ACCEPTED: A. Beg and Y. Chu, "Utilizing Block Size Variability to Enhance Instruction Fetch Rate,"
Journal of Computer Science and Technology (JCST), vol. 7, p. 10, April 2007.

Utilizing Block Size Variability to Enhance Instruction Fetch Rate

AZAM BEG
abeg@uaeu.ac.ae
College of Information Technology, United Arab Emirates University,
Al-Ain, Abu-Dhabi, United Arab Emirates (UAE)
and
YUL CHU
chu@ece.msstate.edu
Department of Electrical & Computer Engineering, Mississippi State University,
Mississippi State, MS 39762, USA

ABSTRACT

In the past, instruction fetch speeds have been improved by using cache schemes that capture the actual program flow. In this paper, we elaborate on the architecture and operation of an instruction cache named Variable-Sized Block Cache (VSBC) that also makes use of the dynamic behavior of a program. Current trace-based cache schemes usually have some instructions stored repeatedly; this redundancy is eliminated in VSBC. Our cache also allows storage of basic blocks of arbitrary sizes, in multiple-way cache structure. An overall comparison of trace miss rate and average trace length shows VSBC to be a better performing cache scheme than TC, using SPECint2000 integer benchmarks.

Keywords: Basic blocks, instruction cache, trace cache, block cache

1. INTRODUCTION AND RELATED WORKS

Caching and *branch prediction* are two techniques that exploit the practical nature of common programs. Caching operation is based on the observation that the programs tend to access contiguous locations in memory (*spatial locality*) or the same memory locations repetitively (*temporal locality*). Effectively, the caches try to approximate the availability of an ideally large memory to the programmer. A fundamental limitation of a conventional instruction cache (IC) is that, due to *taken-branches*, only a single *basic block* can be fetched in a cycle. (A basic block is a set of instructions separated by a control instruction, such as a conditional or non-conditional jump) [1]. The technique of storage of basic blocks has been discussed in several research papers [2], [3], [4]. These techniques, however, still limited fetching of instructions to one or two basic blocks per cycle; this constraint was overcome by Rotenberg, et al's [5] *trace cache* (TC); it stored instructions as the program execution progressed. If the stored instruction sequence was encountered again during the program execution, the instruction sequence was delivered directly from TC to the instruction decoder. As a stored TC line was only accessible by its starting address (and intra-line basic block boundaries were not identifiable), TC suffered from excessive switching from *trace build mode* to *trace utilization mode*. TC's other drawback was redundancy of basic block storage [6], [7], [8], [9], [10]. Black, et al. [11] used basic blocks as units of instruction storage in cache and called this scheme *block cache* (BC). They added hardware complexity by introducing new structures to process traces. Drawbacks of this scheme were block fragmentation and storage of same basic blocks in multiple places. No follow-up research has been reported on BC, since its introduction. Additionally, TC is the only cache scheme that has been used in commercial processors, for example, Intel's Pentium-4. For these two reasons, our comparison of VSBC is being limited only to TC.

A thread is a set of instructions that starts execution at its first instruction and continues execution without interruption [12]. The threads can be generated either dynamically or statically. Dynamic thread generation involves creation and synchronization of some threads by another thread. There is a hardware and performance cost associated with thread communication and synchronization. Static thread models are simpler than dynamic thread models. Static threads are fixed in count, and are stored in the processor [13], [14]. Multi-threading when implemented on a single processor allows switching between threads in one cycle (or even zero cycle). If one thread faces long latency, the other thread may start executing. In a multi-threaded multi-processor system in [14], threads are stored locally to each processor but may migrate to other processors as well. A multi-threaded processor alters the way a memory is accessed. The cache effectiveness is reduced because of changed locality of reference [15].

2. OVERVIEW OF THIS PAPER

In this paper, we present *variable-sized block cache* (VSBC) architecture. VSBC addresses the issue of instruction overlap among traces that occur frequently in conventional TC[1]. VSBC enables storage of basic blocks without the replicated block storage structures as required in BC. VSBC's implementation in hardware is only slightly more complex than TC, but is simpler than BC. Traditional *n*-way associativity in VSBC further improves its performance. Unlike BC, VSBC allows storage of basic blocks of arbitrary lengths.

