Trace Cache Sampling Filter

MICHAEL BEHAR
Intel Corporation, Israel

AVI MENDELSON
Intel Corporation, Israel

AND

AVINOAM KOLODNY
Technion – Israel Institute of Technology

A simple mechanism to increase the utilization of a small trace cache, and simultaneously reduce its power consumption, is presented in this paper. The mechanism uses selective storage of traces (filtering) that is based on a new concept in computer architecture: random sampling. The sampling filter exploits the "hot/cold trace" principle, which divides the population of traces into two groups. The first group contains "hot traces" that are executed many times from the trace cache and contribute the majority of committed instructions. The second group contains "cold traces" that are rarely executed, but are responsible for the majority of writes to an unfiltered cache. The sampling filter selects traces without any prior knowledge of their quality. However, as most writes to the cache are of "cold traces" it statistically filters out those traces, reducing cache turnover and eventually leading to higher quality traces residing in the cache. In contrast with previously proposed filters, which perform bookkeeping for all traces in the program, the sampling filter can be implemented with minimal hardware. Results show that the sampling filter can increase the number of hits per build (utilization) by a factor of 38, reduce the miss rate by 20% and improve the performance-power efficiency by 15%. Further improvements can be obtained by extensions to the basic sampling filter: allowing "hot traces" to bypass the sampling filter, combining of sampling together with previously proposed filters, and changing the replacement policy in the trace cache. Those techniques combined with the sampling filter can reduce the miss rate of the trace cache by up to 40%. Although the effectiveness of the sampling filter is demonstrated for a trace cache, the sampling principle is applicable to other micro-architectural structures with similar access patterns.

Categories and Subject Descriptors: C.1.1 [**Processor Architectures**]: Single Data Stream Architectures; B.3.3 [**Memory Structures**]: Performance Analysis and Design Aids
General Terms: Performance; Measurement
Additional Key Words and Phrases: Trace cache, sampling filter, power dissipation, cache utilization

---

1. INTRODUCTION

High bandwidth instruction supply under power and thermal constraints is a major challenge for modern processors. New architectures, which aim to increase performance by exploiting more instruction level parallelism require the processor's front-end to supply more decoded instructions at each cycle. The trace cache is an effective high bandwidth and low latency instruction supply mechanism [Peleg and Weiser 1995; Rotenberg et al. 1996; Patel et al. 1997; Rotenberg et al. 1999]. By storing instructions in their dynamic order (traces) rather than in their static compiled order the trace cache allows fetching of several basic blocks in one access. Furthermore, the trace cache is

power efficient as it can store instructions in a decoded form [Solomon et al. 2001]. For example, the Pentium 4 [Hinton et al 2001] stores decoded instructions (uops) in the trace cache, thus reducing the number of expensive decodes.

Unfortunately, the trace cache has several drawbacks that lead to low utilization of cache memory space [Postiff et al. 1999; Vandierendonck et a. 2002]. The storage of traces creates some redundancies: the trace might not contain the maximum number of instructions (Fragmentation), the same basic blocks might appear in different traces (Duplication), and the trace has only one entrance point so instructions inside the trace are inaccessible (Indexability). Therefore, a large trace cache is needed in order to provide a reasonable hit rate.

Adding a regular instruction cache in the L1 hierarchy, which eliminates the need to construct the traces from the Level 2 instruction cache, considerably reduces the penalty of a miss in the trace cache. Such a system is described in Figure 1. The traces that are not found in the trace cache are built from the backing regular instruction cache. Nevertheless, since every trace is constructed from the instruction cache, it is present in both caches. This introduces a new form of duplication which reduces the utilization of the L1 cache hierarchy [Ramirez et al. 1999].

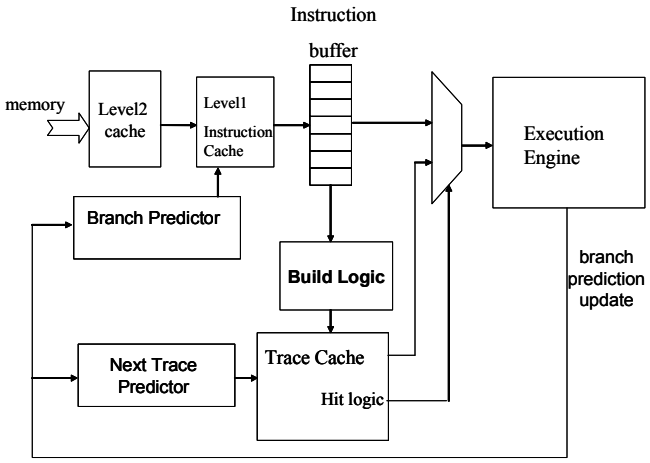

Fig. 1 A trace cache syetm with a backing instruction cache. The Level1 cache hierarchy consists of a trace cache and an instruction cache.If a trace is not found in the trace cache it can be constructed from the L1 instruction cache, reducing the miss penalty.

One way to increase the hit rate is to increase the trace cache size. However, this is too expensive in terms of power and access time. Alternatively, a small trace cache would be effective if only "useful" traces were stored in it. The trace population can be divided

into two groups: the "hot traces" and "cold traces" [Rosner et al. 2001]. "Hot traces" are a small group of traces that are very frequently used, and are responsible for the majority of the instructions executed, and therefore they often reside in the cache. The typical number of unique "Hot traces" is small. It follows the "80/20 principle": a small fraction of the static code is responsible for most of the dynamic code. "Cold traces" are accountable only for the minority of instructions executed, but are numerous (unique traces). Since "cold traces" are rarely executed, the replacement algorithm typically eliminates them from the cache before they are accessed again. It was reported that the majority of builds are of "cold traces" while the majority of the executed instructions come from "hot traces". The target of any good cache replacement algorithm is to cause the cache population to consist mainly of "hot traces", but the entrance of "cold traces" into the cache might cause the eviction of "hot traces". Taking into account that "cold traces" are rarely accessed again prior to their own replacement, this reduces the hit rate of the cache and increases the number of builds. Furthermore, the build of a "cold trace" is power inefficient as the energy invested in building and writing such a trace is a waste.

Selectively admitting traces to the cache (filtering) based on the "hot/cold" principle reduces the number of builds and improves the utilization of the cache. The filter identifies the "cold traces" and prevents them from entering the cache, thus enabling the "hot traces" to better reside in the cache. The filter also reduces the number of builds and replacements in the cache. Previous works have proposed to identify the "hot/cold traces" using profiling or costly hardware [Kosyakovsky et al. 2001; Rosner et al. 2001]. Our objective is to implement a simple but effective filter to improve the performance and power of a small trace cache, whose size is limited by area, power and access time constraints. We concentrate on a model in which only the "hot traces" reside in the cache, exploiting its ability to supply many useful instruction at each fetch, while the "cold traces" are stored in a larger instruction cache, which stores the instructions more efficiently. This combination is attractive for several reasons: usually, the number of "hot traces" is limited and hence they can fit well in a small trace cache. Most of the traces will be executed from the trace cache as the "hot traces" cover the majority of the dynamic code. By holding the "cold traces" in the instruction cache the trace cache is not polluted, the "cold traces" which are numerous are stored in a more compact manner and the duplication between the caches is reduced. In order to accomplish this goal we propose a novel hardware method of filtering, based on a statistical random selection of traces, that is able to accumulate the "hot code" in the trace cache and the "cold code" in the instruction cache. The filter samples the stream of traces which are candidates to be

built, and periodically chooses (samples) a trace to build and to insert into the trace cache. The usage of statistical sampling eliminates the need for complex hardware to perform the "hot/cold" bookkeeping. The results of applying the sampling filter indicate that statistical filtering methods can be efficient in reducing the build rate and the duplication between the instruction and trace caches, improving the trace cache hit rate and the system performance.

The rest of this paper is organized as follows: The following section surveys related work in filtering methods. Next we investigate the "hot/cold" principle by extracting information from a detailed trace cache system model to characterize the behavior of traces in the system. This is followed by a description of the sampling filter, a discussion of why and how it works, and experimental results. We then present several enhancements to the basic sampling filter and end with conclusions and discussion.

2. PREVIOUS AND RELATED WORK

Using a cache hierarchy is a common method to mask the large latency to the memory. The L1 cache is most likely to be on the critical path [Patterson and Hennessy 1994] and therefore cache size is limited by access time. Furthermore the L1 cache is traditionally designed for performance rather than power and therefore is costly in both dynamic and leakage power.

Sahuquillo et al. [1999] propose to split the dada cache to a small filter cache and a much larger main cache. The objective is to store the most heavily referenced blocks in the filter cache. Upon a miss in the L1 the cache controller compares the usage counter of the missing block in L2 and the conflict block in the filter cache. The block with larger counter will remain in the filter cache while the other will pass to the main cache.

Trace cache filtering has been proposed as a way of increasing the performance of a trace based processor. Ramirez et al. [2000] propose to store in the trace cache only traces that contain taken branches, as those traces require several accesses to the instruction cache. The paper also proposed to use this filtering technique combined with a software trace cache [Ramírez et al. 1999], which further reduces the overlap between the instruction and trace cache. A software trace cache uses a compiler to optimize the instruction layout in memory so sequentially executed basic blocks are placed in consecutive memory space. The combined technique aims to reduce the redundancy between the instruction and trace caches. Although those techniques increase the fetch bandwidth and overall performance, they reduce the percentage of instructions originating from the trace cache and therefore they are less attractive for systems that

store decoded instructions in the trace cache. Moreover, systems that have a penalty for a miss in the trace cache (*e.g.* a system that accesses the instruction cache only if the trace cache misses, for power considerations) suffer a severe reduction in performance. Filtering based on the usability of traces was proposed as a means to reduce the build rate as well as to increase the hit rate of trace caches [Kosyakovsky et al. 2001; Rosner et al. 2001].

Profiling could be used to identify the "hot traces" [Kosyakovsky et al. 2001]. Profiling has several drawbacks: the ISA must change in order to supply the hardware with hints which trace to store, the behavior of traces may change during the lifetime of a program or across different inputs, and different inputs may cause different traces to be created. Alternatively, a hardware filter can dynamically identify hot traces, and can adapt itself to different inputs and phases of the program. Such a hardware approach is the FTC-MTC organization [Rosner et al. 2001] that split the trace cache into two caches: The Filter Trace Cache (FTC) and the Main Trace Cache (MTC). All traces are first inserted to the FTC and counters monitor their behavior. Highly used traces are moved to the MTC upon replacement. In this method the FTC serves as a trainer that enables the storage of "hot traces" in the MTC. However, each access to the trace cache requires accessing both the FTC and the MTC, which leads to higher power consumption. Moreover, only part of the trace cache (the MTC) is used for storing the "hot traces".

Redundancies that are due to the way traces are stored can be eliminated or reduced by storing the traces in a different organization, e.g., the blocked-based trace cache [Black et al. 1999]. Such techniques might add complexity and are beyond the scope of this paper.

