Why Not Data Trace Cache

Tomer Y. Morad
Department of Electrical Engineering
Technion, Haifa, Israel 32000
tomerm@tx.technion.ac.il

Uri C. Weiser
Intel Corporation, Haifa, Israel
uri.weiser@intel.com

Avinoam Kolodny
Department of Electrical Engineering
Technion, Haifa, Israel 32000
kolodny@ee.technion.ac.il

Abstract—**Multiple port instruction and data caches are required in order to achieve high performance on wide-issue superscalar microprocessors. However, the area and speed impact of implementing a full blown multiport cache is substantial. The highly predictable nature of the instruction stream has enabled trace caches to effectively fetch instructions from more than one cache block in each cycle. Our analysis shows that the data accesses stream also consists of recurring traces, hinting that a Data Trace Cache that exploits these recurring accesses is feasible. In this paper we show our attempt to design a data trace cache, and explain why its results are inferior to previous simple methods that aim at creating virtual multiport data caches.**

Index Terms—**Cache memories, Data Trace Cache, Memory architecture.**

I. Introduction

MODERN microprocessors achieve high Instruction Per Cycle (IPC) rates by utilizing multiple execution engines in parallel [1]. In order to supply enough instructions for execution in parallel, concurrent access to different cache blocks is required. A multiport cache could be used, but it is an expensive solution in terms of power and area. Instead, instruction trace caches pack instructions from different cache blocks in their dynamic execution order into a single trace cache line [6]. Therefore, a single port instruction trace cache achieves similar performance while avoiding the cost of a multiport instruction cache.

A similar need exists for data supply. Multiple data load instructions should be executed in parallel, since load instructions represent close to a quarter of the dynamic instructions [2]. Fig. 1 shows the IPC improvement as a function of the number of cache memory ports of a very wide machine that can issue and commit up to 32 instructions per cycle. These results for the SPEC2000 GCC benchmark confirm that lack of sufficient cache ports indeed limits microprocessor performance.

In order to avoid the cost of a true multiported data cache [3], we pursue ways to implement a virtual multiport cache. As in instruction trace cache, we could conceive a data trace cache which packs together data from diverse addresses that are required for concurrent access during program execution.

Our analysis of benchmark programs reveals that there are recurring data traces in the data access stream. In the remainder of this paper we explain how our analysis was conducted, and present an architecture for a data trace cache. We evaluate and compare the data trace cache with simpler methods that rely on data access properties of typical computer programs. The evaluation reveals that the simpler methods achieve better performance than the data trace cache.

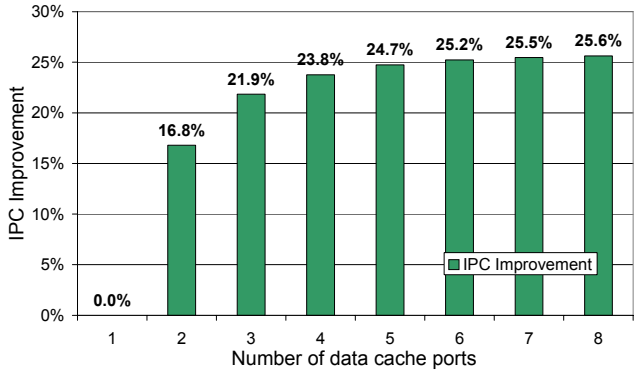

Fig. 1. IPC improvement for SPEC GCC on a 32-wide issue and commit machine as a function of the number of data cache ports. A single port cache is used as a baseline.

Attempts to exploit data traces have been shown in [4] and [5]. Our concept of the data trace cache aims to be a general-purpose cache that does not require recompilation, and is based on similar concepts as the instruction trace cache [6].

Various methods that aim to build a virtual multiport data cache have been researched. Load All Wide (LAW) [7] is a simple method that exploits the spatial locality of the concurrent data accesses, which are usually targeted to the same cache block. Line Buffer (LB) [7] is a mechanism that exploits the temporal locality of concurrent data accesses by caching the last accessed blocks into a small multiport cache.