We compared VSBC's performance with TC by running SPECint2000 benchmarks [18] on single- and multi-threaded functional simulators of both cache schemes. We used only a single cache hierarchy in the simulators. The main focus of our research was VSBC's own performance and not that of a complete processor system. We chose *trace miss rate* and *average trace length* as the performance metrics.

---

[1] In a sim-cache-based [16], [17] TC model that ran 100 million instructions, we measured the instruction overlap among traces for SPECint2000 benchmarks [18]. Some measured values of overlap are: 25.1% for *crafty* benchmark, 38.5% for *mcf*, and 79.5% for *bzip*.

VSBC-ST refers to VSBC in single-threaded environment, and VSBC-MT refers to VSBC in multi-threaded environment. Section 3 describes VSBC architecture and Section 4 explains its operation. Section 5 covers the simulation and modeling results. Finally, Section 6 presents the conclusions.

3. VARIABLE-SIZED BLOCK CACHE – ARCHITECTURE

VSBC's Overall Structure

The VSBC stores instructions in program execution order. Each trace in VSBC is made up of a fixed number of basic blocks. A *hit* to the starting address of any of the basic blocks in a trace is considered a *trace hit*. Multiple branch predictions for end-of-block addresses are also required in a manner similar to the TC [5] and BC [11]. VSBC stores block info and block contents in two separate structures inside VSBC-storage module. The two structures are called *block pointer cache* (BPC) and *basic block cache* (BBC). An overall view of a VSBC-MT-based system is shown in Figure 1.

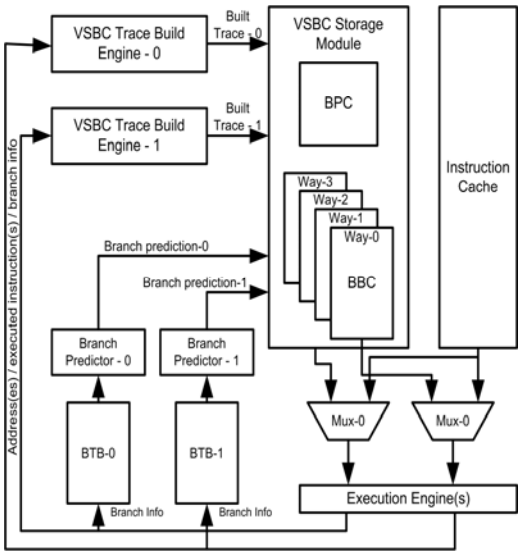

Figure 1: Overall view of a multi-threaded VSBC-based system. Here we show the two-threaded implementation.

VSBC Storage Module

VSBC storage module is mainly made up of two cache structures: BPC and BBC. The full address is used for BPC lookup, whereas BBC needs tag and index fields for lookup. A single line from BPC is shown in Figure 2. The BPC is made up of an array of these lines. Each BPC line corresponds to a single trace. BPC keeps track of valid basic blocks resident in the BBC. Upon detection of a block tail, full linear addresses for both block head and block tail are placed in a BPC line. Once all entries are populated, 'conflicts' start to occur and certain lines have to be replaced. LRU fields in BPC determine which BPC line will be evicted when there is a need for line replacement. *Branch status* bits store *taken* or *not-taken* status of the branches at the end of basic blocks. In Figure 2's BPC line, 3 bits are assigned to the first 3 blocks in a trace. Branch status for the last block is not saved. In VSBC-MT, *thread-ID* field identifies which thread the trace belongs to.