3. CHARACTERIZATION OF TRACE CACHE PERFORMANCE

Processor systems with instruction and trace caches structured as in Fig. 1 have been characterized using an augmented version of the SimpleScalar simulator [Burger et al. 1997]. The baseline machine parameters are listed in Table 1. The machine is 8-way wide. We use 10 benchmarks from the Spec2000 benchmark suite [Henning 2000]. The benchmarks and inputs are listed in Table 2. The first 500 million instructions were skipped and then the benchmark was simulated for one billion instructions. Perlbmk benchmark was simulated until it ended after 880 million instructions.

3.1 Simulation environment

The SimpleScalar simulator was augmented by a trace cache and a next-trace predictor [Jacobson et al. 1997]. The power estimates were obtained using Wattch [Brooks et al. 2000] and Cacti [Wilton and Jouppi 1996]. The power model of all the new structures and algorithms was integrated in Wattch using Cacti following the way the original structures were modeled. The Power values that are presented are based on Wattch clock-gating style, which scales the power according to the usage, e.g., if only 2 ports out of 6 are activated during a cycle then the power dissipation is scaled by a third. We use the Energy Delay square product metric (ED^2) [Brooks et al. 2000] to evaluate the energy performance tradeoffs as it is voltage independent in first approximation.

Traces are constructed in the front-end and thus wrong speculative traces are allowed. Each trace can contain 16 instructions and 3 direct branches (not including the last instruction). All traces end at basic block boundaries in order to reduce the number of unique traces and duplications. Indirect branches or jumps, procedure calls, return instructions and interrupts terminate the construction of a trace. Traces beginning with the same start address but with different branch outcomes can coexist in the trace cache. It is the responsibility of the next trace predictor to supply the correct trace ID, which consists of the start address and the branch direction vector.

We model concurrent access between the instruction cache and the trace cache (CTC). If there is a hit in the trace cache, the build process from the instruction cache is terminated. As the access to the caches is concurrent, the trace cache miss penalty is minimal and therefore a filtering technique that improves the trace cache hit rate has only a moderate impact on the IPC in our model. However, as we study mainly the usage of small trace caches, the penalty associated with misses would be too high in sequential mode. Moreover, as the trace cache is relatively small the added power of accessing it in parallel to the larger instruction cache is small.

3.2 Comparison of systems with different combinations and sizes of caches

The performance and power-performance efficiency of trace cache systems with different L1 cache sizes are studied in this section. The IPC speedup of several trace cache systems with an instruction cache over a 4KB trace cache system without a L1 instruction cache is presented in Figure 2 (average of all benchmarks). The access latency assumed in caches up to 16KB is two cycles, the access latency in the 32KB cache is 3 cycles and the access latency in the 64KB cache is 4 cycles. The configuration denoted "instruction cache" is the only configuration without a trace cache, while the "Trace Cache", "Trace Cache +

8KB" and "Trace Cache + 16KB" are configurations of a trace cache system with a parallel instruction cache (in the L1 hierarchy) of size 8KB and 16KB correspondingly.

Table I. 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	16KB 4-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) 8KB(128 traces) 4-way, LRU, 2-cycle mlatency
L2 Unified cache	1MB 8-ways, 64B blocks, LRU, 12-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

Table II. 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

Several interesting conclusions can be drawn from comparison of the different configurations. The IPC of a trace cache system without a parallel instruction cache (Trace cache) is smaller or equal to the IPC of a system with only an instruction cache for the 4KB, 8KB and 16KB caches while for the 32KB and 64KB caches the performance of the trace cache system is significantly better, e.g., the speedup of the 64KB trace cache system over the 64KB instruction cache system is 1.4. Although the trace cache has a higher bandwidth than the instruction cache, it has a higher miss rate and therefore the instruction cache system performance for the 4KB, 8KB and 16KB caches is better than the performance of the trace cache system. However, once the access time to the cache is increased, as is the case for the 32KB (3 cycles) and the 64KB (4 cycles) caches, the trace cache performance is better than the performance of the instruction cache. The trace cache is able to supply more useful instructions in each fetch than the instruction cache and therefore it is less sensitive to the access time increase than the instruction cache.

Another important observation is that a combination of the instruction and trace caches (Trace Cache + 8KB/16KB Instruction Cache,) delivers more performance than the trace cache or the instruction cache alone, *e.g.*, the 16KB instruction cache yields a speedup of 1.6 while the 8KB trace cache combined with the 8KB instruction cache yield a speedup of 1.86. Therefore, combining the instruction cache with the trace cache is beneficial, as the trace cache can supply more bandwidth and the parallel instruction cache considerably reduces the trace cache miss penalty.

Although larger caches have higher hit rates, increasing cache size to a point where the access time increases is not worth while as the average number of instructions supplied in each cycle is reduced. The trace cache systems "knee" point is the 8KB cache as the performance gain from doubling the trace cache to 16KB is low, e.g., doubling the size of the trace cache from 8KB to 16KB in the Trace Cache +16KB Instruction Cache configuration adds only 1.5% to the IPC.

The performance difference between the trace cache system with a 16KB instruction cache and the one with 8KB instruction cache is 7.3%, 4.3% and 2% for a 4KB, 8KB and 16KB trace caches respectively. The performance difference is reduced as the trace cache size is increased because the hit rate in the trace cache increases as well and therefore the instruction cache is required to supply fewer instructions.

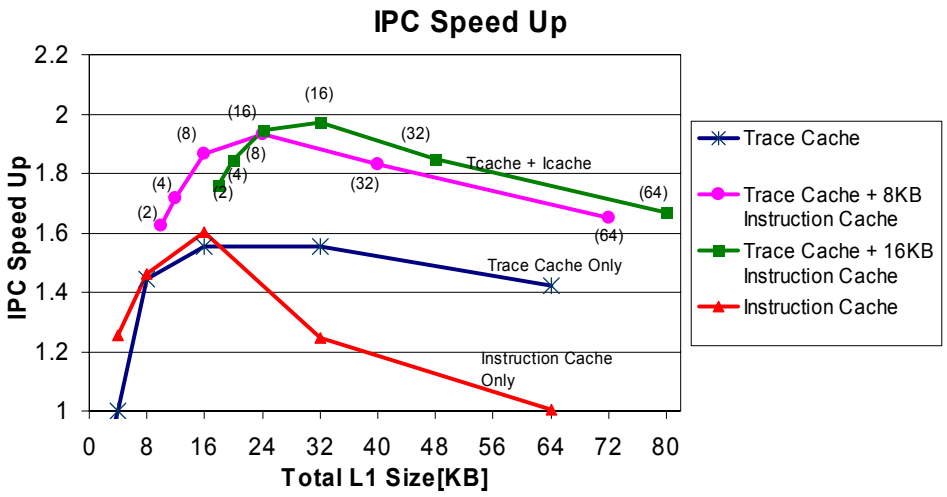

Fig. 2. IPC speedup of several fetch engines over a 4KB trace cache. The total size of the instruction cache and trace cache is presented on the x axis and the trace cache size is noted in parentheses.

The performance-power efficiency is presented in Figure 3. by the ED^2 metric. The systems with combined trace cache and instruction cache are more performance-power efficient than the trace cache and the instruction cache systems. The best performance-power efficiency point is achieved by the combination of an 8KB trace cache and a 16KB instruction cache. Notice that although the best performance is achieved by the combination of a 16KB instruction cache and a 16KB trace cache, this is not the optimal point as the performance gained by the larger cache is shadowed by the increase in power dissipation.

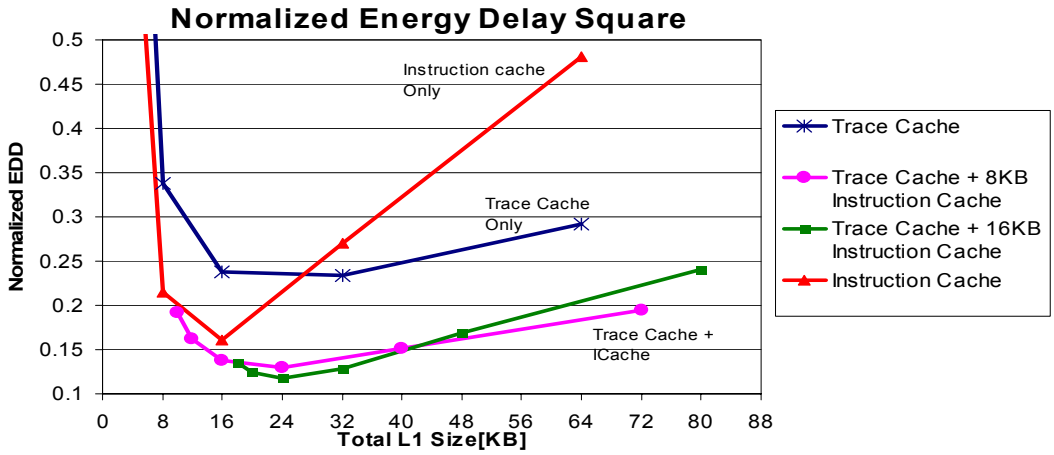

Fig. 3. ED^2 of several fetch engines normalized to a 4KB trace cache. The total size of the instruction cache and trace cache is presented on the x.

Both the IPC and the ED^2 results indicate that the combination of a trace cache with an instruction cache is more beneficial than using only one of the caches. Furthermore, the speedup gained by larger caches is diminished by the larger access time and the ED^2 is increased by the larger power dissipation.

3.3 The "Hot/Cold" principle

The results in this section and the next are consistent with previous works [Rosner et al. 2001]. They are described in order to establish the reasoning for using a statistical sampling filter. We use program profiling in order to measure the number of unique traces created during the program run and the percentage of dynamic code each unique trace is accountable for. The measurements show that many unique traces are created but only few traces are accountable for a high percentage of the dynamic code. This observation is presented in Figure 4., which shows how much of the dynamic code (an average over all benchmarks) can be attributed to the most frequent traces, *i.e.*, the percentage of dynamic code is presented on the y axis and the number of traces is on the x axis. Few frequent traces are accountable for the majority of instructions executed, *e.g.*, the 64 most frequent traces are accountable for 80% of the dynamic code executed. On the other hand, the majority of unique traces are accountable only for the minority of executed instructions.

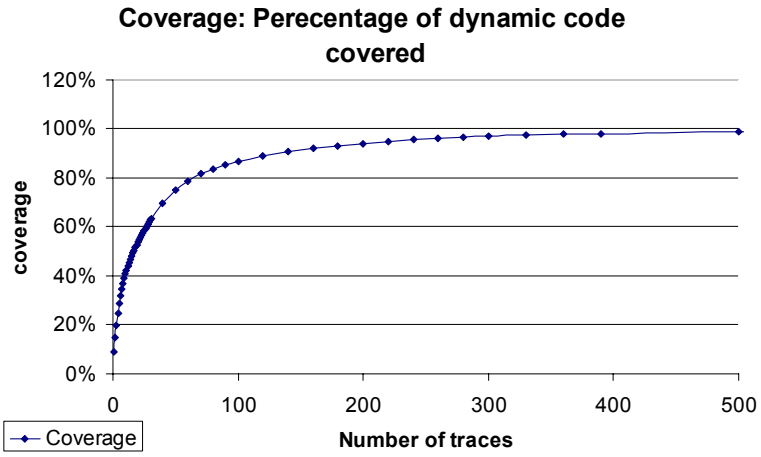

Fig. 4. Percentage of dynamic code covered by the most frequent traces. This is a cumulative Pareto graph, *i.e.*, the traces are ordered by their contribution to the dynamic code and the graph accumulates the contribution.

