Trace Cache Sampling Filter

Michael Behar
Department of Electrical Engineering
Technion, Haifa, Israel
behar@tx.technion.ac.il

Avi Mendelson
Intel Corporation
Haifa, Israel
avi.mendelson@intel.com

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

Abstract

This paper presents a new technique for efficient usage of small trace caches. A trace cache can significantly increase the performance of wide out-of-order processors, but to be effective, the size of the trace cache should be large.

Power and timing considerations indicate that a small trace cache is desirable, with special mechanisms to increase its effectiveness despite the limited size. Hence several authors have proposed various filtering methods to select "good traces" for keeping in the trace cache, from among the general population of traces.

This paper presents a new filtering technique, which is based on sampling. Our new technique suggests that instead of building all the traces and trying to select the good ones among them, it is more efficient to make a preliminary selection of traces. This selection is based on a random sampling approach.

We show that the Sampling Filter improves trace cache and overall system performance, while reducing power dissipation. The Sampling Filter reduces admission of traces that are not used prior to their eviction from the cache, and prolongs the percentage of time a trace is in its live phase during its stay in the cache. Moreover, the Sampling Filter reduces duplication between the trace cache and the instruction cache and thus reduces the overall misses in the first level of cache hierarchy.

1. Introduction

The effectiveness of a single threaded out-of-order processor is strongly dependent on the average number of useful instructions it can fetch in every cycle. Trace caches are very effective in serving this goal by storing instructions in their dynamic order rather than in their static order [11][16][17]. Thus, a trace cache has two major merits: it contains instructions from different basic blocks that would normally require several accesses to the instruction cache in order to be fetched, and it contains only useful instructions (contrary to an instruction cache line that is fetched but might include instructions that the processor doesn't require). The trace cache was found to be very effective if size and power are not limited. However, relatively small trace caches, which are practical in term of access time and power consumption, are very vulnerable in terms of memory efficiency, because of several problems [12]: the same basic blocks can appear in different traces (duplication), some traces don't contain the maximum number of instructions (fragmentation), and the requested block might reside inside a trace and thus become inaccessible (Indexability). Thus, the key for efficient usage of relatively small trace caches is either to change the way the trace is constructed; i.e., the use of basic block traces [1], or to keep only the most valuable traces inside the cache and thus avoid their trashing by less valuable traces. This work will focus on the second option since the logic required to construct the traces out of a block-based cache is very costly in term of power. Filtering (selective admission of incoming traces to the cache) has already been proposed as a way to increase the usefulness of a limited size trace cache. In [13] it was proposed to store only traces containing taken branches, which cannot be fetched in one access from the instruction cache. In [14] it was proposed to filter traces based on their usage. The trace cache is divided into two blocks: the Filter trace Cache (FTC) and the Main Trace Cache (MTC). All traces are written to the FTC, but only traces that have been proven to be "useful" are inserted to the MTC. The success of this filtering method is based on the important observation, that most traces that are built and inserted to the trace cache are rarely used before eviction, and that most of the instructions the processor executes come from a small set of traces ("hot traces") [14]. In [9] it was proposed to use profiling in order to filter out traces that are less frequent and show little time locality.

This paper presents a new class of trace filtering techniques, which is based on statistical sampling of traces. This class of filters aims to improve the quality of the traces residing in a small trace cache, while reducing the power dissipation needed for maintaining the filter's bookkeeping. The paper presents and analyzes the performance and the power of a basic Sampling Filter (SF) and an enhanced version of it, and compares its performance and power with a regular trace cache and with the FTC-MTC organization.

The rest of the paper is organized as follows: In Section 2 the simulation environment and the characterization of traces are discussed; Section 3 describes the sampling filter architecture and compares it with the regular trace cache and with the FTC-MTC organization. In Section 4 the usage of the Sampling Filter with the FTC-MTC organization is demonstrated and Section 5 concludes and proposes related ideas for future studies.

2. Simulation environment and basic observations

This section presents the simulation environment we used, and some of the basic observations that our new proposed technique is based upon.

2.1. The simulation framework

The performance numbers presented below are based on an extended version of the sim-outorder simulator from the SimpleScalar tools set 3.0d [4] that was augmented with a detailed model of the trace cache that includes the impact of wrong path prediction and recovery, along with the simulation of the proposed filter mechanism and with a next trace predictor [7]. The power numbers presented in this paper were computed by an extended version of the Wattch [2] simulator (which is based on Simple scalar) and by Cacti [18] that was used to estimate the power of new structures such as the trace cache, and for examining the impact of power and timing on caches with different configurations. The modeling of the leakage power in Wattch assumes that it always consumes 10% of the maximal dynamic power. In this work, since we assume relatively small structures of the trace cache, we saw that the leakage has only minor impact on the overall power of the chip and so we use the same method as in Wattch.