II. Analysis

In order to characterize the multiple data access behavior of computer programs, a data access trace was searched for recurring access patterns. We define a *burst* as all the loads that were concurrently issued on a specific cycle. For example, consider data access trace in Fig. 2, taken from the SPEC GCC benchmark simulated on a 32-wide issue and commit machine. In this example, we recognize two different bursts, each recurring twice. We define the *cover* of a set of bursts as the number of dynamic loads that reside in all the recurrences of the set of bursts, divided by the total number of dynamic loads. In our example, the cover is 10/12. If such recurring bursts can be found in a data access stream, then storing them in a data trace cache will increase the effective number of memory ports.

Cycle	Port 1	Port 2	Port 3	Port 4
10100	0x7FFF7698	0x7FFF769C	0x7FFF76A0	0x7FFF76A4
10101	0x101A0240			
10102	0x7FFF7698	0x7FFF769C	0x7FFF76A0	0x7FFF76A4
10103	0x101A0242	0x101E2990		
10104	0x101A0240			

Fig. 2. An example data access trace with recurring bursts taken from the GCC benchmark. The values shown are the addresses fetched by an ideal 4-port machine in each cycle.

Best burst coverage analysis was conducted on the data access stream of a number of programs, which were simulated on a 32-wide machine with four memory ports. The results of are shown in Fig. 3, where the best 8192 bursts containing two to four addresses were chosen. The average size of the best 1000 bursts was measured as 3.2 addresses.

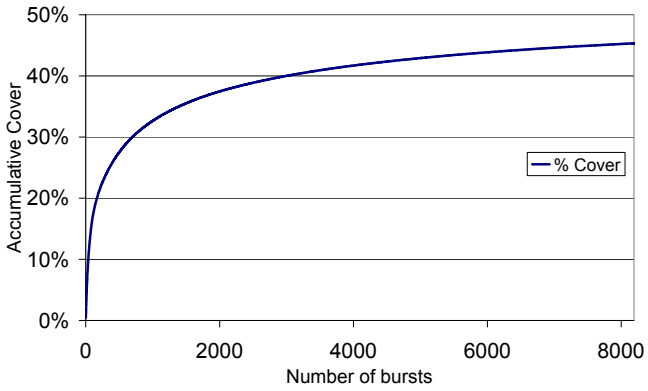

Fig. 3. Percentage of the data access stream covered by the most frequently recurring bursts, for 200 million instructions of the GCC benchmark simulated on a 32-wide machine.

The results clearly show that storing a limited number of bursts can cover a large part of the data access trace. For example, storing 1000 of the best bursts covers a third of the program's data access trace.

III. Data Trace Cache

We consider a new memory system that includes a data trace cache (DTC), a level-1 data cache and a cache controller. The new memory system is shown in Fig. 4.

The cache controller in Fig. 4 has four virtual read ports and a write port. If the CPU requires only one read port at a given cycle, the cache controller requests the data directly from the level-1 data cache, which has only a single read port. In case the CPU requires data from more than one address, the cache controller requests all the data from the entire set of addresses from the DTC. Although the DTC has only one read port, each DTC block contains data from more than one cache block. In parallel to the DTC access, data from a single address is fetched from the level-1 cache. In case of a hit in the DTC, the entire burst of data is returned. In case there was a miss in the DTC, only a single access is serviced by the level-1 data cache.

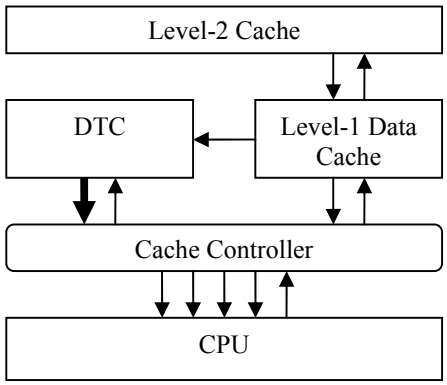

Fig. 4. A memory system with four virtual read ports available to the processor. On a DTC hit, the cache controller can supply up to four different data elements from one DTC block. On a miss, the cache controller supplies only one data element from the level-1 data cache.

DTC blocks contain data from multiple level-1 cache blocks. We have chosen to use the XOR value of the addresses in the burst in order to locate the burst in the data trace cache. The XOR value is divided into block bits, set bits and tag bits, as in conventional caches. Functions other than XOR may also be used.