3.4 The "Hot/cold" principle effect on trace caches

The previous section presented a profile-based characterization of traces to "hot traces" and "cold traces". In this section the impact of the "hot/cold" nature of traces on the trace cache and on the behavior of traces in the cache is presented.

The "live" time of traces, the percentage of time traces are useful, is an important parameter that reveals how well the cache space is utilized. The "live" time [Kaxiras et al. 2001; Rosner et al. 2001] of a trace is the time between the insertion of the trace and the last access to it, prior to its replacement. The "decay" time is the time between the last access to a trace and its replacement. The percentage of time traces are live in different cache sizes is presented in Figure 5. On average, traces are live only 32%, 39% and 49% of the time in a 32(2KB), 64(4KB) and 128(8KB) traces trace caches respectively. This indicates that most of the time traces reside useless in the cache and so the cache space is poorly utilized.

The previous data indicate that most of the time traces are in their decay stage. However, the "hot/cold" principle indicates that a small set of traces is very frequent and therefore one would expect that the replacement policy would enable the cache to retain the "hot traces". If that were the case then the residing traces in the cache would be frequently used traces ("hot traces") and the "live" time should have been much larger than the "decay" time. Additional evidence that this isn't the case is the number of times a trace is used after its insertion and before its eviction from the cache. In Table III. the average number of hits per insertion is presented for a 32(2KB), 64(4KB) and 128(8KB) traces trace caches. For a 32-traces trace cache, 6 out of 10 benchmarks have less than 2 hits per insertion. The number of hits per write is improved in the 64 and 128 traces trace caches but still remains extremely low for some benchmarks, *e.g.*, vortex and mesa.

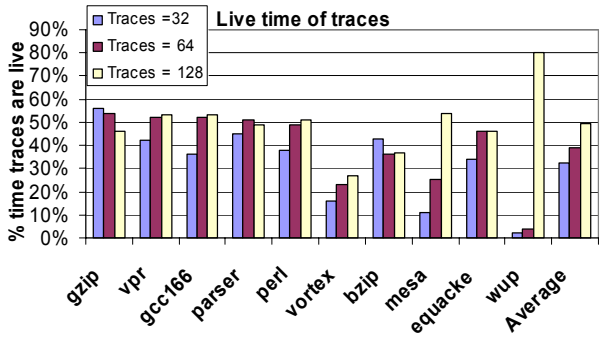

Fig. 5. Percentage of time traces are live in the trace cache. The low percentage of time traces are live indicates that the cache is poorly utilized as traces are useful only in their live period.

Table III. Hits per insertion for a 32(2KB), 64(4KB) and 128(8KB) traces trace caches.

Benchmark	32-traces (2KB)	64-traces (4KB)	128traces (8KB)
164.gzip	5.23	34.93	289.13
175.vpr	2.46	5.51	20.3
176.gcc	10.36	14.55	23.9
197.parser	2.85	11.92	25.85
253.perlbmk	1.19	3.41	17.27

Moreover, the energy invested in building and writing the trace to the cache is wasted. From a power perspective, building traces is an investment and the trace should be used many times in order to return it. Clearly, if most of the writes are of traces that are never used, it poses a problem. The extremely high percentage of builds that result in no hit or less than two hits strongly indicates that most of the builds are unnecessary, resulting in more harm than benefit.

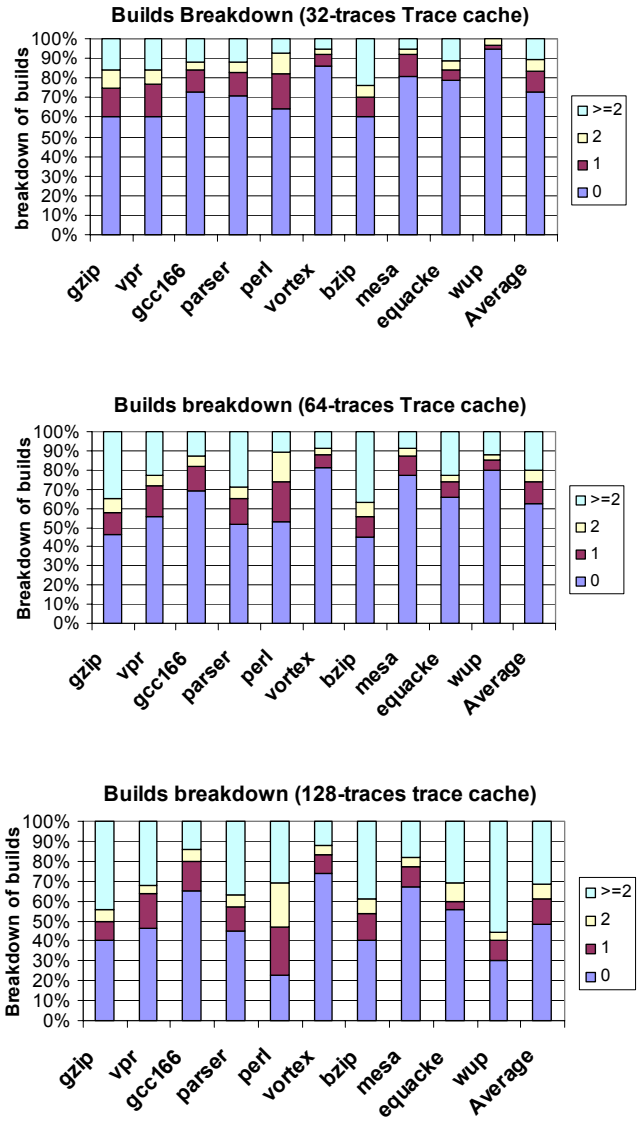

Fig. 6. Builds breakdown by the number hits generated by each build.

The previous section indicated that a small set of "hot traces" is accountable for the majority of the instruction executed while a large set of traces "cold traces" is responsible for the minority of executed instructions. This section showed that most of the builds are of "cold traces", which enter the cache, do not contribute to the hit rate and cause low utilization of the space. Therefore, it would be beneficial to use an entrance filter that would prevent "cold traces" from entering the cache, and would ensure that only "hot traces" reside inside the cache. In the next section a novel sampling filter, which takes advantage of the properties of the "hot/cold" principle, is presented.

4. THE SAMPLING FILTER

4.1 The structure

The sampling filter is based on a new concept in computer architecture. It is an easy-to-implement mechanism that aims to filter "cold traces" out of the trace cache and to maximize the presence of "hot traces" in the cache. This is done by using a statistical sampling approach, which is based on the observation that most of the builds are of "cold traces".

The structure of a trace cache system containing a sampling filter is presented in Figure 7. Instructions can be supplied either from the instruction cache or the trace cache depending on whether there is a hit in the trace cache. If there is a miss in the trace cache then a new trace is built. Normally, in a non-filtered system every trace that is built is inserted into the cache. In contrast with this policy, the sampling filter allows only a fraction of the potential traces to be built and enter the cache. The sampling filter selects traces to be inserted into the cache on a periodic basis. Traces that are not sampled (selected) are discarded. Notice that unlike other filters, which keep track of all the traces in the system and try to decide whether to insert the trace or discard it according to its previous behavior, the sampling filter selects traces with no prior knowledge but rather randomly. This statistical approach of selecting traces doesn't preclude any other filtering method that could be applied to the sampled traces.

The sampling rate (SR) is the rate at which traces are sampled and inserted into the cache, *e.g.*, a sampling rate of 1/10 means that only one out of ten potential traces is inserted into the trace cache. The filtering algorithm can be tuned with different sampling rates achieving different design goals: optimum trace cache hit rate, minimum build rate or maximum performance. A more detailed discussion of the sampling rate will be presented in subsection 4. The implementation of such a filter requires simple hardware

and does not require to access and maintain other structures (for bookkeeping of past behavior information).

In order to exhibit the effectiveness of the new proposed method we present in the next subsection the impact of the sampling filter on trace miss rate, IPC and ED^2. Then we discuss why the sampling filter is effective, how the sampling rate affects the cache and system behavior, and the limitations of the filter.

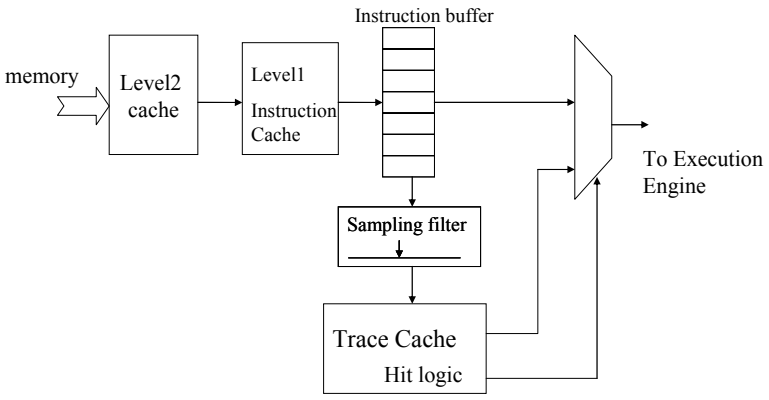

Fig. 7. The front-end of a system with a sampling filter. The sampling filter allows only sampled traces to be constructed and inserted into the cache.

4.2 Results

A trace cache system with a sampling filter (SF) is evaluated versus the baseline machine (No Filter) and versus a system with a FTC-MTC organization (described in section 2). All systems have the same configuration and total cache sizes, besides the fact that the FTC-MTC splits the trace cache into two caches: Main trace cache (MTC) and Filter Trace Cache (FTC). The sampling rate is fixed for all benchmarks and is 1/20 for the 2KB and 4KB caches. The IPC for a 2KB (32 traces) and 4KB (64 traces) is presented in Figure 8. The average IPC is improved by 6.4% and 7% for the 2KB and 4KB trace caches configurations with sampling filter (SF) over the baseline machines (No Filter) respectively. For the 2KB trace cache configuration only two benchmarks showed a reduction of performance: bzip by 0.04% and parser by 0.16%, while benchmarks like mesa and Perlbmk showed 27.7% and 14.75% improvement, respectively. For the 4KB configuration only the parser benchmark showed a reduction in performance of 1.78%. The FTC_MTC organization actually reduces the average performance of all configurations by 3.15% and 1.6% for a 2KB and 4KB trace caches, respectively.

The miss rate of the different system configurations for a 2KB and 4KB trace caches is presented in Figure 9. The average miss rate is reduced by 19% and 24.2% by the sampling filter over the baseline machine for a 2KB and 4KB respectively. For all trace cache sizes, five out ten benchmarks show a decrease in the miss rate using the sampling filter. The FTC_MTC organization is able to reduce the miss rate of five benchmarks out of ten in a 2KB trace cache, and of eight benchmarks in a 4KB trace cache. For some benchmarks the FTC_MTC organization causes a severe increase in miss rate, *e.g.*, the miss rate of a 4KB trace cache under Equake benchmark is increased from 15% to 24%.