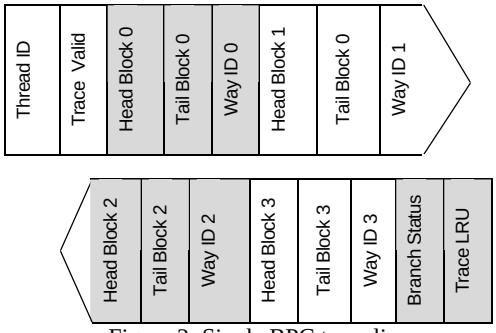

Figure 2: Single BPC trace line

The BBC is composed of two arrays: *BBC data array* and *BBC tag array*. BBC tag array stores tags and performs tag-matching, whereas basic blocks are stored in the BBC data array. Basic blocks can be of any size and are only limited by the number of lines in the BBC-way. The index and set information is derived from the head-addresses of basic blocks. (Head addresses are stored in BPC. The BBC-way, in which a particular basic block resides, is also stored in BPC). An additional field of *thread-ID* is used in multi-threaded VSBC (Figure 3).

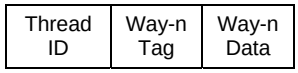

Figure 3: Single BBC line

The basic blocks from a trace can be stored in one or more BBC ways. So BBC data array allows reading of up to 4 blocks in one cycle. Each way has 4 read ports; each port is 16 instructions wide. (16 is an arbitrary limit used in this research). This means that the read ports are capable of supplying a maximum of 4 blocks placed at different locations in each way.

VSBC Trace Build Engine

As shown in Figure 1, each thread in a VSBC-MT system needs its own *trace build engine*. The build engine is quite simple in nature, and primarily consists of a *trace build buffer* (TBB). The *head address* is stored in TBB, one cycle after end of a block is detected. *Tail address* is the address of the control instruction that terminates the currently executing basic block. If a conditional branch ends the block, the *branch status* gets filled. After all TBB fields have been filled, TBB contents are copied into BPC.

Coalescing Buffer

A single trace is made up of basic blocks that may be stored in one or more ways. The task of *coalescing buffer* (Figure 4) is to read the basic blocks from BBC, rearrange and align them, and then send them to the decoder and the execution engine. Depending on the implementation, coalescing buffer can perform its function on a single trace in one cycle. This buffer is replicated for every thread (Figure 1).

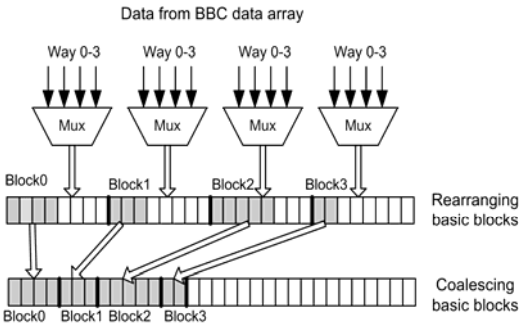

Figure 4: Coalescing Buffer

4. VARIABLE-SIZED BLOCK CACHE – OPERATION

A VSBC-based system essentially operates in two modes: *trace assembly mode* and *trace delivery mode*. The logic inside VSBC-storage module is responsible for deciding VSBC's operating mode. In a VSBC-ST-based system, when the program initially starts running, there is a miss on the VSBC and a single line is requested from the instruction cache, which in turn has to fetch it from the main memory. VSBC does not have any valid data in it at this time and VSBC is in trace assembly mode. As instructions execute, they get stored at appropriate locations in BBC structure inside VSBC storage module. Concurrently, the head and tail addresses of basic blocks are identified and stored in TBB in VSBC trace build engine. After the end-of-block condition is recognized, TBB contents are copied to a BPC line. The BBC-way, in which this block was being stored, is also placed in BPC. After a fixed number of TBB writes to BPC line are done, the trace is considered built.

Three conditions have to be satisfied for a trace hit: (1) current address matching any block *head address* in BPC, (2) tag matching in BBC, and (3) matching of BBC branch bits to predicted branches. Upon a trace hit, VSBC switches to *trace delivery mode* and instructions from BBC are supplied to the decoder and the execution engine.

VSBC's operation in multi-threaded mode is similar to the single-threaded mode. As mentioned earlier, the difference here is that multiple threads get their basic blocks built in their own trace build engines. Each thread also gets its own branch history table and branch predictor [19]. VSBC storage module may see simultaneous write or read requests and has to process them in the round-robin fashion. In our study, we opted for allocation of dedicated BPC lines to threads but kept BBC as a thread-shared resource.