The structure of the traces within the trace cache we use for our model is similar to other works; i.e., a trace can contain up to 16 instructions and up to 4 branches, a trace is terminated also if it reaches indirect jumps, indirect branches, procedure-calls, return instructions and interrupts. Our traces are composed of basic blocks and we don't allow traces to be truncated unless a single basic block is larger than the trace capacity (in our framework: 16 instructions). We allow loop unrolling and don't terminate a trace upon a backward branch. This allows traces to contain enough instructions to have an advantage over the regular fetch mechanism and yet not increase the number of unique traces too much. In this paper we focus on two sizes of the trace cache: one that contains 32 traces, and another that contains 64 traces, both organized as 4-way set-associative. Our trace build mechanism allows traces beginning at the same address, but with multiple paths, to coexist in the trace cache. In this case, it is up to the trace predictor to decide which of the traces to select. The work assumes a trace cache with a backing instruction cache, which are accessed in parallel as was described in [17].

Table 1. μarch settings of the simulated model.

Execution engine	
Decode, Issue, Commit width	8
Functional units	Integer ALU's: 8 4 Mult/Div. Floating point ALU's: 8 4 Mult/Div.
Fetch queue size	32
Register update unit	128
LSQ	64
Memory	
L1 Data Cache	32KB 8-ways LRU, 64B blocks. 2-cycle latency.
L1 Instruction Cache	8KB 4-way, LRU, 32B blocks, 2-cycle latency
Trace cache	2KB (32 traces)/4KB (64 traces) 4-way, LRU, 2-cycle latency
L2 Unified cache	1MB 8-ways, 64B blocks, LRU, 10-cycle latency
Memory	First chunk: 128 cycles
TLB	30 cycles miss penalty
Branch predictor	
Predictor	Bimod 4k-entery
RAS	32
BTB	2K-entery, 4-way
Next trace predictor	4K-entery

The configuration of the baseline machine is presented in Table 1. We chose an 8-way machine as a baseline

since it can take advantage of the improvement in instruction supply and still be power efficient.

We used 10 benchmarks (See Table 2) from the SPEC2000 Benchmark Suite [5] to evaluate our work. We skipped the first 500M instructions and simulated another billion instructions in all our experiments except perlbmk that was ended after 880M instructions.

Table 2. benchmarks list.

Benchmark	Input	Suite
164.gzip	input.graphic	INT
175.vpr	net.in arch.in place.in	INT
176.gcc	166.i	INT
197.parser	2.1.dict –batch ref.in	INT
253.perlbmk	makerand.pl	INT
255.vortex	lendian1.raw	INT
256.bzip	input.graphic	INT
177.mesa	mesa.in mesa.ppm	FP
183.equake	inp.in	FP
168.wupwise	wupwise.in	FP

2.2. Basic observations

We start this section by presenting basic characterization of the utilization of each trace in the trace cache, and characterization of the utilization of the trace cache itself.

Trace Utilization (TU) is defined to be the number of times the system finds the trace in the trace cache per a trace build. Please note this definition does not require that the traces will be unique; i.e., if a trace is replaced and built again, we count it as two different traces. Also, the length of the trace does not affect the utilization of the trace. In Figure 1. and Figure 2. the trace utilization breakdown is presented for 32-traces and 64-traces trace caches respectively. In both configurations the majority of traces that are written to the trace cache are not used prior their eviction from it (TU=0). Moreover, only 10% of the writes results with more than 2 hits (TU>2) for the 32-traces trace cache and 20% for the 64-traces trace cache.

The trace cache utilization (TCU) id defined in equation 1. as the number of hits in a cache divided by the number of writes. This parameter gives a good sense of the power and performance efficiency of the trace cache. A trace cache with a high TCU is assumed to be power and performance efficient. For a 32-traces trace cache the average TCU is five. Moreover, five out of the ten benchmarks we examined have a TCU smaller than 2. Therefore, the power invested in building the trace and writing it to the trace cache is very poorly used.

$$TCU = \frac{\sum hits}{\sum writes}. \quad (1)$$

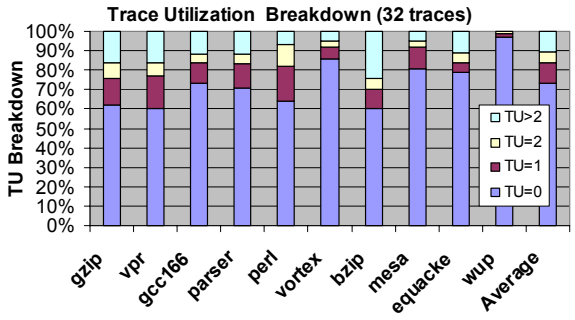

Figure 1. Trace utilization breakdown for a 32-traces trace cache.

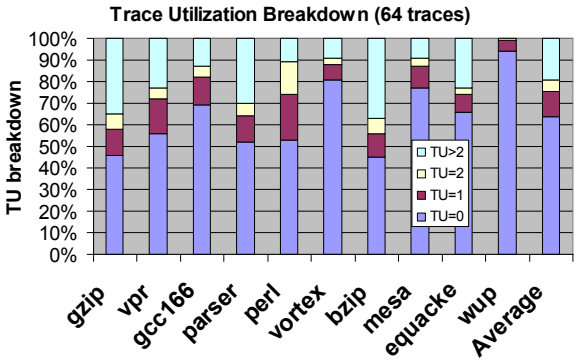

Figure 2. Trace utilization breakdown for a 64-traces trace cache.