The performance-power efficiency (ED^2) is shown in Figure 10. The sampling filter reduces the ED^2 of a 2KB trace cache system by 17.5%, while the ED^2 of the FTC_MTC machine is increased by 5.9% over the baseline machine. The 4KB configuration using the sampling filter shows an ED^2 reduction of 15% over the baseline machine.

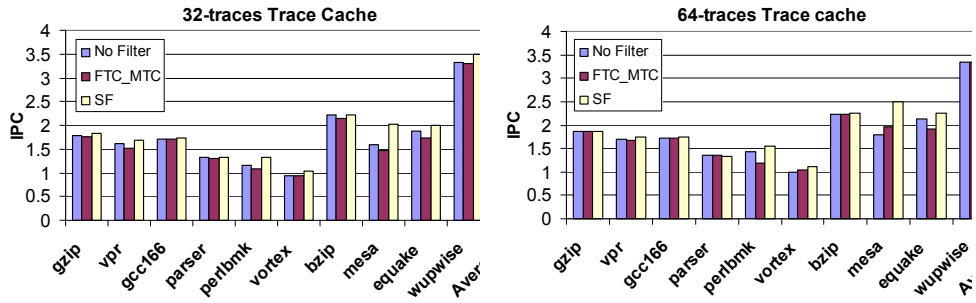

Fig. 8. Performance (IPC) of a 32-traces (left figure), 64-traces (right figure).

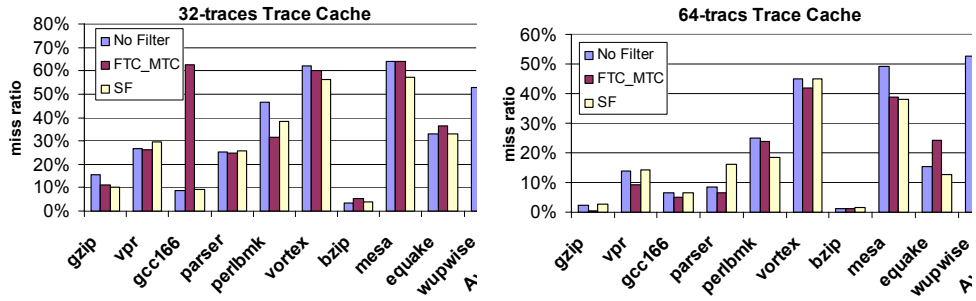

Fig. 9. Miss ratio of a 32-traces (left figure), 64-traces (right figure).

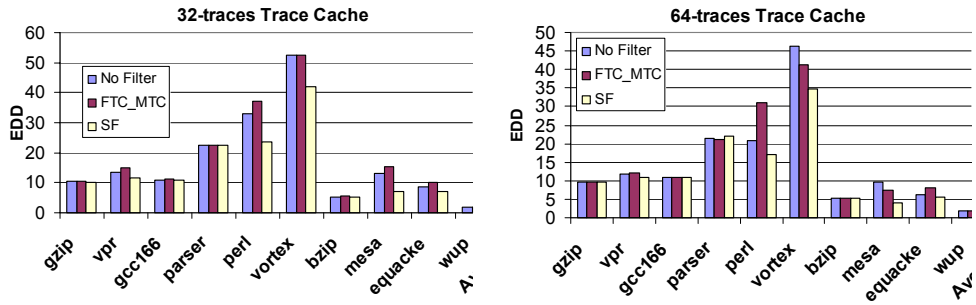

Fig. 10. Energy delay square product of a 32-traces (left figure) and 64-traces (right figure).

The IPC speedup of the sampling filter system with an 8KB instruction cache and trace caches ranging from 2KB to 16KB over the baseline machine with a 2KB trace cache and a 8KB instruction cache is presented in Figure 11 (denoted "Sampling Filter + Trace Cache + 8KB"). The speedups of the non-filtered machines with an instruction cache of 8KB and 16KB are presented as well. Although the sampling filter mainly improves the IPC of the 2KB and 4KB trace caches, the performance of the 8KB and 16KB trace cache systems are improved as well. The system with a 2KB trace cache and a sampling filter has better speedup than a 4KB non-filtered trace cache system with the same instruction cache size. The systems with 4KB and 8KB trace caches with a sampling filter have only 1.45% and 0.46% less speedup than the systems with a double sized non-filtered trace cache. Furthermore, the system with a sampling filter and 16KB trace cache has a better speedup than the non-filtered system with the same trace cache size but with a larger instruction cache (16KB). Thus, the sampling filter can outperform larger non-filtered trace cache systems.

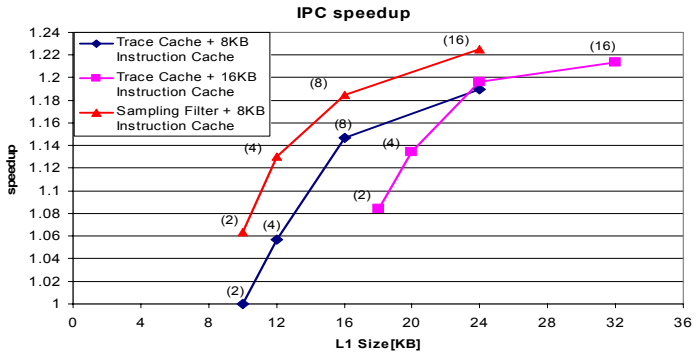

Fig. 11. IPC speedup of trace cache systems over the baseline system (which has a 2KB trace cache and a 8KB instruction cache). The x-axis shows the total size of the Level1 caches and the trace cache size is showed in parentheses.

Another important indication of the success of the sampling filter is the amount of hits provided by each build. This is an important parameter as a trace should be used many times in order to justify the power investment of the build and write process. The ratio between the number of hits per write of the machine with a sampling filter and the baseline machine is listed in Table 4. For 2KB and 4KB caches the average hit per write ratio is improved by a factor of 38 and 36 correspondingly.

Table IV. The hit per write ratio between the filtered and non filtered systems.

Benchmark	2KB	4KB
164.gzip	30.30916	16.33542
175.vpr	16.83468	17.44485
176.gcc	18.41957	18.43172
197.parser	17.90175	8.14527
253.perlbmk	29.31746	30.86217
255.vortex	25.26316	19.92793
256.bzip	15.46405	13.93723
177.mesa	25.42373	30.93333
183.equake	20.70202	28.15018
168.wupwise	180.4333	180.4333
Average	38.00689	36.46014

4.3 Why it Works

The main reason the sampling filter works well is that "cold traces" are rebuilt again and again in an unfiltered cache, with only little chance of being reused. By selecting fewer traces to be inserted to the cache, the number of "cold traces" that enter the cache and pollute it is reduced. Therefore, "hot traces" that reside in the cache are less trashed. As the sampling filter "observes" only the traces that are built and not the traces that are executed out of the trace cache, it now observes a "colder" stream of traces. Therefore, the majority of traces the sampling filter encounters are traces that would only pollute it if inserted as they would have little chance to be accessed before being replaced. Therefore, even a random selection that dilutes this population is beneficial. The sampling process is random and has no prior knowledge of the quality of the trace ("hot/cold"). Nevertheless, the key point is the percentage of "hot/cold" traces in the stream of traces the filter samples. At the beginning of a program run or at working set changes, most of the traces the sampling filter encounters are "hot traces" and they are sampled fairly quickly into the

cache. On the other hand, at the steady state, most of the "hot traces" are executed directly from the trace cache and the majority of traces the sampling filter encounters are "cold traces".

The reduction in the number of writes to the cache also aids the LRU replacement mechanism. If the replacement rate is high (as is the case without filtering) then a cold trace that is inserted to the cache in the MRU position might not become the least recently used by the next replacement. This means that not only a "cold trace" has entered the cache and might have caused the eviction of a useful trace (causing a future miss), but it may not be the first trace to be replaced because of the excessive replacement rate. The sampling filter increases the time between replacements and therefore increases the chance that "hotter" traces in the set will be accessed prior to the next replacement and thus the "cold trace" will become least recently used and thus will be replaced.

The breakdown of builds by the number of hits each build provides is presented in Figure 12 for systems with a filter. In comparison with the non filtered system (Figure 6), the percentage of useless builds (zero hits) is reduced from 72% to 25.7% for a 2KB trace cache and from 62% to 22.8% for a 4KB trace cache. The percentage of builds that result in more than two hits is considerably higher in the sampling filter organization than in the baseline machine.

As indicated in section 3, the percentage of time traces are "live" is low in the baseline machine. The sampling filter increases the presence of "hot traces" in the cache and therefore it is expected that the "live" time of traces will increase. That is because "hot traces" are frequent and can better exploit the time they reside in the cache. The increase in "live" time is shown in Figure 13. Live time is increased from 32% to 73% and from 42% to 72% in a 2KB and 4KB caches, respectively. The increase in percentage of time traces are "live" in the cache indicate that the traces residing in the cache are used more often used and therefore the sampling filter indeed favors "hot traces".

The reduction in number of builds has an impact not only on the power consumption but on the overall supply of instructions from the L1 hierarchy caches (Instruction and Trace caches) as well. In a regular system, traces are built from the L1 instruction cache and then stored in the trace cache. This causes duplication between the caches, which reduces the efficiency of the total cache area. By prolonging the time traces reside in the trace cache, the duplicated code of traces in the instruction cache is gradually replaced by the LRU mechanism. Therefore, "hot traces" reside in the cache while the "cold traces" reside in the Instruction cache. This partitioning is advantageous because the "cold" traces are numerous but rarely executed. Therefore "cold" code should be stored in the

most compact way. One example for this phenomenon is the Vpr benchmark. The trace cache miss rate of the Vpr benchmark is increased from 13.78% in the 4KB trace cache baseline machine to 14.36% in the 4KB sampling filter machine. However, the number of misses in the whole L1 hierarchy is 2.64 times smaller in the sampling filter machine, which increases the IPC of the sampling filter machine by 5% over the baseline machine. In the next subsection, a detailed measurement of the improvement of instruction supply from the L1 is presented.

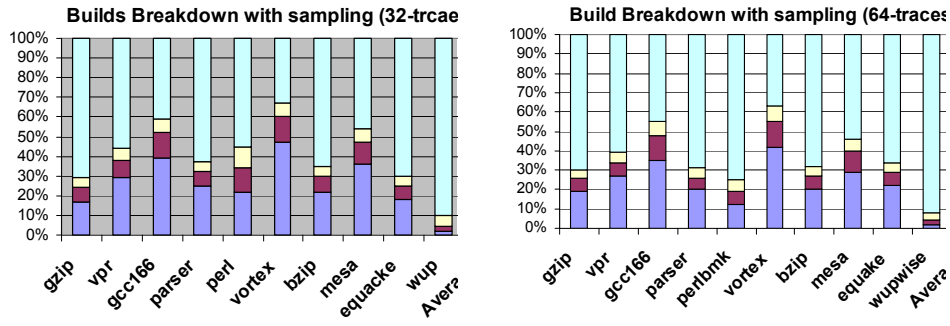

Fig. 12. Build breakdown by the number of hits resulted by each build for a 32-traces (left figure) and 64-traces (right figure). The percentage of useless builds is reduced dramatically.