5. VARIABLE-SIZED BLOCK CACHE – SIMULATION & MODELING

We created trace-based functional simulators to study VSBC and to compare its performance with TC. The simulators did not provide any timing information, such as cache latency. For performance comparison, we used 10 benchmark programs (listed in Table 1) from SPECint2000 suite [18]. The programs were compiled with *gcc* compiler (version 2.7.2.2 using -O0 option). Using these programs, we created single- and multi-threaded workloads (sets of traces), as given in Table 2. The simulation parameters are listed in Table 3.

Table 1: Benchmark programs for comparing VSBC with TC

Benchmark	Description	Input Data Set
bzip	Compression	input.random
crafty	Game playing: chess	crafty.in
gap	Group theory, interpreter	test.in
gcc	C language compiler	cccp.i
gzip	Compression	input.compressed
mcf	Combinatorial optimization	inp.in
parser	Word processing	test.in
perlbmk	PERL language	test.pl, test.in
vortex	Object-oriented database	lendian.raw
vpr	FPGA circuit placement & routing	net.in, arch.in

Table 2: Workloads for single- and multi-threaded simulations

Workload/ Mix #	Thread Count	Benchmarks
WL0a-WL0j	1	bzip, crafty, gap, gcc, gzip, mcf, parser, perlbmk, vortex, vpr
WL1	2	bzip, crafty
WL2	2	gap, gcc
WL3	2	parser, perlbmk
WL4	2	vortex, vpr
WL5	4	bzip, crafty, gap, gcc
WL6	4	gap, gcc, gzip, mcf
WL7	8	bzip, crafty, gap, gcc, gzip, mcf, parser, perlbmk
WL8	8	gap, gcc, gzip, mcf, parser, perlbmk, vortex, vpr
WL9	16	bzip, crafty, gap, gcc, gzip, mcf, parser, perlbmk, gap, gcc, gzip, mcf, parser, perlbmk, vortex, vpr

Table 3: Simulation parameters for TC and VSBC

Parameter	TC	VSBC
Number of lines in BPC	N/A	512, 1024
Max number of traces	512, 1024	512, 1024
Number of ways in TC/BPC	1	1
Cache capacity (KB)	1K, 2K, 4K, 8K, 16K	1K, 2K, 4K, 8K, 16K
TC/BBC associativity	1 way	1-way, 2-way, 4-way
Number of threads	1, 2, 4, 8, 16	1, 2, 4, 8, 16
Max basic blocks per trace	4	4
Max possible number of instructions per trace	16	Not limited
Max number of instructions delivered per cycle	16	16
Branch history table size	1024 entries	1024 entries

VSBC's Comparison with TC

We ran simulations for different configurations of TC and VSBC to collect the performance data. In order to make a comparison for a similar amount of hardware, we used the same cache size for TC and VSBC. For example, a VSBC (BBC) of 1K capacity was compared with the TC of 1K capacity. With cache sizes of 1K, 2K, 4K, 8K, and 16K, the simulations were run for both caches (in single-way configuration). For single-threaded workloads (WL0a-WL0j in Table 2), the trace miss rate and average trace length comparisons are shown in Figure 5 and Figure 6, respectively. Similar comparisons for multi-threaded workloads (WL1-WL9 in Table 2) are shown in Figure 7 and Figure 8. The notations in Figure 5 and Figure 6 can be understood with these two examples: "bzip 1K" represents the miss rate or trace length comparison for *bzip* benchmark when run on a 1K cache; and "crafty 8K" represents the miss rate or trace length comparison *crafty* benchmark when run on an 8K cache. The notations of Figure 7 and Figure 8 are explained with two more examples: "WL1_2thd_1K" stands for the relative miss rate or trace length when a WL1 (2-thread) workload is run on a 1K cache, and "WL7_8thd_8K" stands for the relative miss rate or trace length for a WL7 (8-thread) workload when run on an 8K cache.

