Dynamic Trace-Level Reuse across Multiple Executions

Vikram.V, Varadharajan Ponnappan., A.P.Shanthi
reretune@yahoo.com, vrajanap@yahoo.com, a.p.shanthi@cs.annauniv.edu

Department of Computer Science and Engineering,
College of Engineering, Anna University,
Chennai, India.

ABSTRACT

Trace level reuse is a mechanism of using the results of previous traces, thereby reducing redundant computations and data dependencies. These mechanisms do not retain the results after the termination of the program. In this paper, we propose a reuse mechanism, which stores the results of traces of previous executions of the program, and uses these results for later executions of the program. Apart from reusing trace results within the same execution, our scheme retains the trace reusability information for successive runs of the program. We analyze the performance improvement achieved using this technique, and propose an implementation model. This method leads to increased reuse factor of a trace, compared to normal trace-level reuse mechanisms.

1. INTRODUCTION

Value Reuse mechanisms have been proposed at different granularities, ranging from instruction level to trace level. A trace can be defined as a dynamic flow of instructions executed during an execution of a program. In Trace-level reuse techniques, the trace is reused if it appears again in the program with the same input values. For this purpose, a trace-reuse cache is maintained, which holds the results of previous traces. This enhances the fetch mechanism and eliminates dependencies by skipping the execution of instructions of a reusable trace. By these techniques, the clock cycles required for executing those skipped instructions are saved, thereby reducing overall CPI. Moreover, the data dependences that would have otherwise been present due to these instructions are also eliminated.

The Trace-reuse cache is indexed by the current Program Counter value. The PC value in the cache entry is used to check whether a reusable trace starting with the current Program Counter value exists in the cache. Since there can be more than one trace starting with the same PC value, in order to uniquely identify a particular trace, the inputs that the trace uses are also needed. The inputs can be either register or memory locations. These register and memory locations are fetched and compared with those in the trace cache. If they match, then the processor state is changed to the state after the execution of the trace, skipping the execution of all the instructions in the trace. The processor state immediately after the trace can be identified by the contents of registers, control flags and memory locations after the execution of the trace. These locations are updated to change the state of the processor. The next-PC value in the cache entry gives the PC of the instruction that immediately follows the trace. The Program Counter is made to point to this value, from where normal execution can be continued, as if the trace were executed.

Current reuse techniques have not taken into account, the reusability that might be present across executions of a program. Our approach exploits this reusability, by storing the trace results for later executions. The rest of the paper has been organized as follows. Section 2 discusses related work in this field. Section 3 elucidates our approach and section 4 describes the design issues involved in our idea. Section 5 analyzes the performance issues in this model.

2. RELATED WORK

Avinash Sodani et al [2] proposed a model of dynamic instruction reuse, in which they have analyzed the various schemes for instruction reuse and empirical analysis of instruction repetition in typical programs. They have found a dynamic instruction repetition rate of 70% for integer and graphics

benchmarks. Their analysis shows that a significant amount of redundant calculation is present at instruction level.

Antonio González et al [1] extended the concept of instruction reuse to trace-level reuse. For their model, they used a cache, which they call a Reuse Trace Memory (RTM). The RTM stores trace entries along with trace input and output. They also proposed methods for dynamic trace collection. They analyzed the performance improvement for reuse at trace level compared to reuse at instruction level.

They further proposed that comparison for checking for reusability of a trace can be done either by comparing operand values, or comparing the register/memory names. For the latter approach, a valid bit is assigned for each such register. When the register contents change, the bit is set, which means it is no longer reusable. This method is faster compared to the first method in the sense that the operands need not be fetched for doing comparison. But, the drawback of this method is that even if the register is written with a value which is its actual value itself, it sets the valid bit, meaning that the register contents have changed. Furthermore, they did not take into consideration, the values of control and status flags, which may affect the flow of control of the program and hence the trace.

Mauricio L. Pilla et al [6] proposed a new technique called "Reuse through Speculation on Traces"(RST). They addressed the problem, where the inputs of reusable traces are not ready when the test is done. Their method predicts the trace input values. They also analyzed the limits of RST for modern processors with deep pipelines, as well as the effects of constraining resources on performance and showed a 43% speedup over the conventional non-speculative trace reuse method.

One of the major limitations of these trace reuse techniques is that, they consider only reusability in the current execution of the program. But a greater extent of reusability can be achieved, if the results of traces in previous executions are also taken into consideration. For instance, programs which have very little or no loops or subroutines, will have comparatively lesser reusability within the program. When the trace-reuse techniques are applied for such programs, the improvement in performance may not be good. Our approach takes these factors into consideration, and gives a better performance even in the above circumstances.

3. PROPOSED REUSE MODEL

In current trace-reuse techniques, the trace-reuse information is not retained after the termination of the program. In our method, we store this reuse information in what we call the Program Trace Reuse Table (PTRT), which is stored along with the program code in the disk. When the application is loaded into memory for execution, the PTRT is also loaded into the Trace-Reuse Cache (TRC).

In our technique, even programs which have no reusable traces can be made reusable by extending the concept of reusability across executions of the program. In general, the percentage of trace-level reuse across executions of programs of any kind will be more compared to that of within a single execution of a program.

The Structure of Program Trace Reuse Table

The PTRT is the table, where information about the traces of previous executions is stored. Apart from PC value and input and output values, the table stores the values of control and status flags at the start of the trace. These values are also used to identify a trace, because the branch conditions within the trace may be affected by these flags. Figure 1 shows how a PTRT entry will look like.