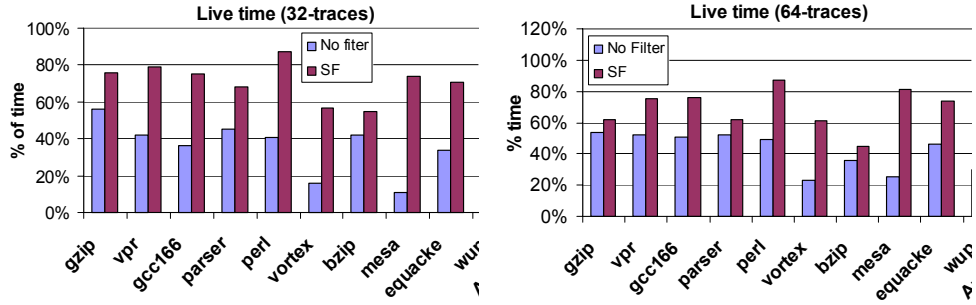

Fig. 13. Percentage of time traces are "live" in the cache for a 32-traces (left figure) and 64-traces (right figure).

4.4 Sampling Rate

The impact of the sampling rate (SR) on the quality of the filtering and on system behavior is discussed in this subsection. We consider the impact of the sampling rate on the miss rate of the trace cache, the overall misses in the L1 hierarchy (Instruction cache and trace cache), the IPC and ED^2. All the data presented in this section relate to the 2KB trace cache system.

The impact of the sampling rate on the trace cache miss rate for various benchmarks and for the average over all benchmarks is presented in Figure 14. Clearly, each benchmark has a different optimum point at which the miss rate is minimal. The optimum sampling rate point for the average of all benchmarks is 1/7, at which the miss rate is reduced by 11.3%, while the optimum point for the Parser benchmark is 1/3 and for the Mesa benchmark is 1/35. For all benchmarks except vpr and bzip, a high sampling rate (1/2-1/5) achieves most of the reduction in miss rate, because even a small reduction in the replacement rate is enough to assist the LRU mechanism and to reduce the pollution of the cache considerably. On the other hand, as the sampling rate is reduced beyond the optimum point, the trace cache miss rate starts to increase. Typically, the increase in miss rate is not as sharp as the decrease. This increase of miss rate is due to the fact the time locality is exploited less efficiently will be further discuss in subsection 4.5.

As described in 4.3, the sampling filter decouples the trace and instruction caches by storing traces in the trace cache for a longer time. This enables the corresponding blocks in the instruction cache to be replaced by the LRU mechanism. In Figure 15, the number of misses in the L1 hierarchy (instruction cache and trace cache) is presented for several benchmarks. The results are normalized so the number of misses in a system with no sampling filter is one. The figure indicates that the lower the sampling rate is, the stronger the decoupling effect is. Hence, the number of misses in the L1 hierarchy is reduced. For a sampling rate of 1/7 (best hit rate in trace cache) the average reduction in misses in the L1 hierarchy is 15%, while at a sampling rate of 1/60 the reduction in misses is of 28%.

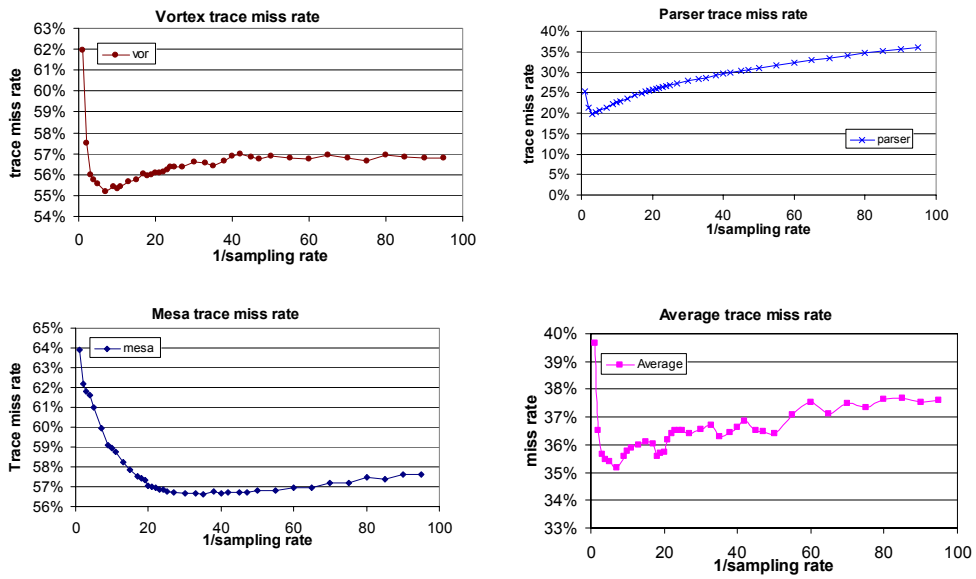

Fig. 14. Trace cache miss rate as a function of the inverse sampling rate. Each of the benchmark has a different optimal sampling rate. A sharp reduction in miss rate is achieved using high sampling rates while lower sampling rates tend to gradually increase the miss rate.

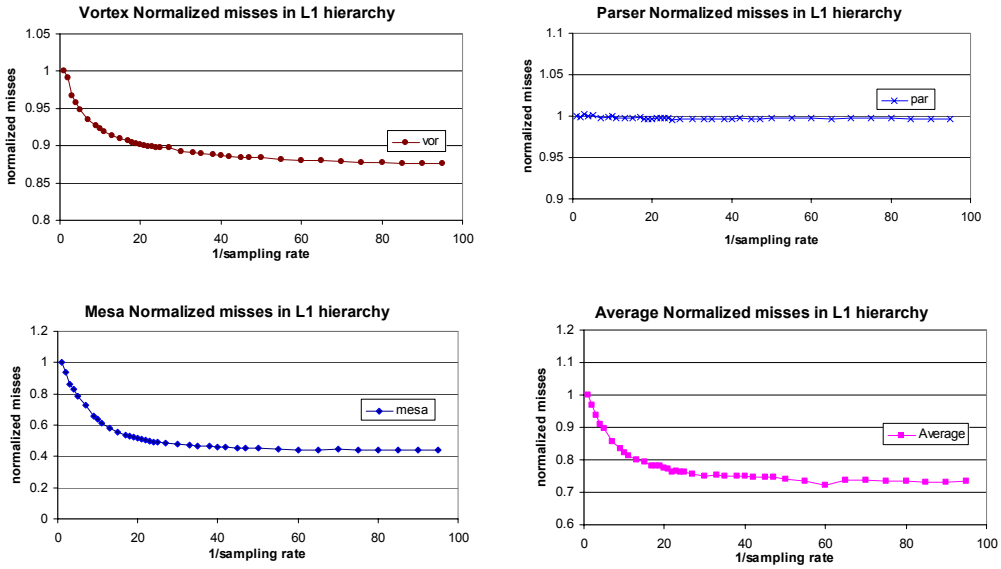

Fig. 15. Normalized misses in the L1 hierarchy as a function of inverse sampling rate. A lower sampling rate increases the decoupling between the instruction and trace cache, reducing the number of misses in the whole L1 hierarchy.

The performance (measured in IPC) of the entire system is affected both by the miss rate of the trace cache and the entire L1 hierarchy, as both have an impact on instruction

supply. The IPC vs. the inverse of the sampling rate is presented in Figure 16. The reduction in misses in the L1 hierarchy dominates the miss rate in the trace cache as the penalty of a miss in the L1 hierarchy is higher than a miss in the trace cache (in our model there is concurrent access to the trace cache and the instruction cache). The Parser benchmark performance tightly follows the trace cache miss rate behavior of the trace cache as the instruction cache is big enough to supply most of the missing instructions and therefore the decoupling is not important for this benchmark.

Different optimal points of sampling rate for trace cache miss rate and overall L1 miss rate exist. Both the trace cache hit rate and overall L1 hit rate will have different impact on the performance of the machine depending on the penalty associated with each one, e.g., in a sequential access mode (instruction cache is accessed only if there is a miss in the trace cache) the penalty for missing in the trace cache is higher and so the optimal sampling rate for performance is closer to the optimal sampling rate yielding the best trace cache hit rate. Notice that from a power perspective other tradeoffs might exist, if the build power is considerable (e.g., if constructed traces are also optimized [Patel and Lumetta 2001; Rosner et al. 2003]). In such a case, a lower sampling rate might be attractive even if it doesn't yield the best performance.

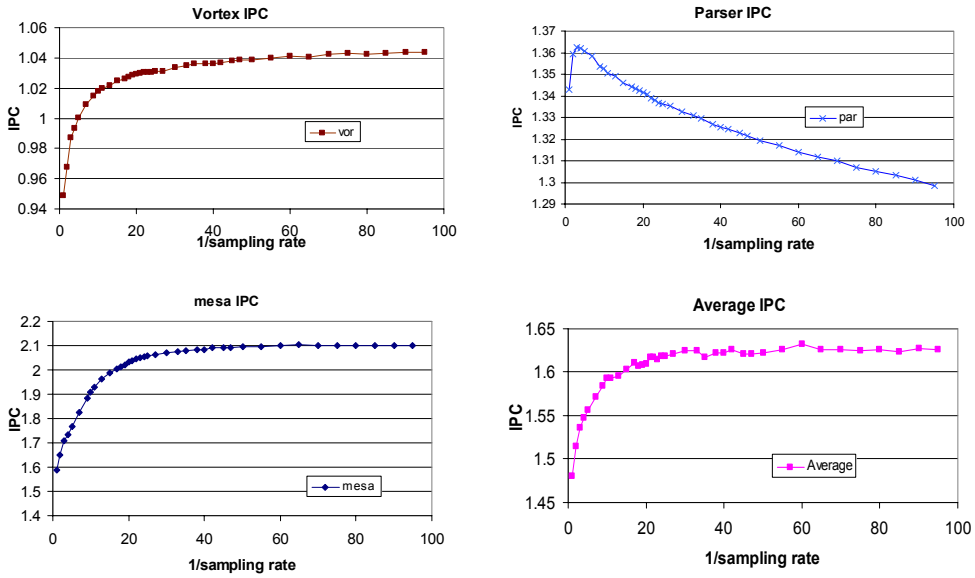

Fig. 16. IPC as a function of the inverse sampling rate. All benchmarks performance numbers but parser's are better for lower sampling rate as the reduction in misses in the L1 hierarchy is more important than the moderate increase in the trace miss rate. The parser benchmark performance follows tightly the miss rate behavior of the trace cache as the instruction cache is big enough to provide all the trace cache misses.

4.5 Sampling Filter Limitations

The sampling filter excels in its simplicity and its ability to provide a powerful mechanism to filter "cold traces", reduce the build rate and increase the overall L1 cache hit rate. Nevertheless, it has several limitations: the statistical sampling method is good as long as the "hot/cold" principle is in effect, some access patterns might be destructive and the cache might adapt itself more slowly to changes in the working set. In this subsection we examine the limitations of the sampling filter.