The replacement rate (RR) of traces in the trace cache is defined in equation 2.

$$RR = \frac{\sum replacements}{\sum accesses}. \quad (2)$$

Figure 3. shows the replacement rate of a 32-traces and 64-trcaes trace caches. It is clear from the figure that for some applications (like Bzip), even a small trace cache can contain the whole program, but for other applications such a trace cache is too small. On average, the replacement rate is high (34% and 22% for a 32-trace and 64-trace caches respectively).

The above observations indicate that for a limited area trace cache, traces are replaced too frequently by less effective traces and cause the entire trace cache

mechanism to be ineffective in terms of performance and power. Therefore, it is critical to filter "bad" traces out of the general population of traces.

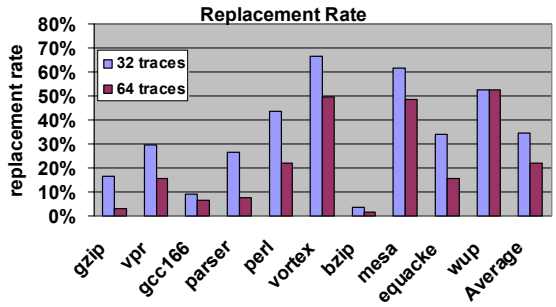

Figure 3. Replacement rate for 32-traces and 64-traces trace caches.

3. THE SAMPLING FILTER

Unlike other proposed filtering techniques that try to keep track of all traces in the program in order to classify them as "hot traces" (that need to be kept) or "cold traces" (that can be discarded), the new proposed technique uses a statistical approach. By using statistical methods, we suggest to randomly select traces that are candidates for storing in the trace cache. Please note that by doing so, we do not preclude any other filtering techniques, which can be applied on the chosen subset of the traces.

The structure of a system that supports the basic sampling algorithm is shown in Figure 4. On top of a trace cache system as described in [17] we add a sampling capability that chooses periodically, for example every X builds, to save a trace. Traces that are not sampled (selected) are discarded. The sampling rate is the rate at which traces are sampled, i.e., if every tenth trace is inserted to the trace cache, the sampling rate is 1/10. This filtering mechanism requires minimal hardware and can be easily implemented.

In order to establish the new proposed technique effectiveness, the next subsection provides some performance (IPC) and power efficiency (ED^2) simulation measurements as well as trace cache behavior (hit rate and coverage). We choose the energy delay square metric (ED^2) as it is independent of voltage in first approximation [2]. Next we will extend the discussion in order to understand why the sampling filter works and how it can be further improved.

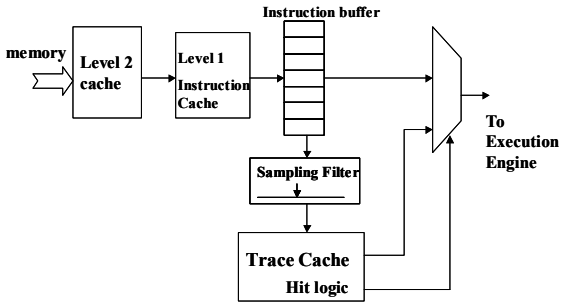

Figure 4. The sampling filter system.

3.1. The Impact of the Sampling filter

In this section we compare several fetch engine configurations. The different configurations and area budget are summarized in Table 3. The regular trace cache (CTC32) and the Sampling Filter (SF32) machines all have a 8KB backing instruction cache and a 2KB trace cache size. The FTC-MTC organization has also an 8KB backing instruction cache and a 2KB total trace cache that is divided equally between the FTC and the MTC. The SF32 uses a constant sampling rate of 1/20 for all the benchmarks.

Table 3. Fetch engines configurations

CONFIGURATION NAME	CONFIGURATION DESCRIPTION	TOTAL AREA
I8KB	Instruction cache	8KB
I16KB	Instruction cache	16KB
CTC32	Concurrent trace cache	10KB
FTC-MTC	Filter trace cache + Main trace cache	10KB
SF32	Concurrent trace cache with a sampling filter	10KB

Figure 5. shows the IPC improvement of these fetch engines over a machine without a trace cache, which has an 8KB instruction cache only (I8KB). Doubling the Instruction cache to a 16KB cache (I16KB) improves the IPC by 9.6%, while the regular trace cache (CTC32) improves performance by 11%. The FTC-MTC achieves 11.6% improvement while the Sampling Filter (SF32) achieves 17.7% improvement. This demonstrates that the combination of a small trace cache (total area of 10KB) and sampling technique can outperform a larger instruction cache (16KB) and the other trace cache organizations occupying the same area.

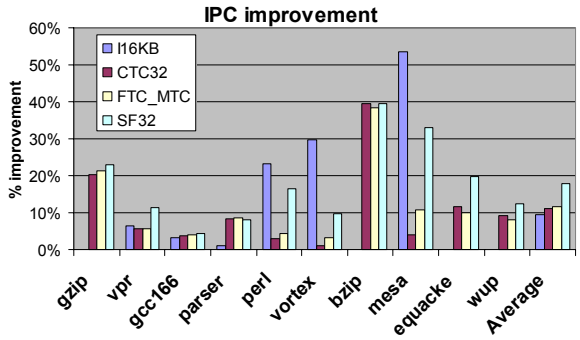

Figure 5. IPC improvement over a regular 8KB instruction cache.

