Trace classification and its use for profile-based trace cache management

Oleg Kosyakovsky Avinoam Kolodny

Avi Mendelson

Intel – Israel Technion, EE Department – Israel

{Oleg.Kosyakovsky, avi.Mendelson}@intel.com kolodny@ee.technion.ac.il

Abstract

This paper proposes a new technique to improve the performance and the power consumption of trace-cache architectures. The new technique uses profiling and is based on the observation that traces can be classified into 4 different classes in respect to their appearance patterns (we name them after different kinds of travelers): ***rare traveler*** is a trace which appears infrequently, and is used very little each time, ***seasonal tourist*** appears infrequently, but shows a lot of activity during a few appearances, ***frequent flyer*** is a trace that gets activated repeatedly throughout the program, and tends to be re-executed extensively each time it is brought to the trace cache, and ***air crew*** is a trace being used frequently during most of the program lifetime.

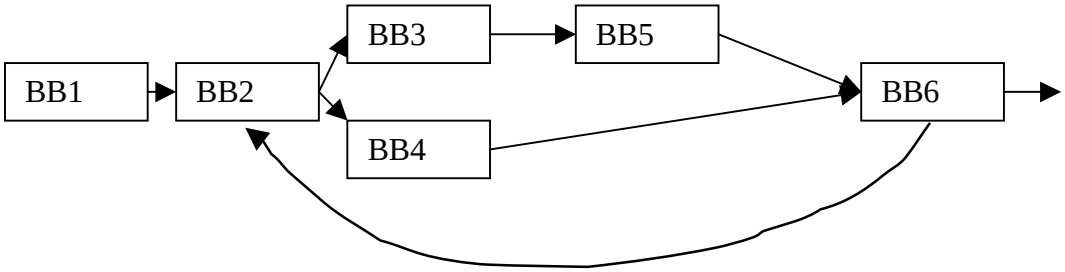
In the current technology, it is not worth to build a trace which does not show locality of references since it wastes more performance and power than the gain from using trace-cache. Thus we propose to use the above classification as a guideline for the processor's front-end when to build a trace and when to fetch it directly from the instruction cache. This paper will show that the new mechanism can be very effective for managing trace caches that suffer from high miss rate, and in particular, small to medium trace caches.

1 Introduction

Trace cache was conceived to support the increasing demand for wider fetch bandwidth in modern high-end processors. Typically, programs contain small chunks of instructions in sequential memory locations, separated by branching instructions. These

chunks are called basic blocks, and their average size is 4-5 instructions. Increasing the effective fetch bandwidth beyond the average size of a basic block is a non-trivial task, since a single fetch operation reads sequential stream of instructions, and “effective path” (the trace of the program) may consist of multiple basic blocks scattered in non-contiguous memory locations. Thus, enlarging the effective fetch bandwidth requires to place the code in such a way that sequence of instructions that use to be executed in the same “trace”, will be located in a sequential order in the memory. Trace cache suggests to do it dynamically, by constructing these traces and keeping them in a separate storage area called trace cache [13][17][18][19].

Figure 1-1 exemplifies the principles of trace cache operation. At run time, the fetch unit identifies basic blocks of sequential instructions and tries to pack as many basic blocks as possible into a trace structure, until one of the termination conditions such as max number of basic-blocks or max number of instructions is fulfilled. A trace is identified by its starting instruction address and by the branch conditions along its execution path. The traces are kept in cache structure that aims to keep only those traces which are most likely to be executed again in the future.



Trace cache

T1=BB1,BB2,BB3,BB5	T2= BB2,BB4,BB6,BB2
T4=BB1,BB2,BB4,BB6	T3= BB6,BB2,BB4,BB6

Figure 1-1 **An example of program flow and corresponding trace cache contents**

The performance benefit of the trace cache extends beyond fetching effectiveness: most trace-caches keep the instructions in decoded form, thus saving decoding energy and time. This feature is especially useful when the decoding pipeline stage is long, complex

and power-hungry, like the one for Intel's IA32 architecture [9][22][23]. Power and energy consumption are becoming a major concern of processors at all performance levels, not just at the low-end ones. Modern implementations of CISC architectures pose a particular challenge in the design of the processor's front-end (instruction fetch and decode) whose power consumption may reach as high as 28% of the overall processor power [10].

It is desirable to keep the instructions in decoded form in order to save power and improve performance, but the size of decoded instructions can be 3-4 times longer than that of undecoded ones. For example, when suggesting the use of 32KB cache, one cannot assume that it will hold up to 8K instructions (4 bytes for instruction), but rather ~2K instructions only. Increasing the cache size may harm both the access time and power consumption, thus our research is focused not just on large caches as most of other research did, but also on smaller trace caches, containing relatively few traces.

Previous research [18] indicates that using trace cache with relatively small size is not effective because of the cost of building/inserting traces into the trace-cache, and because of trashing of traces within the cache as well.