Figure1 - A PTRT entry

Trace start PC value (TSPC)	**Input register names and values**	**Input memory locations and contents**	**Output register names and values**	**Output memory locations and contents**	**Initial values of Control and status flags**	**Values of Control and flags after trace completion**	**Next PC value (NPC)**	**Hit Count**	**Clock cycles gained**

Trace identification and Dynamic PTRT updation

At any point during the execution of the program, we remember the first possible instruction after the previous trace, that could be the start of a new trace, called Trace Start PC (TSPC). We keep track of all the live registers and memory locations that we have seen so far in the current trace. Live registers or memory locations are those to which have been accessed during the current trace. The input registers/memories for a trace are that live registers/memory from which we read before any writes to them. Similarly, the output registers/memories for a trace are those to which values are written during the current trace.

Once we encounter an instruction after which the trace cannot continue, we mark it as the end of current trace. Then we create a new entry in the PTRT with the remembered start PC value of the trace as the index. This entry consists of input and output register/memory values, input flag values and PC value of current instruction, which is the instruction that follows this trace. This PC value is referred to as Next PC value (NPC). Since the flow of control of traces can be affected by branch instructions which may depend on status and control flags, these are also included in the PTRT entry. Instructions like software interrupts mark the end of the trace, as these instructions change the flow of the program dynamically. Therefore, the instructions before such instructions are taken to be the end of current trace and updation to PTRT is done as described above.

Dynamic Trace Reuse using PTRT

At each possible trace entry point, the current PC value is used to index into PTRT. There may be more than one entry for a given PC value, where each such entry uniquely identifies the trace by virtue of its input registers and memory values. If any entry is found corresponding to this PC value, then the input register and memory values are fetched and are compared with those in the PTRT entry. If they match, it means this trace has already been executed in the past. The output of the trace, which comprises of register, memory and flag values is written from the PTRT entry into corresponding output registers, memory locations and flags. Then the Program Counter is updated with NPC.

When the program is terminated, the PTRT is stored back to the disk along with the file. This enables the reuse of trace results when the program is executed in the future.

4. DESIGN ISSUES

4.1 PTRT Storage Issues

One of the major design issues in our technique is the storing of PTRT in the disk. There are many ways in which this can be done. One of the methods is to store all the PTRTs in a separate meta-file. The locations of the programs to which the PTRTs belong to are also stored along with them. The major problem of this method is that if the program is moved to a different location, then the metafile needs to be updated. Moreover, the PTRT of the program has to be loaded from the meta-file leading to higher disk access time, as it may be stored in a different disk block from the program.

The second alternative is to store the PTRT along with the program code itself. This makes loading the PTRT easier and also takes care of relocation problem, since the PTRT is stored as a part of the program. We have used this method for our implementation.

4.2 PTRT entry Replacement techniques

There are two situations where replacement is needed.

4.2.1 Global replacement

The situation where a PTRT entry has to be created and the TRC is already full. In this case, the following factors are considered for deciding the victim.

a. The overall frequency of usage of trace entries corresponding to each TSPC - The sum of hit counts of all trace entries of a given TSPC value is used for this purpose

b. Speed up factor - The average number of clock cycles that can be saved by using all the trace entries corresponding to each TSPC entry, which is stored in the PTRT entry.

Taking these two issues into consideration, a relative priority is assigned to each of these issues and used in devising the replacement algorithm.

4.2.2 Replacement within a group of same TSPC entries

Here we consider the case, where a new trace entry has to be created, and there is no room for an extra entry with that PC value. The issues to be addressed here are slightly different than that of Global Replacement. They are

a. The frequency of usage of each and every trace entry within a PC entry.
b. Speed up achieved by using the particular trace entry.

4.3 Loading PTRT into the cache.

Since the PTRT is stored along with the program code in secondary storage, a provision must be provided for loading it into Trace Reuse Cache (TRC). This requires support from Operating System, specifically the Loader. While loading the program, the loader loads the PTRT into TRC(cache).

5. PERFORMANCE ANALYSIS

Since the conventional trace-reuse mechanism is a subset of our method, the performance improvement gained from them would be the least performance gain in our method, under similar circumstances. The trace-reuse techniques were able to give an average speedup factor from 1.43 to 3.03, for 1-cycle reuse latency. Our method gave good results for processes that are more CPU bound.

One of the bottlenecks that may arise in our method is the retrieval time for the PTRT from the disk. However, if PTRT is stored along with the program code, then it can be loaded simultaneously along with program code.

6. CONCLUSION

In this paper, we have proposed a trace-reuse mechanism that retains trace results of previous executions of a program to enhance reusability. This method focuses on the reusability existing among multiple executions of a program, and an implementation model has also been given, that can be incorporated in commercial processors, with some support from the operating system.

REFERENCES

[1] Antonio González, Jordi Tubella and Carlos Molina ,“Trace-Level Reuse”- In Proceedings of the 1999 International Conference on Parallel Processing.

[2] Daniel Citron and Dror G.Feitelson , “Revisiting Instruction Level Reuse” - In Proceedings of the Workshop on Duplicating, Deconstructing, and Debunking (WDDD), May 2002.

[3] Avinash Sodani and Gurindar S. Sohi ,“Dynamic instruction reuse” – In Proceedings of the 24th annual international symposium on Computer architecture, 1997.

[4] Saisuresh Krishnakumaran and Sai Arunachalam ,“Towards Economic Trace Caches: A Profile Based Approach” – Poster session of the 10th International Conference on High Performance Computing 2003, Hyderabad, India.

[5] D.C. Burger and T.M.Austin. - The SimpleScalar Tool Set, Version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin-Madison, 1997.

[6] Mauricio L. Pilla, Amarildo T. da Costa, Felipe M. G. Franca, Bruce R. Childers, Mary Lou Soffa
“The Limits of Speculative Trace Reuse on Deeply Pipelined Processors” - 15th Symposium on Computer Architecture and High Performance Computing ,2003.