Figure 6. shows the reduction in energy delay squared (ED^2) of several fetch engines compared with a regular 8KB instruction cache. The 16KB instruction cache achieves a reduction of 14.5% in ED^2. The trace cache and the FTC-MTC organization achieve 16.5% and 20% reduction in ED^2 respectively, while the Sampling Filter achieves a 27% reduction in ED^2. This indicates that the sampling filter is the most performance-power efficient among the compared alternatives.

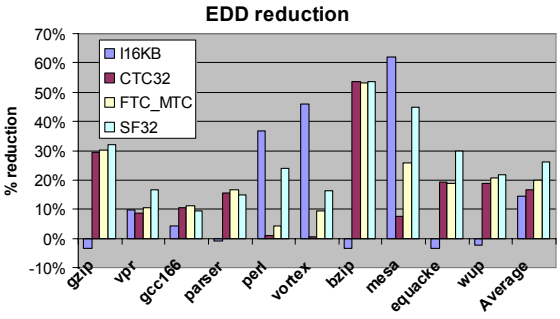

Figure 6. ED^2 reduction over a regular 8KB Instruction cache. The Sampling Filter proves to be the most performance-power efficient out of all the configurations.

The impact of sampling on the trace cache behavior is presented in Figure 7. The coverage (the percentage of instructions originated from the trace cache) of the Sampling Filter configuration compared with the regular trace cache increases from 56.5% to 66.3%. The hit rate of the Sampling Filter configuration increases from 66% to 72.6%.

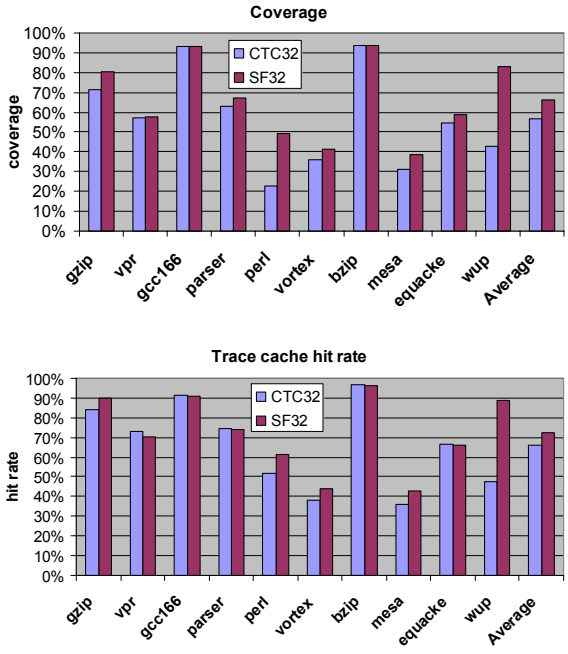

Figure 7. Trace cache coverage (top figure) and hit rate (bottom figure) of a regular trace cache (CTC32) and the sampling filter organization (SF32).

3.2. Why it work

The reason that our new technique works so well is a combination of two effects: the reduction of pressure of new coming traces on the small trace cache, together with the impact of the LRU mechanism. As was published in some research in the past, it is known that most of the instructions a trace cache based processor executes come from a relatively small number of traces ("hot traces"). These traces, regardless of the random selection, will be selected eventually, and will be placed in the trace cache. The main impact of the sampling filter is then, on the "cold traces". In section 2 it has been shown that the majority of writes are of "cold traces" with zero TU rate. The filter reduces the number of "cold traces" writes. This reduces the pressure on the small trace cache and enables the LRU mechanism to better capture the "hot traces", so "cold traces" that happened to enter the cache can be identified as such, and be replaced.

In order to justify the above claims, we present a new set of experiments. Figure 8. shows the impact of using the basic sampling algorithm on the trace utilization (TU), and in particular we focus on the percentage of traces that have TU=0. We can observe that the sampling technique reduces the population of these traces dramatically from 73.8% in the non-

filtered system to 25.6% in the sampling filter system. The impact of such reduction in the "useless traces" is twofold: it saves a lot of wasted power and it prevents cache pollution by inefficient traces.

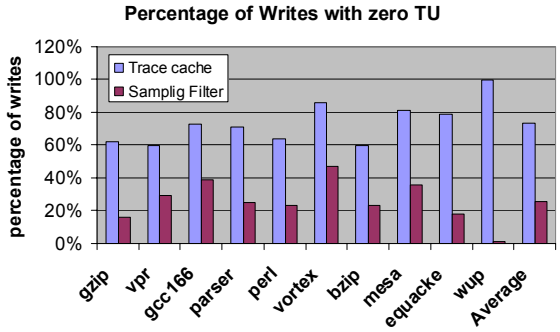

Figure 8. Percentage of writes with zero Trace utilization (TU).