An attempt to improve the utilization of the trace-cache was proposed in [17]. A simple filtering mechanism was examined: Trace cache was divided so that part of it was used as a filter to identify frequently-used traces, while the other part was used to store them. Although a significant improvement in terms of power consumption was achieved, the filter part was found to be as big as the actual trace cache part, and the feasibility study indicates that more improvement could be achieved by means of more sophisticated techniques.

This paper proposes to use profiling information to reduce the number of times traces need to be built, and improve the utilization of the traces within the trace cache. We suggest to apply a profiling technique [11] on the program and store attributes describing the "behavior" of traces. Such profiling may be done in software or in hardware, during run time or in a dedicated code-optimization step. In this work we are examining the use of software-based profiling. Cache management policy that uses the stored profile information to improve performance and save power is proposed. We used simulations to perform trace profiling on several benchmark programs and to

analyze some characteristics of traces. Then the same simulation environment is utilized to test feasibility of a profile-based cache management policy.

The rest of the paper is organized as follows: Section 2 provides basic definitions and observations we made on the behavior of current trace cache technology. Section 3 describes the trace classification we introduced, while section 4 presents the results of applying trace filtering based on the classification we made in Section 3. We conclude this paper in Section 5.

2 Basic definitions and observations

We start this section with a short description of traces and trace-cache organization, then describe our software simulation environment. Next, we derive the formulas for estimating performance and power. The section concludes with several basic observations on trace-cache utilization.

2.1 Definitions

A trace is uniquely defined by its starting address and the outcomes of the branches along its path. A trace contains whole BB (basic blocks), and the only case in which a BB could principally be broken is when a single BB doesn't fit in the maximum size. In this work we limit the size of a trace to be 64 instructions (unlike [18[19] where 16 instruction limit is used) and the maximum number of BB in a trace is chosen to be 3. Since the restriction of BB number in a trace turned out to be stronger than the restriction on the number of instructions, average trace size was measured to be 12-13 instructions. Similar to other work, we apply additional trace termination conditions such as indirect and backward jumps always terminate a trace.

A trace cache consists of controls and data area. The control of each cache entry holds a trace tag composed out of its starting address, number of BBs and branch outcome of each block (encoded as a single bit). By requiring traces to end on boundaries of basic blocks, each trace is uniquely identified by its tag. We allow different traces to start at the same location as long as their tag is different. We consider trace caches that are 1,2,4 or 8-way set-associative. A set is managed using LRU replacement policy. Most

of the study in this paper will refer to cache sizes of 16-256 traces. If we consider 12-16 bytes representation of a decoded instruction, this corresponds to trace caches ranging from 3.5-4K bytes up to 50-60K bytes.

Similar to [17], we use an abstract model of a machine in which instruction processing occurs in three major sub-systems:

- ***Trace-building sub-system*** fetches instructions from conventional instruction memory, decodes them and groups decoded instructions into ***traces***.
- ***Trace bypassing subsystem*** fetches instructions directly from the instruction cache to the instruction windows, without creating the trace.
- ***Trace-management sub-system*** aims to handle the trace-cache and to decide upon trace-miss whether to build the trace or to use the trace bypassing logic.

The trace-management sub-system plays critical roles in both the performance and power domains. On one hand, it is expected to provide high throughput and a correct stream of instructions to the execution sub-system. On the other hand, it is expected to limit the use of the trace-building subsystem, which typically incurs long delays and consumes high power. Section 4 suggests the use of profiling information to help the trace management to be more efficient in terms of execution time and power consumption.

2.2 The simulation environment

We based our analysis on the SimpleScalar machine and its simulation tool suite [2]. Two software modules were added to the sim-outorder program: a *Trace_Collector* object, and a *Trace_Cache* object. The *Trace_Collector* is invoked at the dispatch stage of the processor, monitoring the instruction stream. If a trace need to be built, the trace collector collects the instructions till it reaches one of the trace-termination-conditions, and puts them as a trace into the trace cache. The *Trace_Cache* we implemented, is organized as a set associative cache and uses standard LRU policy as its replacement mechanism.

In our simulator, only addresses of instructions and trace identifiers are actually stored in *Trace_Cache*. In this simple simulation arrangement, the fetch engine actually gets instructions from lower-hierarchy memories, and detailed implementation of the

trace cache is bypassed. However, each executed trace is detected, hit-count and miss-count are updated, and access delays can be set depending on whether there was a hit or a miss. Additional data are collected during the simulation and recorded in a trace database, e.g. number of times each trace was built, number of times it was executed, number of hits/misses for each trace, and some statistics of activity over time. Since the primary purpose of this work was to characterize the traces a program creates during run time, we assumed that branch prediction is perfect, and traces are built as part of the front-end of the machine.