For each replacement algorithm there exists a destructive access pattern, e.g., the LRU performs very poorly for an access pattern that sequentially accesses blocks which are mapped to the same set and are slightly more numerous than the set associativity. Such a destructive access pattern exists also for the sampling filter. A destructive access pattern will consist of repetitively accessing bursts of traces such that the LRU will cause them to trash each other. In Figure 17, such a case is presented for a two-way associative cache. Traces A, B and C are mapped to the same cache line and are accessed for n times in a burst repeatedly. Without a filter, the first access in the burst is always a miss but the n-1 following accesses always hits. Therefore, the miss rate is 1/n. As the sampling filter inserts a new trace every 1/SR traces, the start of every new burst won't be inserted. In the worst case the sampling rate is 1/n and only the last access in the burst is inserted. In this case the hit rate will be zero as the inserted trace will never be used prior to its replacement. Such an access pattern is typical for the Parser benchmark.

The second limitation of a sampling filter is related to changes in the working set. Normally, a change in the working set introduces misses; new traces are constructed and inserted into the trace cache. As the sampling filter won't insert every new constructed trace it has a longer learning (adjustment) period. During that period, the cache will suffer a decrease in the hit rate. In Figure 18. the miss rate as a function of time (in cycles) is presented for various sampling rates for the first 150k cycles of the Perlbmp benchmark. The lower the sampling rate, the longer it takes the cache to capture the working set and to arrive to a steady state, *e.g.*, after 30k cycles the system with no sampling filter has reached the steady state, the system with a sampling filter at a sampling rate of 1/20 reaches the steady state after 50K cycles and the system with a sampling rate of 1/100 reaches the steady state after 130k cycles.

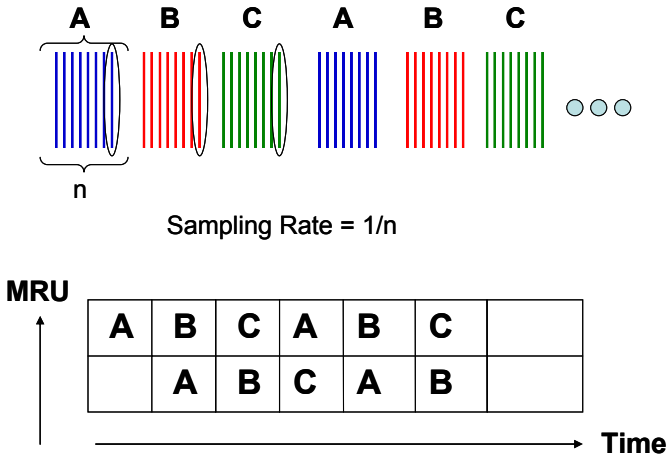

Fig. 17. An anti-sampling filter access pattern example.

The trace cache miss rate of the Perlbmk benchmark as the program progresses (in committed instructions) is shown in Figure 19. for a sampling rate of 1/2 and 1/50 over 10 million executed instructions. A zoom over one change in the working set is shown in Figure 20. for various sampling rates. During a change in the working set, the miss rate of the sampling filter system increases significantly compared to the system without a filter. A lower sampling rate causes a higher miss rate during working set changes and a longer time for the cache to readapt. However, the adaptation time is typically insignificant and it doesn't impact the overall hit rate by much.

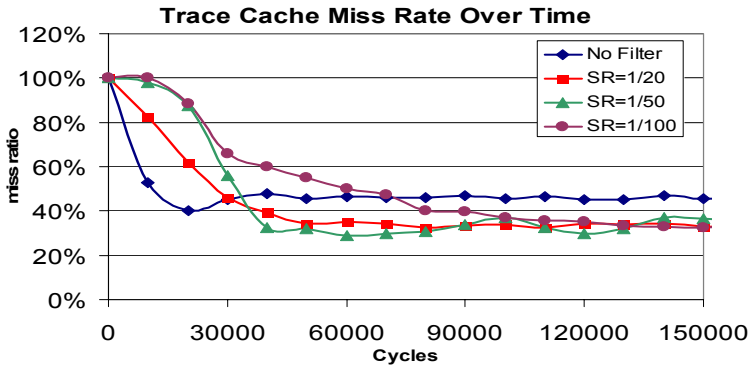

Fig. 18. Trace cache miss rate over time for various sampling rates for the Perlbmk benchmark.

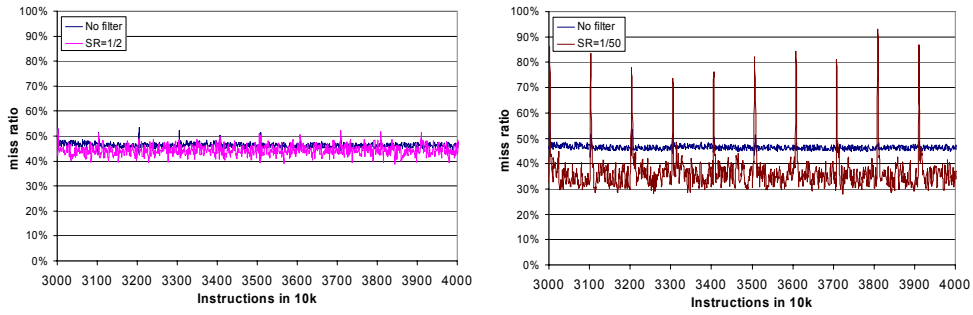

Fig. 19. Miss rate of a trace cache as a function of Perl benchmark progress in instructions committed for a sampling rate of 1/2 (left figure) and 1/50 (right figure).

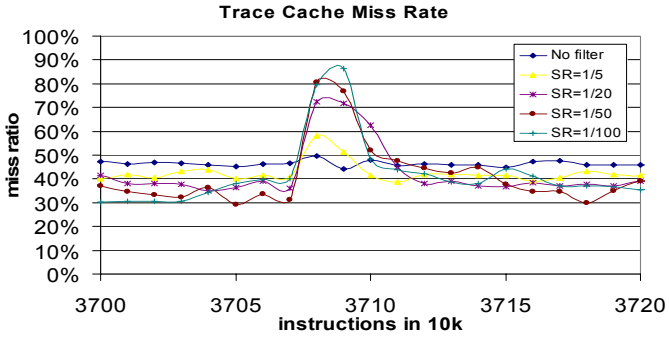

Fig. 20. Miss rate of a trace cache as a function of Perl benchmark progress in instructions committed for various sampling rates.

5. SAMPLING FILTER ENHANCEMENTS

5.1 Addressing sampling filter limitations

The sampling filter is very efficient in reducing the number of builds, increasing the coverage and hit rate for some benchmarks, and it excels in its low hardware complexity. Nevertheless, the sampling filter has several limitations (described in section 4.5): slower adjustment time to a new working set and sensitivity to destructive access patterns. The slower adjustment to a new working set happens even if the new working set had been encountered previously in the past, because of lack of memory. The sampling filter uses a random approach and therefore it can't distinguish between "hot traces" and "cold traces" even if those traces are encountered again and again. Therefore, a "hot trace" that is evicted from the cache cannot enter the cache until it is sampled again. In this section we address the limitations of the sampling filter by presenting two classes of solutions: the first class allows traces which are already known as "hot" to bypass the sampling filter

and enter the trace cache directly. The other class of solutions tightly captures "hot traces" in the cache.

5.2 Keeper table add-on

The keeper filter adds a capability to recognize "hot traces" at the filtering stage, enabling them to bypass the sampling filter. The Keeper filter combined with a sampling filter trace cache system is presented in Figure 21. Counters monitor the behavior of traces in the cache. Each time a trace is found in the cache, its counter is increased (until a saturated value). If a trace is replaced and its trace counter is higher than a certain threshold then the trace is considered to be a "hot trace" and its trace ID (consisting of the start address and branch direction vector) is stored in a separate structure: the Keeper table. The Keeper table is just an n-way small tag array and holds no data. The enhanced filtering mechanism checks the Keeper table for every new trace. If there is a tag match then the trace is considered "hot" and therefore is inserted immediately to the cache via the "hot path", bypassing the sampling apparatus. Traces whose trace IDs don't have a match in the Keeper table are subject to the random sampling. If the trace passes the sampling filter then it is inserted to the cache via the "cold path", otherwise it is discarded. The sampling filter is essential for training the keeper table and for cases in which the keeper table is too small to contain all the "hot trace" IDs.

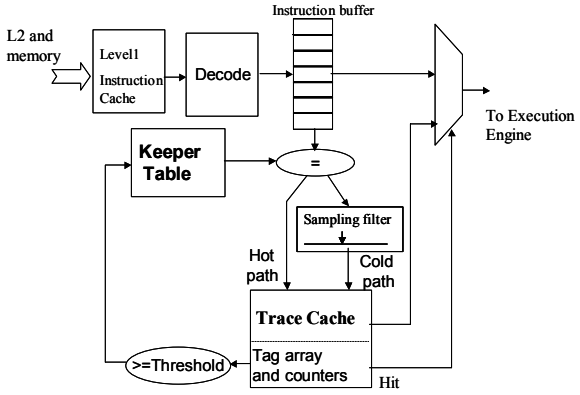

Fig. 21. The keeper filter system. All traces inserted to the cache are monitored by counter and upon replacement the trace id of traces with a counter value higher than a threshold are stored in the keeper table. Traces can be inserted to the cache via the “hot path” if there trace id match in the keeper filter or otherwise through the “cold path”. The “hot path” allows traces to bypass the sampling filter while the “cold path” consists of the sampling filter.

The addition of a keeper table has two effects: it ensures that "hot traces" will be inserted immediately to the trace cache and thus will exploit the maximum reusability potential of their current appearance, and it has a "cooling" effect on the stream of traces the sampling filter encounters. The first effect directly addresses the limitations of the sampling filter. It adds the capability to remember "hot traces" (via the keeper table) and therefore eliminates the learning period when they appear again. The "hot" path also makes the system less vulnerable to access patterns that repeatedly access the same trace. The second effect has an important impact on the stream of traces observed by the sampling filter. As already demonstrated most of the builds are of "cold traces" and the "hot traces" are mostly executed from the trace cache. By extracting the previously encountered "hot traces" from the stream of constructed traces, the remaining stream contains in higher probabilities "cold traces" and therefore applying the sampling filter with a lower sampling rate is even more attractive.

The additional hardware that is needed in order to implement the Keeper filter is the Keeper-table itself and the counter associated with each trace in the cache. The Keeper table is maintained small as it doesn't hold the traces and just stores the tag of the "hot trace" IDs. Moreover, as the Keeper-table holds information only of "hot traces" and isn't required to handle the bookkeeping of the entire trace-name space, which is much larger than the "hot trace"-name space, it can be efficiently small. The access to the keeper table prior to the insertion of a trace to the trace cache doesn’t add any delay to the critical path (supplying traces to the execution engine). The keeper table is updated only when a new trace is built and after a trace is replaced, so the number of accesses to this structure is fairly small. As the number of accesses to the keeper-table is small and the structure is maintained small, the power penalty of adding this structure is not excessive.