An important indicator for the quality of a trace is the proportion between its "live" time and its "decay" time as was defined in [8][14]. A "live" time of a trace is measured from the time it was saved in the trace cache till the last time it was used. The "decay" time of a trace is measured from the last time it was used, until its eviction from the trace cache. Since the decay time is considered to be a waste of resources, we try to reduce it. Figure 9. shows the impact of the sampling technique on the lifetime of traces. While in the regular trace cache the average "live" time of a trace is only 32%, after applying our new sampling technique, about 75% of the time, a trace is "live". For small trace caches the utilization of the area is very important and so it can explain why we see a vast improvement in performance due to our technique.

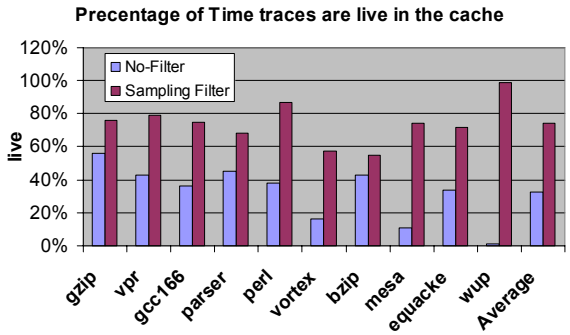

Figure 9. Percentage of time traces are live in the cache.

So far we saw that the sampling technique improves both the trace utilization and the "live" time of traces within the cache. Table 4. shows that the proposed sampling technique also improves the overall utilization of the entire trace cache (TCU). This result has significance of its own. Several works have proposed to use hardware to optimize frequent code on the fly [10][15]. By increasing the TCU, the sampling filter ensures that optimized code will be reused many times prior to its replacement. Therefore, costly hardware optimization can be applied on traces that are inserted to the trace cache because the number of insertions is low and the utilization rate is high. The trace cache utilization rate increased 21.2 times for a Sampling Filter configuration over the regular trace cache (see Table 3.2).

Table 4. TCU of a regular trace cache, a SF system and their ratio.

BENCHMARK	REGULAR TRACE CACHE	SAMPLING FILTER	RATIO
gzip	5.13	160.69	31.3
vpr	2.45	41.65	17.0
gcc166	10.37	191.43	18.5
parser	2.84	51.13	18.0
perlbmk	1.19	36.84	31.0
vortex	0.57	14.4	25.3
bzip	28.1	435	15.5
mesa	0.59	15.02	25.5
equake	1.97	39.77	20.2
wupwise	0.9	162.39	180.4
Average	5	115	21.2

3.3. Power consideration in Sampling Filter

So far we focused on the performance aspects of the Sampling Filter technique. This subsection extends the discussion to power considerations and shows that the sampling technique is also advantageous in terms of power. The main reasons for that are the significant reduction in power that is used to write inefficient traces to the cache, and the better utilization of the trace cache that leads to fewer builds from the instruction cache. In Table 5. the number of writes per 100 committed instructions is presented for a regular trace cache and a Sampling Filter organization. On average, the sampling filter organization has 29 times fewer writes to the cache than a regular trace cache.

Table 5. Number of writes per 100 committed instructions in a regular trace cache, SF system and their ratio.

BENCHMARK	REGULAR TRACE CACHE	SAMPLING FILTER	RATIO
Gzip	2.25	0.08	28.14
vpr	3.47	0.18	18.82
gcc166	0.71	0.04	18.79
parser	3.22	0.18	18.28
perl	0.79	0.03	26.51
vortex	5.76	0.26	22.19
bzip	0.44	0.03	15.73
mesa	5.37	0.24	22.56
equacke	3.58	0.16	22.74
wup	3.48	0.04	96.00
Average	2.91	0.12	28.98

Figure 10. shows the fetch stage energy of three equal area trace configurations: a regular trace cache, the FTC_MTC organization and the Sampling Filter organization as well as the 16KB instruction cache. All the results are normalized so the energy consumption of a regular 8KB instruction cache is always 1. It is clear from the figure that increasing the size of the instruction cache increases the energy consumption while using a small (and therefore power efficient) trace cache reduces the energy consumption. This is true even though our model assumes a parallel access to the trace cache and the instruction cache, because a successful access to the trace cache terminates the build from the instruction cache. Therefore, in comparison to the instruction cache model, using a small trace cache adds cheap accesses to the trace cache and eliminates more expensive accesses to the instruction cache. The regular trace cache reduces the energy consumption by 17% while the FTC_MTC organization reduces it only by 4.8% as this organization involves accessing both the filter and the main trace caches in parallel. On the other hand, the Sampling Filter reduces the fetch stage power by 27% over a regular 8KB instruction cache. This is due to the improvement in the hit rate, which reduces the number of accesses to the L1 instruction cache and to the massive reduction in writes to the cache.

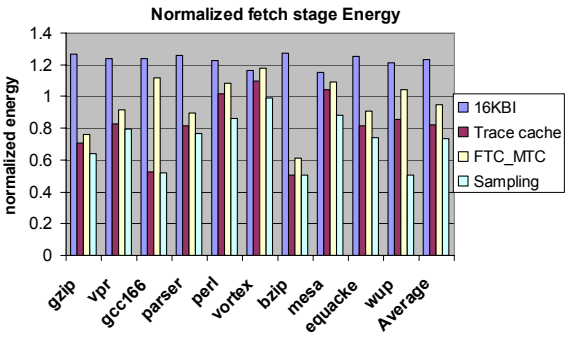

Figure 10. Normalized (I8KB is 1) fetch stage energy of the different fetch engines.

