propose a filtering mechanism to keep the relevant traces in the cache.

Kobayashi **[7]** characterized the dynamic behaviour of loops. The mechanism does not detect all loop paths. Tubella and Gonzalez [15] proposed a hardware loop detection algorithm for the purpose of creating threads in a multithreaded architecture.

Techniques for extracting hot control flow paths dwing program execution has been studied in the context of dynamic optimizers [9], [3]. These proposals attempt to create long basic block sequences, with a high likelihood of repetition, and apply dynamic optimizations on them. The generated set of paths do not cover more than 80-90% of instruction execution in most benchmarks.

In [I], Ball and Larus generate a set of acyclic paths encountered during program execution. In half of the benchmarks the acyclic paths in the training execution span less than 90% of the reference execution. Note that acyclic paths based on static program structure can encompass **an** entire loop body or subroutine path. **Small** variations in the actual path traversed in the reference execution can reduce the instruction execution span of the profiled set of acyclic paths. The algo**rithm** employed in this paper identifies frequently repeating portions of loop paths, and hence the profiled set of paths **is** able to provide wider instruction coverage.

5. CONCLUSIONS

Trace cache performance is closely tied to the trace selection strategy and to the behaviour of loops in the executing program. ***An*** ideal trace selection strategy should consider program behaviow and choose a set of traces having the highest repetition count and covering anadequate fraction of program execution. In this paper, we have studied loop behaviour and addressed the trace selection problem.

We have studied the dynamic execution hehaviour of loops in the SPEC CPU2000 integer benchmarks. The size, and instruction coverage of loop paths were measured. These significant loop paths in *6* of the benchmarks fit within typical L1 cache sizes. Loop paths show wide variations in size; sizes ranging from 8 instructions to 100,000 instructions are observed for significant loop paths.

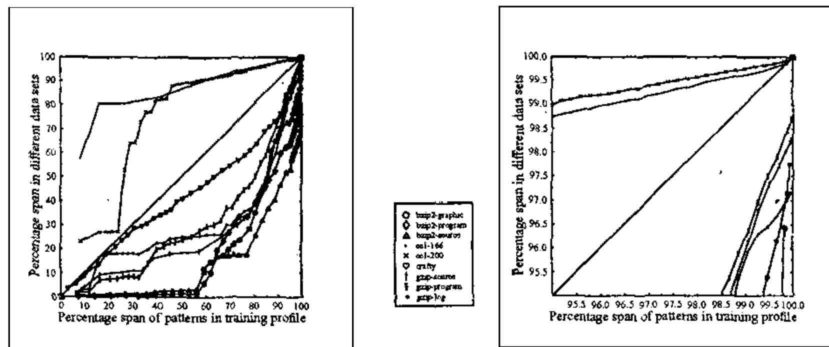

Figure 6. Variation of Pattern Set with Data Inputs.

We generated reasonably small sets of traces that cover 99% of CPU2000 benchmark training execution using the SEQUITUR algorithm. We chose one algorithm to post-process the DAG constructed by SEQUITUR and extracted traces of length 4 basic blocks with required coverage. In 10 of the 12 benchmarks, traces that cover more than 95% of execution require less than 128 KB of storage.

We studied the coverage of the SEQUITUR generated patterns in program execution with different inputs. The set of traces generated using training execution has more than 95% coverage in reference execution of 11 of the 12 benchmarks. This motivates the generation of these traces using basic block trace of a profiling execution. It can be stored in the executable, and used by instruction fetch unit during subsequent executions to achieve improved fetch bandwidth.

REFERENCES

[1] T. Ball and J. R. Larus. Efficient Path Profiling. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, 1996.

[2] T. Chilimbi and M. Hirzel. Dynamic Hot Data Stream Prefetching for General-Purpose Programs. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2002.

[3] B. Fahs, S. Bose, M. Crum, B. Slechta, F. Spadini, T. Tung, S. J. Patel, and S. S. Lumetta. Performance Characterization of a Hardware Mechanism for Dynamic Optimization. In *Proceedings of the 34rth International Symposium on Microarchitecture*, 2001.

[4] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 3rd edition, 2003.

[5] J. L. Henning. SPEC CPU2000: Measuring CPU Performance in the New Millennium. *COMPUTER*, July 2000.

[6] P. J. Joseph and T. Matthew Jacob. Analysis of Control Flow Patterns in the Execution of SPEC CPU2000 Benchmark Programs. *Technical Report*, Department of Computer Science & Automation, Indian Institute of Science, August 2003.

[7] M. Kobayashi. Dynamic Characteristics of Loops. *IEEE Transactions on Computers*, c-33(2), February 1984.

[8] J. R. Larus. Whole Program Paths. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, May 1999.

[9] M. C. Merten, A. R. Trick, R. D. Barnes, E. M. Nystrom, C. N. George, J. C. Gyllenhaal, and W. mei W. Hwu. An Architectural Framework for Runtime Optimization. *IEEE Transactions on Computers*, 50(6):567–589, 2001.

[10] C. G. Nevill-Manning and I. H. Witten. Identifying Hierarchical Structure in Sequences: A Linear-Time Algorithm. *Journal of Artificial Intelligence Research*, September 1997.

[11] A. Peleg and U. Weiser. Dynamic Flow Instruction Cache Memory Organized Around Trace Segments Independent of Virtual Address Line, January 1995. U.S Patent 5,381,533.

[12] R. Rosner, A. Mendelson, and R. Ronen. Filtering Techniques to Improve Trace-Cache Efficiency. In *Proceeding of International Conference on Parallel Architectures and Compilation Techniques*, September 2001.

[13] E. Rotenberg, S. Bennett, and J. E. Smith. Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching. In *Proceedings of 29th International Symposium on Microarchitecture*, December 1996.

[14] J. Stokes. The Pentium 4 and the G4e: an Architectural Comparison, 2001. http://www.arstechnica.com/cpu/01q2/p4andg4e/p4andg4e-1.html.

[15] J. Tubella and A. Gonzalez. Control Speculation in Multithreaded Processors through Dynamic Loop Detection. In *Proc. of the 4rth Int'l Conference on High Performance Computer Architecture*, Feb 1998.