A Data Trace Cache can be organized as a table, illustrated in Fig. 5. Besides the tag value, each DTC cache block contains n entries of addresses, values and valid bits. Since there are no restrictions on the origins of the data that reside in a data trace cache block, each data element must be accompanied by its address in memory.

Data trace cache blocks are gathered dynamically as the program executes. When there is a data trace cache miss, a DTC block is allocated. The DTC block is filled by snooping the level-1 data cache for its required data. When the required data arrives, the DTC sets its valid bit.

Tag	$Addr_1$	$Data_1$	V_1	…	$Addr_n$	$Data_n$	V_n

Fig. 5. A DTC capable of storing bursts with up to n entries per line. Each entry can originate from a different cache level-1 block.

Store instructions present a major difficulty to the data trace cache. The first problem is locating specific addresses inside the DTC. Since the index function is the XOR value of the addresses, we must keep a lookup table to be able to locate single addresses for store operations. Once locating all the trace cache lines that contain the value that needs to be updated, all these lines must be updated. Updating several blocks require many lookups, resulting in long store latencies.

IV. Simulations and Results

All experiments in this paper were conducted with the SimpleScalar toolset [9] with our modifications. The benchmarks used were the SPEC2000 benchmarks with the

internal instruction set of the tool. Only a subset of the SPEC2000 was used due to technical issues. The benchmarks were simulated on 32-wide issue and commit machines. The level-1 data and instruction caches were 64kb, 32 byte blocks, 4-way associative, 1-cycle latency; the level-2 was a united cache of 256kb, 4-way associative, 6-cycle latency. All benchmarks were fast-forwarded by 500 million instructions, and simulated for 500 million instructions. The DTC was 4-way associative with 128 sets for a total of 512 bursts. Each burst contained up to four different addresses with 8 bytes of data. The size of the data in the DTC totals 16kb. The hit latency of the DTC and the level-1 caches was 1 cycle, whereas the miss penalty was 6 cycles.

In order to evaluate the potential benefits of using a data trace cache, we ignore at first the long store latencies caused by the duplication of data across multiple data trace blocks. Fig. 6 shows the IPC improvement of the data trace cache. Without the store latencies, the data trace cache promises to deliver a high IPC improvement of 21% over a single data port machine on average.

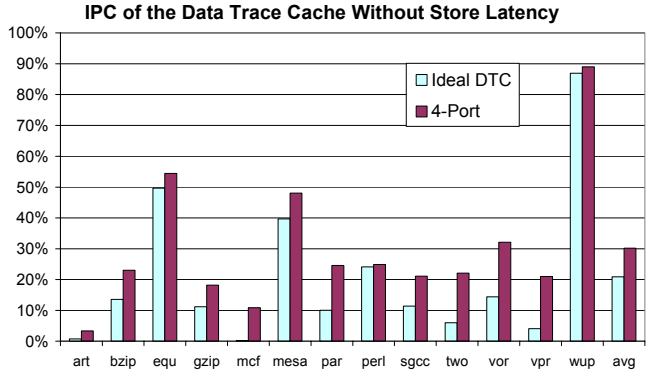

Fig. 6. IPC improvement of the data trace cache without accounting for store latencies on the various SPEC2000 benchmarks, compared with a single data cache port machine.

Fig. 7 shows the hit rate of the data trace cache. Consistent with the static burst coverage analysis in Fig. 3, the average hit rate of the data trace cache was 48%. In the case of GCC, the data trace cache architecture with 512 bursts achieved similar performance as can be achieved by using a static allocated data trace cache with 2000 bursts. The results indicate that there were indeed recurring bursts of data, but these recurring bursts were not prevalent enough for the data trace cache to exhibit a high hit-rate such as achieved by traditional level-1 data caches.

V. Comparison With Existing Methods

Two notable methods that aim at achieving multiport data cache performance are Load All Wide (LAW) and Line Buffer (LB) [7].

We have found that in the SPEC benchmarks, most of the best 64 bursts that cover the data access trace contain addresses from within the same data level-1 cache block. This indicates that the data access trace has *spatial locality*. For example, consider the stack accesses. At the end of functions, the previous values of the registers are popped off the stack. These accesses are to consecutive addresses in memory. Other examples of accesses to consecutive addresses include text manipulation, vector analysis, lossless compression, etc. Therefore, instead of issuing loads to the same cache block on different cycles, these loads can be combined and be issued on a single cycle. Load All Wide is a method that does exactly that.