The counters associated with each trace in the cache can be implemented by augmenting the tag array to contain the counter value besides the tag. Each time the tag is probed for a match the counters of the set are probed too and in case of a hit in one of the ways the counter is increased and stored back. The trace array is unchanged.

An alternative way to implement the counter mechanism is to store the counters value in a separate structure: the counter table. The counter table is a cache without the tag array that has the exact same dimensions of the trace cache: same number of sets and same number of ways. Each time a change in the counters value is needed (increase due to hit or reset due to a replace) the information regarding the set number, the way number and the needed action is passed to this separate structure via a buffer. There is no need to match any tag: only the correct set and way are accessed. The advantage of this method is

that the trace cache itself is unchanged so there is no increase in the access time. The structure of the system with a counter table is presented in Figure 22. Upon a hit or replacement the set, way and the operation to be preformed on the counter is passed to the Cmd buffer. The counter table is updated by reading the operations and destination from the Cmd buffer offline with no impact on the cache behavior.

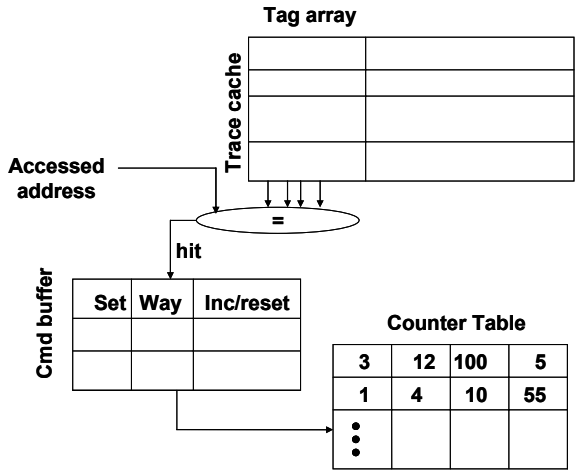

Fig. 22. The counter table structure. The counter table contains counters value of the corresponding trace in the trace cache. It eliminates the need to change the trace cache structure.

5.3 Sampling Filter with FTC-MTC organization

An alternative approach to enhance the sampling filter performance is to "cool" the input stream of traces to the sampling filter by tightly capturing the "hot traces" in the cache by some "sticky" mechanism. The FTC-MTC organization tightly captures "hot traces", as traces that were proven useful are stored in the main trace cache (MTC) with a reduced likelihood of replacement.

The structure of the combined architecture is presented in Figure 23. Only sampled traces are inserted into the FTC. The FTC observes the behavior of the traces using counters and once a trace is replaced its counter is checked against a threshold and if the trace is identified as a "hot trace" it is stored in the MTC.

The sampling filter enables the FTC to better learn the behavior of traces. The FTC purpose is to identify the most frequent used traces so they could be stored in the MTC. The sampling filter reduces the replacement rate and allows the FTC to monitor the behavior of traces for longer time and thus to expose their true nature. The FTC-MTC organization is able to capture in the MTC the most frequent traces ("hot traces") much

more effectively than a regular cache and therefore the stream of potential traces (traces not found in the cache) contains a higher percentage of "cold traces". This makes the usage of a sampling filter more attractive as the sampling filter is based on the statistical observation that most of the builds are of "cold traces".

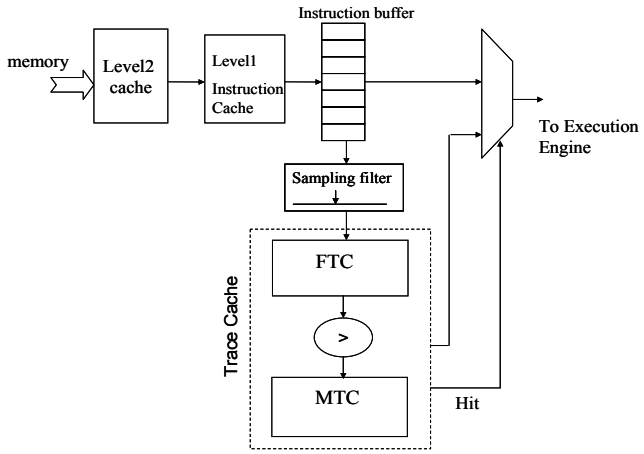

Fig. 23. The FTC-MTC system with a sampling filter. The trace cache is composed from a Filter Trace Cache and the Main Trace Cache. Traces are inserted to the FTC through the sampling filter and only the most useful traces in the FTC are passed to the MTC.

5.4 Frequency Based Replacement

The usage of counters to monitor each trace in the cache, as done by the Keeper filter, can be exploited in order to make better replacement decisions than relying only on LRU. LRU is the most common replacement policy of processor caches due to its simplicity and adaptivity. LRU is based on the assumption that the least recently referenced blocked is the less likely block to be referenced in the future. This method, though simple, makes only a limited use of the access pattern history. It ignores the frequency of usage of blocks giving the last sequence of references all the weight in the decision which block should be replaced. Using a filter reduces the number of replacements considerably, and thus increases the number of references to blocks in the set between two replacements. This raises the question whether the LRU restricted scope can actually predict which of the traces will be the least "useful" in the time between two consecutive replacements. The least valuable trace would be referenced the least between two replacements. An alternative replacement policy could be based on the frequency of appearance (FBR: Frequency based replacement). Such policies as the FBR [Robinson and Devarakonda 1990], LRFU [Lee et al.] have been proposed in the past in context of

virtual memory, database caching and web caching. Using a frequency based replacement algorithm seems to be too complex for processor caches because it requires managing the counters for hits, and adds complexity to the replacement itself. On the other hand, if we consider that the keeper filter already maintains the counters in a way that doesn't add latency and that the number of replacements is reduced significantly by the filter, this replacement policy can become practical. The number of replacements is reduced significantly and thus the number of counter comparisons applied is reduced too.

The new replacement policy algorithm always chooses the least frequent trace to be replaced. As before, each time there is a hit, the appropriate counter is increased to a saturated value. In order to ensure that a frequent trace which becomes infrequent will be replaced eventually, at each replacement all the counters of the set are divided by an aging factor. Choosing an aging factor that is a power of two enables one to exchange the costly division operation with a simple shift operation. As the replacement algorithm is more complex, the latency to insert a new trace to the cache is increased by one cycle.

The average miss rate of all benchmarks except wupwise as a function of inverse sampling rate for a 4KB trace cache with an LRU replacement mechanism and a trace cache with FBR replacement mechanism is presented in Figure 24. The Wupwise benchmark was removed from the average as the it shows a massive reduction in the miss rate from 52.8% to 10.9% for both the LRU and FBR so it dominates the other results. The advantage of using FBR is two-fold: it reduces the miss rate and allows to use a lower sampling rate. The best miss rate achieved by the sampling filter and the LRU replacement policy is 16.9% at a sampling rate of 1/5 while the combination of the sampling rate with FBR achieves a miss rate of 14.8% at a sampling rate of 1/20. Furthermore, at a sampling rate of 1/90 the average miss rate of the LRU system is as high as the miss rate of the non-filtered system while the miss rate of the FBR system is 16.4%, still better than the best miss rate achieved by the sampling filter and LRU combination.

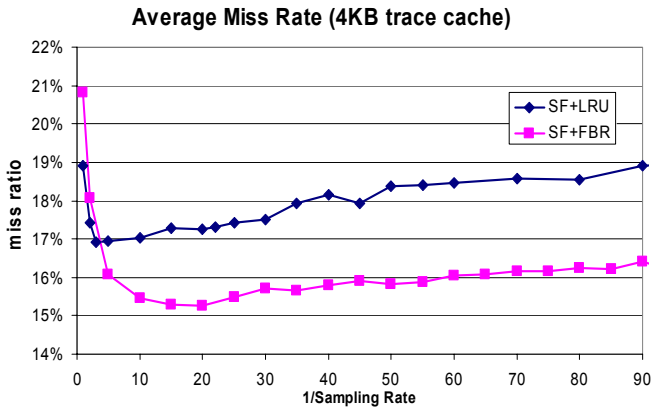

Fig. 24. Average miss rate over all benchmarks but Wupwise of a 4KB trace cache with a LRU and FBR replacement mechanism as a function of the reverse sampling rate.

5.5 Comparing Results of Different Enhancements

The Miss rate of a 2KB and 4KB trace caches with different filtering techniques applied is presented in Figure 25. For the 2KB trace cache, the combination of the sampling filter with the FTC-MTC organization achieved 13.1% lower miss rate than the sampling filter. The keeper filter and the FBR reduce the miss rate by 16.1% and 10.2% over the sampling filter cache, respectively. The average miss rate was reduced from 33.8% to 23% (a 31.9% reduction in miss rate) by the keeper filter over the baseline machine. For the 4KB trace cache, the combination of the sampling filter and the FTC-MTC organizations reduces miss rate by 20.9%, the keeper reduces miss rate by 11.9% and the FBR reduces the miss rate by 9% over the miss rate of a sampling filtered system. The best filtering enhancement for the 4KB cache is the FTC-MTC organization that reduces the miss rate from 22% (non filtered cache) to 13.2% (a 40% reduction).

The IPC of a 2KB trace cache and a 4KB trace cache with different filters is presented in Figure 26. Both for the 2KB and 4KB caches the IPC gained from the enhancements to the basic sampling filter technique is negligible. This is mainly due to the low trace cache miss penalty in the concurrent access model, and due to the fact that decoupling between the instruction and the trace cache is achieved by the simple sampling filter.

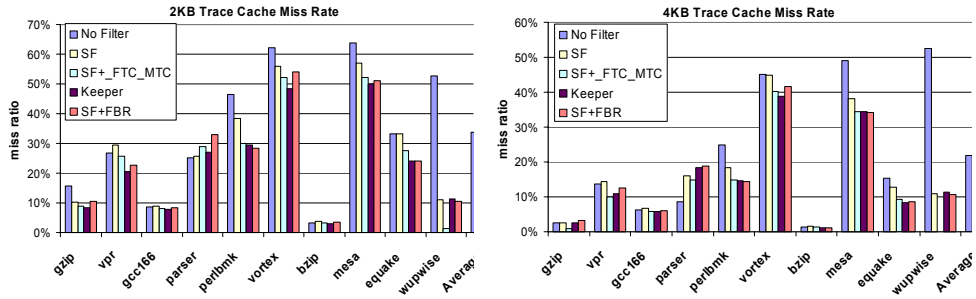

Fig. 25. Miss rate of a 2KB (left figure) and a 4KB trace cache (right figure) with different filter techniques applied.

The ED^2 reduction of the different filter mechanisms over the non-filtered system for a 2KB and 4KB trace cache is presented in Figure 27. For both the 2KB and 4KB caches the only benchmark that showed an increase in the ED^2 is the parser benchmark. On average the enhancement of the sampling filter by a keeper table, by a FBR or by the FTC-MTC organization achieves better performance-power efficiency reduction than the usage of the sampling filter only.