In the single-threaded environment, VSBC's miss rate reduction over TC varied from 43% to 95%, yielding an average improvement of 73.7% (Table 4). The miss rate reduction percentages dropped slightly when cache sizes were increased. Larger block benchmarks (e.g., *crafty, gcc, gzip, perlbmk*) had better miss rates than smaller block benchmarks. With the multi-threading workloads, VSBC consistently performed better than TC with trace miss rate improvements ranging from 69% to 95%; the average improvement was 85.7% (Table 4).

The miss rate performance gains over TC are made possible by reduction in the block overlap among the traces. VSBC with 1K trace capacity has miss rates comparable to 16K TC. However, if we keep increasing TC's cache capacity, its performance gap with VSBC will start to narrow. To further improve VSBC's performance, use of a better branch prediction scheme is recommended. Hossain suggested 98% or higher accuracy of branch prediction to utilize the full potential of a trace-based cache [20].

Trace length gains varied widely in single-threaded environment (Figure 6). On the lower side, VSBC's trace length gains ranged from -10% to 7%, for five of the benchmarks; for the other five benchmarks, the gains ranged between 70% and 254% of the TC trace lengths. For single-threaded workloads, the overall improvement in trace lengths was 79.7% (Table 4).

For the multi-threaded workloads, VSBC's trace lengths improvements over TC ranged from -3% to 293%, with an average improvement of 86.1% (Table 4). While multi-threading, BPC gets equally divided among the threads. For example, for dual threads, half the BPC lines are dedicated to one thread and the other half to the other thread. On the other hand, all BBC lines are open to all threads, which can cause the traces from different threads to clobber each other. The combination of reduced BPC capacity per thread and the inter-thread collisions may be the reason for a wide variation of performance while multi-threading.

A point to note is that if BBC were configured in such a way that dedicated BBC-lines were assigned to each thread, we would essentially have the equivalent of multiple instances of completely independent single-threaded VSBC-based systems; for this reason, it does not make sense to simulate VSBC with dedicated-line BBC configurations.

VSBC's Design Space Study

As the subject of this research is VSBC itself, we conducted additional simulations to study VSBC's own design space (Table 5). Regarding, the sensitivity of VSBC's miss rate to BBC size, we observe the expected improvement in miss rate, when BBC size is increased. In response to change in BBC-associativity, we see the usual cache behavior of gradually flattening miss rates with higher associativity. Multiple threads cause the miss rate to vary widely which can be attributed to the change in locality of reference. One can also see that the trace lengths remain relatively unchanged despite variation in BBC size. If VSBC operation is changed in such way that a (smaller length) *partial trace hit* (explained earlier) starts new trace builds, the average trace lengths may improve further. Change in BBC-associativity does not affect the trace length much. This invariability is because the blocks belonging to a given trace are stored in a single way; availability of additional BBC-ways does not benefit trace lengths. We, however, observe a wide variation in trace lengths in the multi-threading environment; variations in miss rate and trace length seem to be the result of clobbering of traces by different threads in shared- BBC.

6. CONCLUSIONS

VSBC architecture presented in this research paper eliminates some of the drawbacks that similar cache schemes have. VSBC avoids frequently occurring instruction overlap among TC traces. VSBC does not have BC-like redundant block storage structures and the related complexity of hardware.

We compared VSBC with the baseline TC by running SPECint2000 benchmarks on single- and multi-threaded TC and VSBC functional simulators. A 1K VSBC provides similar miss rates as a 16K TC. Use of a better branch predictor is expected to further improve VSBC's performance. VSBC sustains its lead over TC in multi-threaded mode. Using the performance criteria of trace miss rate and average trace length, VSBC seems to be a better performing cache scheme than TC.