3.4. Trace and Instruction cache decoupling

The purpose of the sampling filter is to reduce the percentage of low utilization rate traces. By reducing the number of writes to the trace cache the sampling filter also accomplishes a reduction in the overall miss rate at the Level 1 caches hierarchy (Trace cache and Instruction cache together). The backing instruction cache is important because it provides the instructions to build a trace upon a trace cache miss. Accessing the Level 2 cache to build traces would reduce the performance because of the L2 longer access time. After the trace is built the code is present in the trace cache as well as in the instruction cache. Thus, the code is duplicated and the Level 1 memory is not used efficiently. If the trace is repeatedly rebuilt then it will continue to be duplicated in both the trace cache and the instruction cache. The ability of the Level 1 backing instruction cache to provide a high percentage of the trace misses is essential for maintaining a high instruction bandwidth. The sampling filter decouples the Trace Cache and the Instruction Cache by prolonging the lifetime of traces in the trace cache. At first, the basic blocks of a trace that was inserted to the trace cache are present in the instruction cache as well. But, those basic blocks are gradually replaced by the LRU replacement policy of the instruction cache, because the trace cache holds and serves them repeatedly over time. Consequently, duplication among the caches is reduced, and the overall instruction supply out of L1 caches is improved. In order to demonstrate the decoupling effect, we conduct a new set of experiments on a system with a small 4KB backing instruction cache. Figure 3.8 shows the instruction cache miss rate for various sampling rates. The miss rate is presented only for benchmarks that have an instruction cache miss rate higher than 0.5%. As the sampling rate decreases the decoupling effect is stronger and so the instruction cache miss rate decreases.

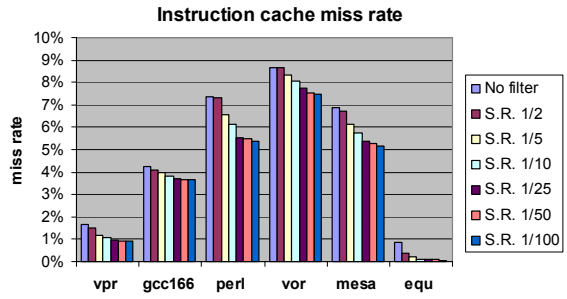

Figure 11. Add/change instruction supply in L1.

The impact on the IPC is presented in Figure 3.9. The Average IPC over all the benchmarks is improved by 12.1% for a sampling rate of 1/100. The sampling filter improves the perlbmk benchmark by 43% over the "regular" trace cache (sampling rate of 1/100) as the Level 1 cache hierarchy is able to supply many more instructions.

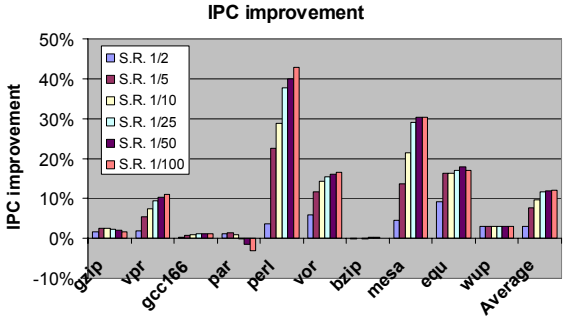

Figure 12. IPC improvent for different sampling rates

4. Combining the sampling filter with the FTC-MTC organization.

The sampling filter is orthogonal to the FTC-MTC principle, hence the two can be combined. By placing the sampling filter in front of the FTC cache (see Figure 13.), the utilization rate of traces in the FTC can be improved. Moreover, by reducing the number of writes to the FTC it can better monitor the behavior of traces. The decision whether to discard the trace or store it in the MTC is taken after a longer period and thus the observation better reflects the nature of the trace. Figure 14. and Figure 15. show the improvement in IPC and the reduction in ED^2 of several filter organizations over a regular trace cache respectively. The combination of the sampling filter with the FTC-MTC organization improves the IPC by 9.5% and the ED^2 by 16.2% while the sampling filter improves the IPC and ED^2 only by 6.7% and by 10%, respectively. The combination of the sampling filter with the FTC-MTC organization outperforms both the sampling filter and the FTC-MTC organization, applied separately.

The Hit rate and Coverage of different trace cache organizations are presented in Figure 4.4 and Figure 4.5 respectively. The combination of the sampling filter with the FTC-MTC organization increases the average hit rate by 17% over a regular trace cache (from 66% to 77.2%) while the sampling filter increases the hit rate only by 9.9% (from 66% to 72.5%). The coverage shows the same tendency, the coverage of the sampling filter is improved by 17.2% (from 56.5% to 66.3%) and the combination of the sampling filter and FTC-MTC organization improves the coverage by 24.2% (from 56.5% to 70.2%) over a regular trace cache.