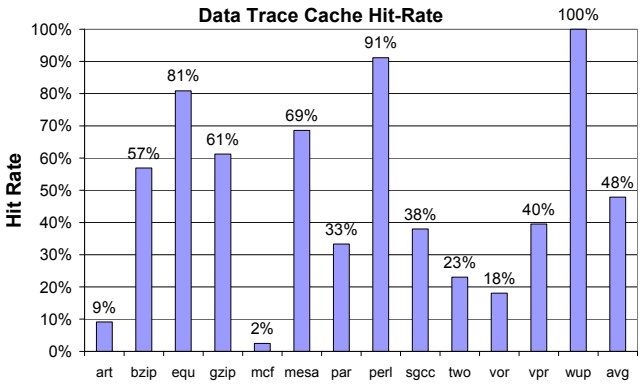

Fig. 7. The hit rate of the data trace cache for the different benchmarks, ranging from 2% to 100%. The results are consistent with the static burst coverage analysis.

The data access trace also has *temporal locality*. An access to a cache block will usually be shortly followed by another access to the same block. Line Buffer [7] is a method that sustains the last accessed blocks in a small multiported cache, enabling multiport access to the most recently accessed cache blocks.

Fig. 8 shows a comparison between the presented data trace cache and the existing methods LAW and LB. The size of the line buffer was 8 blocks, each containing 32 bytes for a total of 256 bytes. The results show substantial IPC improvement for LAW and LB, surpassing the ideal data trace cache on average even when store latencies are not accounted for. For similar performance, LB and LAW require considerably less area than the ideal data trace, which also needs additional area to keep the addresses of the different data in each data trace cache block.

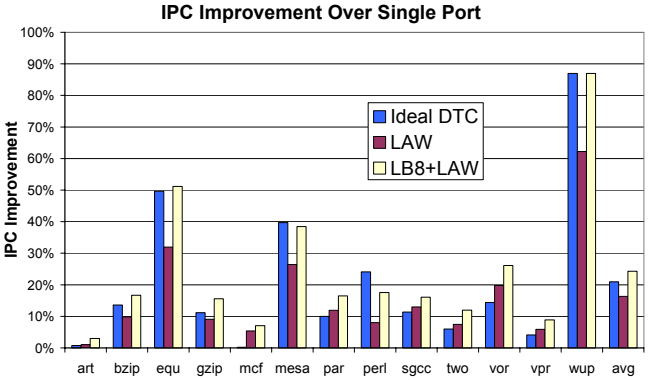

Fig. 8. The IPC improvement of the data trace cache, load all wide and line buffer with 8 blocks on the various SPEC2000 benchmarks, compared to a single data cache port machine.

VI. Accounting For Store Instructions

Since data may reside in multiple data trace cache blocks, executing a store instruction requires multiple accesses to the DTC. Each such write operation takes up one cycle to complete. Thus, the store latency grows with the amount of duplications inside the DTC. When such store latencies are taken into account, the IPC degrades substantially, since the flux of pending stores fills the write buffer. Therefore, in order for the data trace cache to be effective, the negative effect of stores must be minimized.

Limiting the number of recurrences of variables inside the DTC sets an upper bound for the store latency, shortening the average store latency. As a result, the size of the lookup table that is used to locate data among the different cache blocks is reduced as well. We have found that four recurrences are enough for covering the useful data traces while not incurring long store latencies.

Our simulations show that approximately 50% of the store instructions target the stack, indicating that stack data are very volatile. Since reducing the total number of stores to the data trace cache lowers the total cycles in which the DTC executes stores, we prevent stack data from entering the DTC. By limiting the number of recurrences of variables inside the DTC and by preventing stack data from entering, the overall store latency is reduced significantly. However, most of the potential of the DTC diminishes, as the average IPC improvement shrinks to a mere 2.5%. As a result of the need to update all of the copies of the same data inside the data trace cache by the store instructions, most of the potential promised by the data trace cache is eliminated.

VII. Discussion

In an effort to increase the data access bandwidth available to the processor, a data trace cache was proposed, in analogy to the instruction trace cache. However, simple methods were found to be more successful than the presented data trace cache.