Table 4: Trace length and miss rate comparison for single- and multi-threaded environments (BBC = 1KB, 2KB, 4KB , 8KB, 16KB; N_{BPC} = 512; W_{BBC} = 1)

Workload	Average TC trace length	Average VSBC trace length	VSBC vs. TC trace length gain	Average TC miss rate	Average VSBC miss rate	VSBC vs. TC miss rate reduction
WL0a-WL0j	12.5	24.3	**79.7%**	15.6	4.4	**73.7%**
WL1-WL9	12.6	24.0	**86.1%**	45.8	5.9	**85.7%**

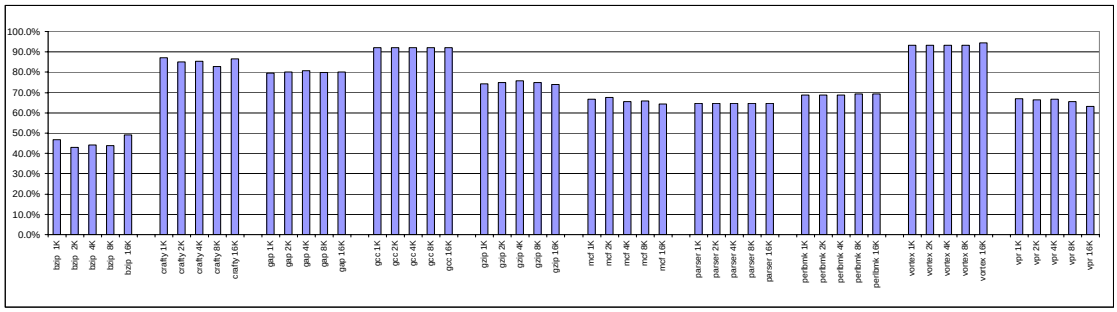

Figure 5: VSBC's miss rate gain with TC in single-threading environment. On average, VSBC is 73.7% better than TC. (Workload = WL0a-j; N_{BPC} = 512; W_{BBC} = 1).

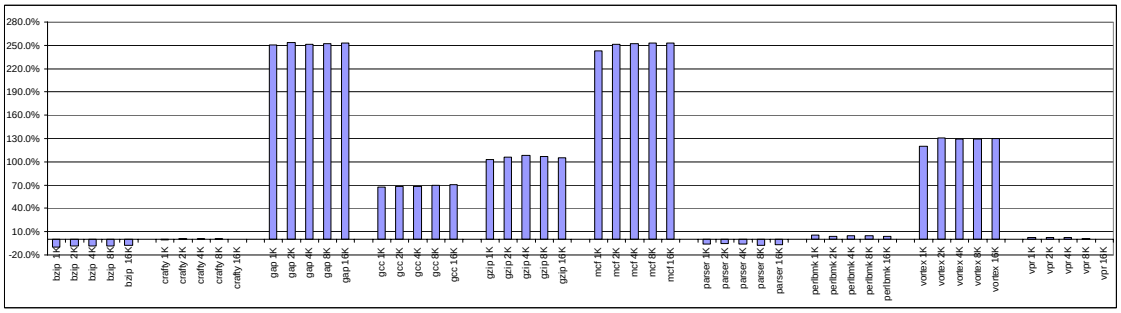

Figure 6: VSBC's trace length gain with TC in single-threading environment. On average, VSBC is 79.7% better than TC. (Workload = WL0a-j; N_{BPC} = 512; W_{BBC} = 1).

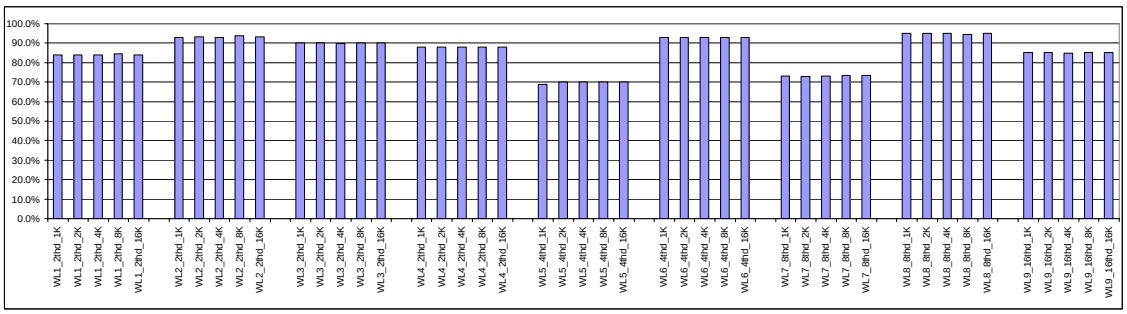

Figure 7: VSBC's miss rate gain with TC in multi-threading environment. On average, VSBC is 85.7% better than TC. (Workload = WL1-9; N_{BPC} = 512; W_{BBC} = 1).