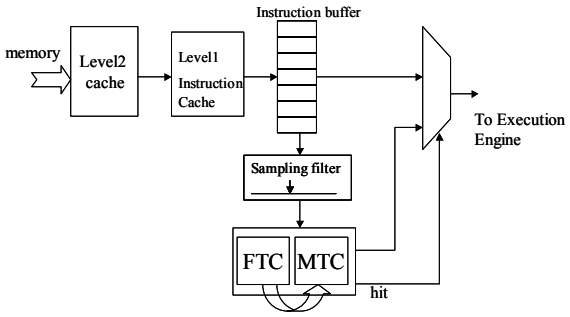

Figure 13. Sampling Filter with FTC-MTC.

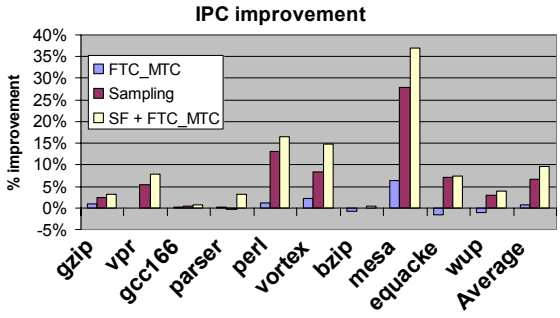

Figure 14. IPC improvement of different filters organization over a regular trace cache.

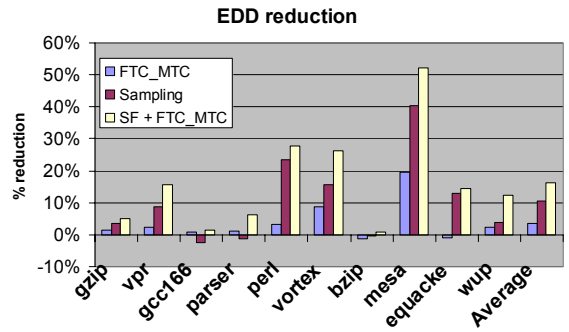

Figure 15. EDD reduction of different filters organization over a regular trace cache.

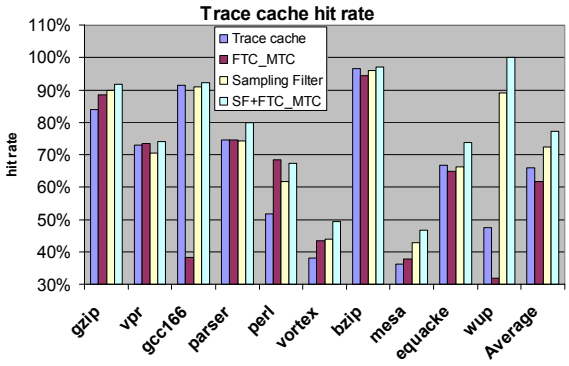

Figure 16. Trace cache hit rate achieved by different organizatoin.

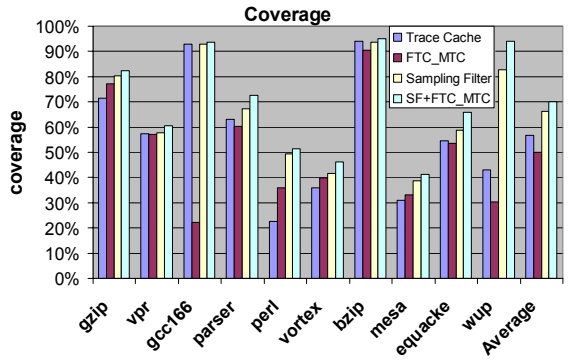

Figure 17. Coverage achieved by different trace cache organizations.

These results indicate that the ability of the FTC-MTC organization to capture the “hot traces” in the MTC is well complemented by the ability of the Sampling Filter to reduce the number of “cold traces” writes. Moreover, as the "hot traces" are better preserved in the cache, the population of traces that is built (and therefore is subject to filtering) is even "colder" than before.

The effect on the live time of traces is presented in Figure 18. The percentage of time traces are live in the FTC-MTC organization is increased to 81% (the sampling filter increases the live time to 75%). This indicates that the small trace cache is now storing mainly "hot traces" and that "cold traces" are efficiently blocked.

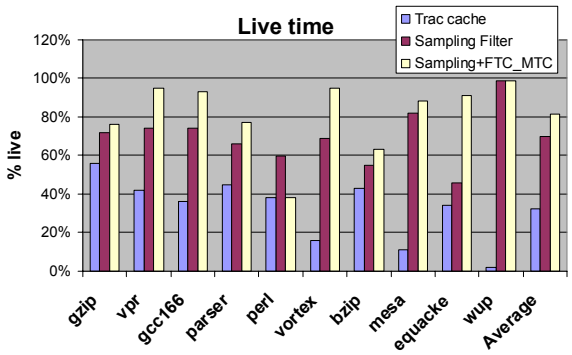

Figure 18. Live time of traces in a regular trace cache, the sampling filter and the sampling filter with the FTC-MTC organization.