The typical data access trace exhibits high temporal and spatial locality. Due to these localities, most of the dynamic loads that can be issued concurrently by the data trace cache can also be issued by LAW and LB. Moreover, LAW and LB can issue bursts of loads concurrently even if the burst pattern has not been previously encountered, contrary to the data trace cache which needs first to encounter the bursts in order to later optimize them. Therefore, high performance can be achieved by these simple methods, without dealing with the complexity associated with the data trace cache.

The rigid nature of the data bursts that were stored in the data trace cache contributed to its low performance. In instruction trace cache blocks, each basic block ends with a branch instruction, leaving only two options for the identity of the successive basic block. This is not the case in the data trace cache, since load instructions may access data from anywhere in memory. As a result, the data trace cache contains numerous redundancies, which consume space and lead to lower overall hit rate.

A major drawback of the data trace cache is the complexity of handling store instructions. Since multiple copies of the same data may reside in the DTC, executing store instructions involves updating multiple blocks. Unlike instruction traces, data traces tend to be updated much more than code. While barring stack data from entering the data trace cache reduces the overall store latency, it keeps a large portion of the DTC potential unrealized.

Another drawback of the data trace cache is its inability to effectively support partial hits, since all of the addresses are required in order to calculate the location of the correct data trace cache block.

The implementation of the Data Trace Cache as presented in this paper is very simple and straightforward. Its performance could be improved by using heuristics and by adding sophistication. However, we do not believe that these potential improvements would have changed our conclusions.

In conclusion, although there are recurring address patterns in the data access stream, we have found that the data trace cache is not cost effective compared with existing simple methods which exploit spatial and temporal locality.

Acknowledgements

We thank Avi Mendelson, Nir Magen, Antonio González and others from Intel Corporation who have helped us with this research. We dedicate this work to the memory of Nir Magen.

References

[1] Y. N. Patt, S. J. Patel, M. Evers, D. H. Friendly, J. Stark, “One Billion Transistors, One Uniprocessor, One Chip”, IEEE Computer, Vol. 30, Issue 9, pages 51-57, Sep. 1997.

[2] B. Calder; G. Reinman, “A Comparative Survey of Load Speculation Architectures”, Journal of Instruction Level Parallelism 1, pages 1-39, 2000.

[3] Tatsumi, Y.; Mattausch, H.J., “Fast quadratic increase of multiport-storage-cell area with port number”, Electronics Letters , Volume: 35 , Issue: 25, Pages:2185 - 2187, 9 Dec. 1999

[4] F. Mueller, T. Mohan, B.R. de Supinski, S.A. McKee, and A. YooProc , “Partial Data Traces: Efficient Generation and Representation”, PACT 2001 Workshop on Binary Translation, September 2001.

[5] S.A. McKee, D.A.B. Weikle, K.L. Wright, C.W. Oliver, A.P. Voss, M.H. Salinas, R.H. Klenke, T.C. Landon, Wm.A. Wulf, and J.H. Aylor, “Evaluation of Dynamic Access Ordering Hardware”, UVa Technical Report CS-95-50, October 1995.

[6] A. Peleg; U. Weiser, “Dynamic Flow Instruction Cache Memory Organized Around Trace Segments Independent of Virtual Address Line”, US Patent 5,381,533, March 30, 1994.

[7] K. M. Wilson, K. Olukotun, M. Rosenblum, “Increasing Cache Port Efficiency for Dynamic Superscalar Microprocessors”, Proceedings of ISCA-23, May 1996.

[8] Hinton, G.; Upton, M.; Sager, D.J.; Boggs, D.; Carmean, D.M.; Roussel, P.; Chappell, T.I.; Fletcher, T.D.; Milshtein, M.S.; Sprague, M.; Samaan, S.; Murray, R, “A 0.18-µm CMOS IA-32 processor with a 4-GHz integer execution unit”, Solid-State Circuits, IEEE Journal of , Volume: 36 Issue: 11, Page(s): 1617 –1627 , Nov 2001.

[9] D. Burger and T. Austin, “The Simplescalar Tool Set, Version 2.0”, Technical report CS-TR-97-1342, Univ. of Wisconsin, Madison, June 1997.