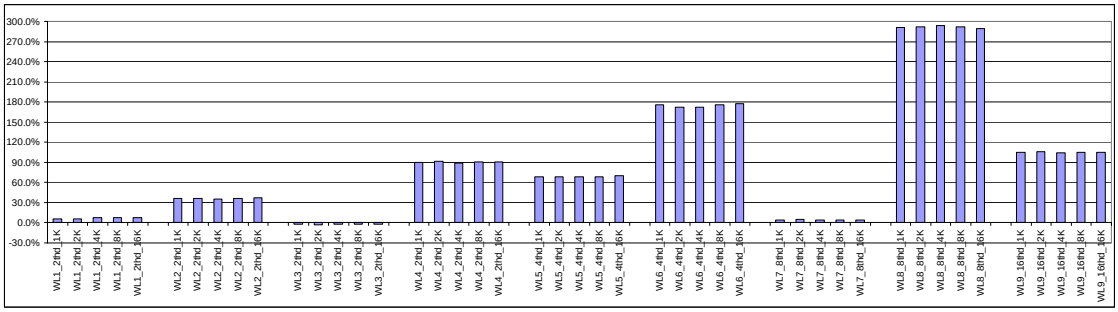

Figure 8: VSBC's trace length gain with TC in multi-threading environment. On average, VSBC is 86.1% better than TC. (Workload = WL1-9; N_{BPC} = 512; W_{BBC} = 1).

Table 5: VSBC's design space study

Sensitivity to VSBC cache (BBC) size (Workload = WL0a-j; N_{BPC} = 512; W_{BBC} = 1; N_{th} = 1)						
BBC ->	**1K**	**2K**	**4K**	**8K**	**16K**	**Comments**
Miss rate	5.9	5.1	4.1	3.6	3.1	A drop in miss rate happens with increase in BPC capacity
Ave. trace length	23.9	24.3	24.4	24.2	24.4	The trace length is relatively insensitive to cache size

Sensitivity to VSBC-BBC associativity (Workload = WL0a-j; BBC = 1KB; N_{BPC} = 512; N_{th} = 1)					
W_{BBC} ->	**1**	**2**	**4**	**8**	**Comments**
Miss rate	5.9	4.8	4.7	4.7	After an initial drop, the miss rate flattens out with an increase in associativity
Ave. trace length	23.9	24.3	24.2	24.3	The trace lengths are not affected noticeably with the change in BBC-associativity

Sensitivity to workload thread count (Workload = WL1-9; BBC= 4KB; N_{BPC} = 1024;W_{BBC} = 4)						
N_{th} ->	**1**	**2**	**4**	**8**	**16**	**Comments**
Miss rate	4.4	5.9	7.1	4.3	7.3	Miss rates vary widely possibly due to cross-thread trace clobbering in BBC
Ave. trace length	24.3	14.7	31.6	34.2	25.9	Trace lengths also fluctuate apparently due to cross-thread trace clobbering in BBC

7. REFERENCES

[1] J. Hennessy, and D. Patterson, "Computer Architecture: A Quantitative Approach," 3rd ed., Morgan Kaufman Publishers, Inc, CA, 2003.

[2] E. Hao, P.Y. Chang, M. Evers, and Y. Patt, "Increasing the Instruction Fetch Rate via Block-Structured Instruction Set Architectures," Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture, pages 191 – 200, 1996.

[3] S. Dutta, and M. Franklin, "Control Flow Prediction with Tree-Like Sub-Graphs for Superscalar Processors," Proceedings of the 28th Annual International Symposium on Microarchitecture, pages 258 – 263, 1995.