5. Discussion and conclusions

In this paper we investigated the impact of filtering on small trace caches and proposed a novel filter: the sampling filter. Small trace caches are efficient in terms of power and access time but suffer from low utilization of the memory space. We showed that a small trace cache combined with a larger backup instruction cache can be very power efficient in comparison to a regular instruction cache because it reduces the number of accesses to the more expensive cache. In order to increase the effectiveness of small trace caches, filtering mechanisms can be applied. The sampling filter is a novel filter that is based on a random sampling approach. Rather than inserting each trace to the trace cache and then monitoring its behavior, the sampling filter reduces the number of writes to the trace cache. It exploits the fact that most writes to the trace cache are of traces that are not used prior their eviction. The traces that are executed many times from the trace cache (and contribute most of the committed instructions) are captured quickly by the sampling filter and maintained in the cache more efficiently by the LRU mechanism. The LRU mechanism benefits from the longer time between replacements as "cold traces" that are entered to the cache can be replaced at the next replacement (as time pass they become least recently used).

This paper showed that the sampling filter improves the trace cache behavior in terms of coverage and hit

rate while the fetch stage power is reduced. The power of the fetch stage is reduced as the number of writes to the trace cache can be dramatically reduced while the number of hits in the trace cache increases. The coverage improvement can be especially beneficial for systems that store instructions in the trace cache after some processing, e.g. the Pentium 4 [6]. From a system perspective, the IPC and ED^2 are improved as well.

The sampling filter also improves the utilization of the Level 1 caches hierarchy (instructions cache and trace cache together) by decoupling the instruction cache and the trace cache.

The combination of the FTC-MTC organization with the sampling filter yields better results than each of the filters alone. This leads us to believe that the sampling filter random selection can be enhanced by a mechanism that retains the "hot traces" in the cache better than the LRU. Future research will focus on implementing such mechanisms, while maintaining the filter power efficiency. We also intend to present an adaptive mechanism to optimize the sampling rate for each program and trace cache size dynamically.

References

[1] Bryan Black, Bohuslav Rychlik, and John Paul Shen. "The block-based trace cache". Proceedings of the 26th Annual Intl. Symposium on Computer Architecture, May 1999.

[2] David Brooks, Pradip Bose, Stanley E. Schuster, Hans Jacobson, Prabhakar N. Kudva, Alper Buyuktosunoglu, John-David Wellman, Victor Zyuban, Manish Gupta, and Peter W. Cook, "Power-Aware Microarchitecture: Design and Modeling Challenges for Next-Generation Microprocessors," IEEE Micro, November/December, 2000.

[3] David Brooks and Vivek Tiwari and Margaret Martonosi, " Wattch: a framework for architectural-level power analysis and optimizations", in ISCA 2000 pages 83-94.

[4] Douglas C. Burger and Todd M. Austin. *The SimpleScalar Tool Set, Version 2.0*. University of Wisconsin, Madison Tech. Report. June 1997.

[5] J. Henning. SPEC CPU2000: Measuring CPU Performance in the New Millennium. *IEEE Computer*, pp. 28-35, 2000.

[6] G. Hinton et al., "The microarchitecture of the Pentium 4 processor," in Intel TechnologyJournal, 2001

[7] Q. Jacobson, E. Rotenberg, J. E. Smith, " Path- Based Next Trace Prediction," in *Proceedings of the 30th International Symposium on Microarchitecture*, pp. 14-23, December 1997.

[8] S. Kaxiras, Z. Hu and M. Martonosi, "Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power", in Proc. of the Int'l Symposium on Computer Architecture, 2001, pp.240--251.

[9] O. Kosyakovsky, A. Mendelson and A. Kolodny, "The Use of Profile-based Trace Classification for Improving the Power and Performance of Trace Cache Systems", in *4th Workshop on Feedback-Directed and Dynamic Optimization*, Dec. 2001.

[10] S. Patel and S. Lumetta, "rePlay: A Hardware Framework for Dynamic Optimization", in *IEEE Trans. on Computers*, 50(6), pp 590-608, June 2001

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

[12] M. Postiff, G. Tyson and T. Mudge, "Performance Limits of Trace Caches", in *Journal of Instruction-Level Parallelism,* vol. 1, Oct. 1999.

[13] A. Ramirez, J.L. Larriba-Pey, and M. Valero, "Trace Cache Redundancy: Red and Blue Traces**",** in *Proc. 6th Intern. Symp. on High-Performance Computer Architecture*, pp. 325-333, 2000.

[14] R. Rosner, A. Mendelson, and R. Ronen, "Filtering Techniques to Improve Trace-Cache Efficiency", in Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, pages 37-48, September 2001.

[15] Roni Rosner, Yoav Almog, Micha Moffie, Naftali Schwartz, Avi Mendelson: "PARROT: Power Awareness Through Selective Dynamically Optimized Traces", in PACS 2003: 196-214

[16] E. Rotenberg, S. Bennett and J. Smith, "A Trace Cache Microarchitecture and Evaluation", in *IEEE Trans. on Computers*, 48(2), pp 111–120, Feb. 1999

[17] E. Rotenberg, S. Bennett, and J. Smith, "Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching", 29th International Symposium on Microarchitecture (MICRO-29), Dec. 1996

[18] S.J.E. Wilton and N.P. Jouppi. "CACTI: An enhanced cache access and cycle time model." IEEE Journal of Solid-State Circuits, Vol. 31(5):677-688, May 1996.