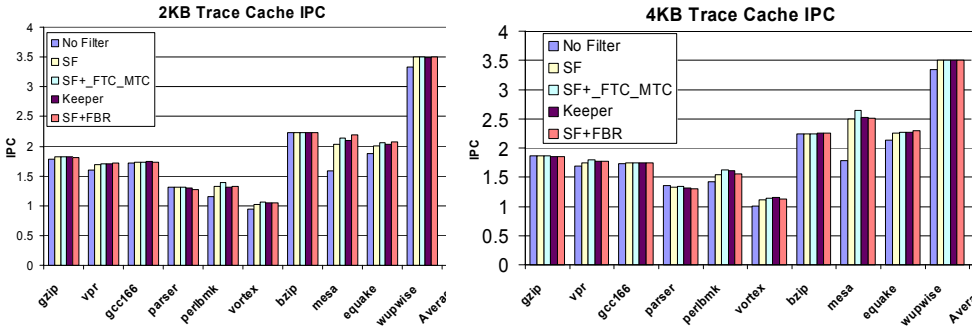

Fig. 26. IPC a 2KB (left figure) and a 4KB trace cache (right figure) with different filter techniques applied.

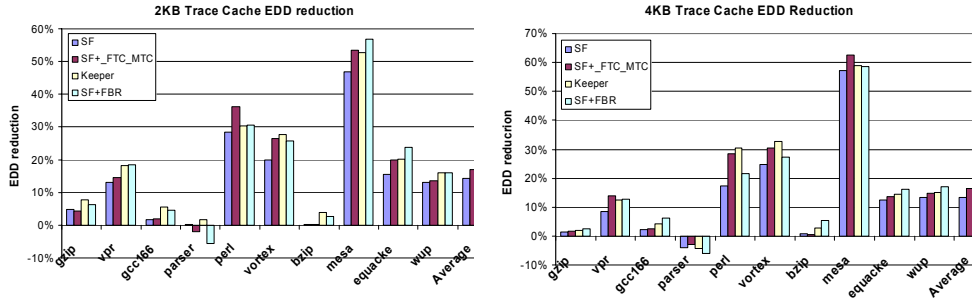

Fig. 27. The ED^2 reduction for a 2KB (left figure) and a 4KB trace cache (right figure) with different filter techniques applied over the non-filtered system.

6. CONCLUSION AND DISCUSSION

This paper presents a novel filtering approach for trace caches: the sampling filter. The sampling filter is a simple but effective mechanism to filter out "cold traces" and allow "hot traces" to better reside in the cache. This improves the cache utilization, the hit rate and overall system performance. Furthermore, the number of builds is reduced significantly, which in turn reduces the power associated with trace builds. The sampling filter is based on the observation that most of the builds are of "cold traces", which have only little chance to be accessed again prior their eviction from the cache, while "hot traces" reside fairly well in the cache but are occasionally disturbed by the "cold traces". Rather than performing bookkeeping over all traces in the program, the filter selects traces on a periodic basis. This requires minimal hardware, which makes this filter efficient. The sampling filter is extremely effective in increasing the utilization of the cache: the average "live" time of traces in the cache increased from 42% to 72%, the number of hits per build increased by a factor of 36 and the miss rate was reduced by 24% for a 4KB trace cache. Furthermore the filtering of "cold traces" allowed to reduce the duplication between the instruction cache and trace cache and thus to increase the number of instructions supplied from the Level1 cache hierarchy.

The sampling filter is limited by longer adjustment time to new working sets, burst access patterns and the time a "hot trace" resides outside the cache once replaced. This limits the useful sampling rate. In order to deal with those limitations several mechanisms were proposed, which enhance the sampling filter. The keeper table performs bookkeeping in order to allow "hot traces" to bypass the sampling filter. Unlike other filters that perform bookkeeping, the keeper table stores only information about the most useful traces rather than all traces, enabling the bookkeeping table to be quite small. A 32-entries keeper table reduces the miss rate by 11.9% over the basic sampling filter system for a 4KB trace cache. The combination of the FTC-MTC organization with the sampling filter allows the "hottest" traces to reside very tightly in the MTC, as traces are inserted to this cache only after passing two filters in series: the sampling filter and the threshold filter. This reduces the miss rate by 21.9% over the basic sampling filter system for a 4KB cache. The third extension to the sampling filter mechanism proposed in this paper is the usage of a frequency based replacement (FBR) algorithm instead of the conventional least recently used (LRU) mechanism. It was shown that the FBR mechanism allows to reduce the sampling rate and to deliver a better hit rate, e.g., the miss rate of a 4KB cache using a sampling filter with a sampling rate of 1/90 and FBR

was lower than the optimal miss rate of a sampling filter system with LRU replacement achieved at a sampling rate of 1/5.

One of the main goals of this work was to increase the performance of the processor while reducing the power consumption. This paper demonstrated that increasing the cache size in order to improve performance might defeat the purpose, because a larger cache has a longer access time and it dissipates more power. The naïve alternative of using a small trace cache reduces power at the cost of lower performance. Our approach to store the few frequently used traces in a small filtered cache while storing the majority of the static code in the more storage efficient instruction cache allows increasing performance without using larger caches.

Although the method is demonstrated for optimizing the power efficiency of the front-end of the machine (trace cache), the same approach can be applied to many other aspects of modern computer architecture design. For example, the sampling technique can be applied to any cache structure that stores a population of data items which exhibits a 80/20 behavior. Following this paper, various other architectural structures can be improved by introducing statistical sampling.

REFERENCES

Black, B., Rychlik, B., and Shen, J. P. 1999. The block-based trace cache. In *Proceedings of the 26th Annual international Symposium on Computer Architecture* (Atlanta, Georgia, United States, May 01 - 04, 1999). International Conference on Computer Architecture. IEEE Computer Society, Washington, DC, 196-207.

Brooks, D., Bose P., Schuster, S.E., Jacobson, H., Kudva, P.N., Buyuktosunoglu, A., Wellman, J.A., Zyuban, V., Gupta, M., and Cook, P.W. 2000. Power-Aware Microarchitecture: Design and Modeling Challenges for Next-Generation Microprocessors, IEEE Micro, November/December 2000 (Vol. 20, No. 6) pp. 26-44.

Brooks, D., Tiwari, V., and Martonosi, M. 2000. Wattch: a framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th Annual international Symposium on Computer Architecture* (Vancouver, British Columbia, Canada). ISCA '00. ACM Press, New York, NY, 83-94.

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

Henning, J. L. 2000. SPEC CPU2000: Measuring CPU Performance in the New Millennium. *Computer* 33, 7 (Jul. 2000), 28-35.

Hinton, G, Sagar, D., Upton, M., Boggs, D., Carmean, D., Kyker, A., and Roussel, P. The microarchitecture of the Pentium 4 processor. Intel Technology Journal, Q1, 2001.

Jacobson, Q., Rotenberg, E., and Smith, J. E. 1997. Path-based next trace prediction. In Proceedings of the 30th Annual ACM/IEEE international Symposium on Microarchitecture (Research Triangle Park, North Carolina, United States, December 01 - 03, 1997). International Symposium on Microarchitecture. IEEE Computer Society, Washington, DC, 14-23.

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

Kosyakovsky, O., Mendelson, A., and Kolodny, A. 2001. 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.

Lee, D., Choi, J., Kim, J. H., Noh, S. H., Min, S. L., Cho, Y., and Kim, C. S. 2001. LRFU: A Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies. IEEE Trans. Comput. 50, 12 (Dec. 2001), 1352-1361.

Patel, S. J.,Friendly, D. H., and Patt. Y. N. 1997. Critical Issues Regarding the Trace Cache Fetch Mechanism. Technical Report CSE-TR-335-97, Department of Electrical Engineering and Computer Science, University of Michigan, May 1997.

Patel, S. J. and Lumetta, S. S. 2001. rePLay: A Hardware Framework for Dynamic Optimization. *IEEE Trans. Comput.* 50, 6 (Jun. 2001), 590-608.

Patterson, D. A., and Hennessy, J. L. 1994. Large and Fast: Exploiting Memory Hierarchy in *Computer Organization & Design The Hardware/Software Interface*: Morgan Kaufmann, 1994.

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

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

Ramirez, A., Larriba-Pey, J.L., and Valero M. 2000. *Trace Cache Redundancy: Red and Blue Traces***..** In *Proceedings of the 6th Intern. Symp. on High-Performance Computer Architecture*, pp. 325-333, 2000.

Ramírez, A., Larriba-Pey, J., Navarro, C., Torrellas, J., and Valero, M. 1999. *Software trace cache*. In *Proceedings of the 13th international Conference on Supercomputing* (Rhodes, Greece, June 20 - 25, 1999). ICS '99. ACM Press, New York, NY, 119-126.

Robinson, J. T. and Devarakonda, M. V. 1990. *Data cache management using frequency-based replacement*. In *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Univ. of Colorado, Boulder, Colorado, United States). SIGMETRICS '90. ACM Press, New York, NY, 134-142

Rosner, R., Mendelson, A., and Ronen, R. 2001. *Filtering Techniques to Improve Trace-Cache Efficiency*. In *Proceedings of the 2001 international Conference on Parallel Architectures and Compilation Techniques* (September 08 - 12, 2001). PACT. IEEE Computer Society, Washington, DC, 37-48.

Rosner, R., Almog, Y., Moffie, M., Schwartz, N., and Mendelson, A. 2004. *Power Awareness through Selective Dynamically Optimized Traces*. In *Proceedings of the 31st Annual international Symposium on Computer Architecture* (München, Germany, June 19 - 23, 2004). International Conference on Computer Architecture. IEEE Computer Society, Washington, DC, 162.

Rotenberg, E., Bennett, S., and Smith, J. E. 1999. *A Trace Cache Microarchitecture and Evaluation*. *IEEE Trans. Comput.* 48, 2 (Feb. 1999), 111-120.

Rotenberg, E., Bennett, S., and Smith, J. 1996. *Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching*. In *Proceedings of the the* 29th International Symposium on Microarchitecture (MICRO-29), Dec. 1996

Sahuquillo, J., and Pont, A. 1999. *The Filter Cache: A Runtime Cache Management Approach*. In Proceedings of the 25th Euromicro Conf., 1999, IEEE Computer Soc. Press, Los Alamitos, Calif., pp. 424-431.

Solomon, B., Mendelson, A., Orenstein, D., Almog, Y., and Ronen, R. 2001. *Micro-operation cache: a power aware frontend for the variable instruction length ISA*. In *Proceedings of the 2001 international Symposium on Low Power Electronics and Design* (Huntington Beach, California, United States). ISLPED '01. ACM Press, New York, NY, 4-9.

Vandierendonck, H., Ramírez, A., Bosschere, K. D., and Valero, M. 2002. *A Comparative Study of Redundancy in Trace Caches (Research Note)*. In *Proceedings of the 8th international Euro-Par Conference on Parallel Processing* (August 27 - 30, 2002). B. Monien and R. Feldmann, Eds. Lecture Notes In Computer Science, vol. 2400. Springer-Verlag, London, 512-516.

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

Received November 2005;