[4] S. Dutta, and M. Franklin, "Control Flow Prediction Schemes for Wide-Issue Superscalar Processors," IEEE Transactions on Parallel and Distributed Systems, Volume 10, Issue 4, pages 346 – 359, April 1999.

[5] E. Rotenberg, S. Bennett, and J. E. Smith, "A Trace Cache Microarchitecture and Evaluation," IEEE Transactions on Computers, Volume 48, Issue 2, pages 111 – 120, February 1999.

[6] D.L. Howard, and M.H. Lipasti, "The Effect of Program Optimization on Trace Cache Efficiency," Proceedings of International Conference on Parallel Architectures and Compilation Techniques, pages 256 – 261, 1999.

[7] Q. Jacobson, and J. Smith, "Trace Preconstruction," Proceedings of the 27th International Symposium on Computer Architecture, pages 37 – 46, 2000.

[8] S. Patel, D. Friendly, and Y. Patt, "Evaluation of Design Option for the Trace Cache Fetch Mechanism," IEEE Transactions on Computers, Volume 48, Issue 2, pages 193 –204, Feb 1999.

[9] S. Patel, M. Evers, and Y. Patt, "Improving Trace Cache Effectiveness with Branch Promotion and Trace Packing," Proceedings of 25th Annual International Symposium on Computer Architecture, pages 262 –271, 1998.

[10] D. Howard, and M. H. Lipasti, "The Effect of Program Optimization on Trace Cache Efficiency," Proceedings of International Conference on Parallel Architectures and Compilation Techniques, pages 256 – 261, 1999.

[11] B. Black, B. Rychlick, and J. Shen, "The Block-Based Trace Cache," Proceedings of the 26th International Symposium on Computer Architecture, pages 196 – 207, 1999.

[12] K. Kavi, A. R. Hurson, P. Patadia, E. Abraham, and P. Shanmugam, "Design of cache memories for multi-threaded dataflow architecture," Proceedings of the 22nd

Annual International Symposium on Computer Architecture, June 1995.
[13] P. Kakulavarapu, et al, “A comparative performance study of a fine-grain multi-threading model on distributed memory machines,” Proceeding of the IEEE International Conference on Performance, Computing, and Communications Conference, Feb. 2000.
[14] L. Alkalaj, “Performance of multi-threaded execution in a shared-memory multiprocessor,” Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing, Dec. 1991.
[15] D. Lioupis, and S. Milios, “Exploring cache performance in multithreaded processors,” Microprocessors and Microsystems, Volume 20, Issue 10, July 1997.
[16] D. Burger, and T. Austin, “The SimpleScalar Tool Set, Version 2.0,” University of Wisconsin-Madison Computer Sciences Department Technical Report #1242, June 1997.
[17] http://www.simplescalar.com/
[18] http://www.specbench.org/cpu2000/
[19] J. Gummaraju, and M. Franklin, “Branch prediction in multi-threaded processors,” Proceedings of International Conference on Parallel Architectures and Compilation Techniques, Oct. 2000.
[20] A. Hossain, “Trace Cache in Simultaneous Multi-threading,” PhD dissertation, Dept. of Computer Engineering, Syracuse University, 2002.

ABOUT AUTHORS

Azam Beg received his PhD degree in computer engineering from Mississippi State University, USA. Since 1987, he has worked in the diverse fields of computer architecture, electronics, and control systems. In 2005, he joined College of Information Technology, United Arab Emirates (UAE) University, UAE as an assistant professor. Before moving to academia, his last 9 years were spent at Intel Corporation, USA working on test, validation, and design of flash memory and microprocessors. His current research focuses on computer architecture and artificial intelligence applications.

Yul Chu received his PhD in computer engineering from University of British Columbia, Canada. Presently, he is an assistant professor with the Department of Electrical and Computer engineering, Mississippi State University, USA. His research interests include hardware, compilers, operating systems and network schemes for high performance computing. His interests also include embedded cache and branch prediction schemes for FPGA/VLSI architectures.