2.3 Performance and Power consumption Evaluation Criteria

The actual performance of a trace based system may depend on various parameters such as design style, number of stages in the pipeline and other chip design considerations. In order to avoid the over-complication, we decided to simplify the estimation of the overall performance and power consumption of the system, by using the following abstract model:

Assume that the CPI (Cycles Per Instruction) for a system that uses I$ (instruction cache) is given by $CPI_{I\$}$ and the CPI of a system that uses perfect T$ (trace-cache) is given by the parameter $CPI_{T\$}$ (a perfect T$ has a hit-rate of 100%,). We also assume that any build of a new trace causes B_{time} cycles for the machine to stall, and that the number of builds is given by #B. So, if the program contains N instructions, we can derive the following equations

$$cycles\_execute\_from\_I\$ = N * CPI_{I\$} \quad (1)$$

$$cycles\_execute\_from\_T\$ = N * CPI_{T\$} + B\# * Btime \quad (2)$$

and

$$exec\_time\_ratio = \frac{cycles\_execute\_from\_T\$}{cycles\_execute\_from\_I\$} = \frac{CPI_{T\$}}{CPI_{I\$}} + \frac{\#B}{N * CPI_{I\$}} * Btime \quad (3)$$

Equations 1-3 indicate that the capability of improving performance of a given architecture by adding trace cache depends on the ratio between CPI of the system with

T$ and CPI of the system without T$, and on the overhead we add to the system due to build process.

Using a trace cache allows us to reduce the power consumption of the front-end of the machine, because the instructions are stored in a decoded form; but we need to pay for the power it costs to build the traces. Thus, assuming that the average power consumption of an instruction coming from the I$ is $P_{I\$}$, the average cost of an instruction fetched from the T$ is $P_{T\$}$ and the power needed for building a trace is given by P_B, we get the following expressions for power consumed by the machine's front-end:

$$Power\_execute\_From\_I\$ = N * P_{I\$} \quad (4)$$

$$\boldsymbol{Power\_execute\_From\_T\$ = N * P_{T\$} + \#B * P_B} \quad (5)$$

$$\boldsymbol{power\_ratio = \frac{Power\_execute\_From\_T\$}{Power\_execute\_From\_I\$} = \frac{1}{P_{I\$}} * \left(P_{T\$} + \frac{\#B * P_B}{N} \right)} \quad (6)$$

In section 4 we will extend these equations to reflect the modifications we are proposing for a new technique that allows to fetch instructions either selectively from the T$ or directly from the I$.

2.4 Trace cache behavior and design-parameter considerations

In order to understand the tradeoffs in designing trace-cache based systems, we look at how the trace-cache affects the performance and front-end power consumption when running three applications: CC1, PERL and MK88SIM out of the Spec95 performance benchmark, using equations (3) and (6). Figures 2.1 and 2.2 present the relative execution time and power consumption of a T$ machine compared with a machine that uses I$ only.

These results were generated using the parameters shown in Table 2.1.

Parameter	Value	Parameter	Value
**$CPI_{T\$}$** - (CPI from T$)	1/4	**$P_{T\$}$** - (power per instruction working from T$)	1
**$CPI_{I\$}$** - (CPI working from I$)	1/2	**$P_{I\$}$** - (power per instruction working from I$)	1.5
B_{time} - (cycles per trace build)	8 cycles	**P_B** - (Power to build a trace)	16
N (number of instructions) **#B** (number of builds)	Taken from simulator		

Table 2.1 Parameters used in performance and power evaluation

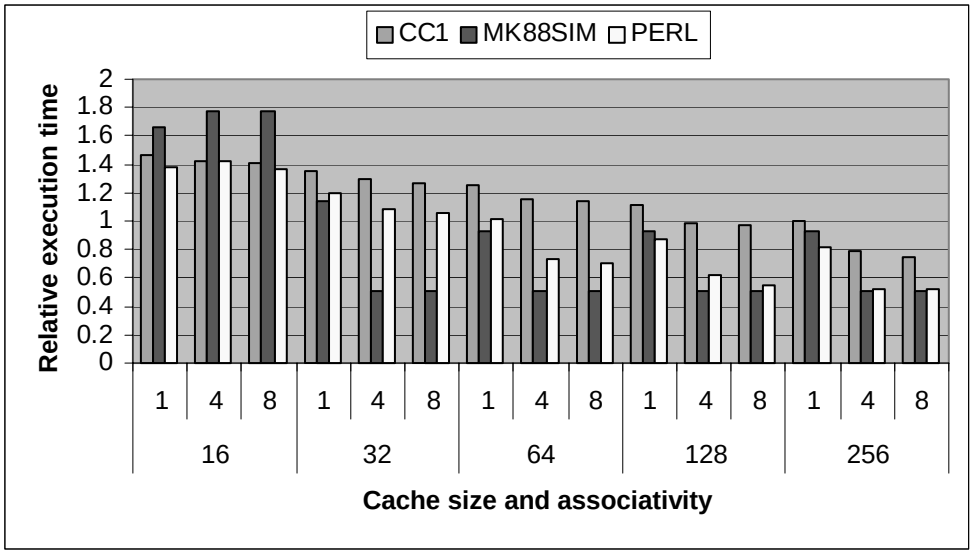


Figure 2-1Ratio of Execution Times

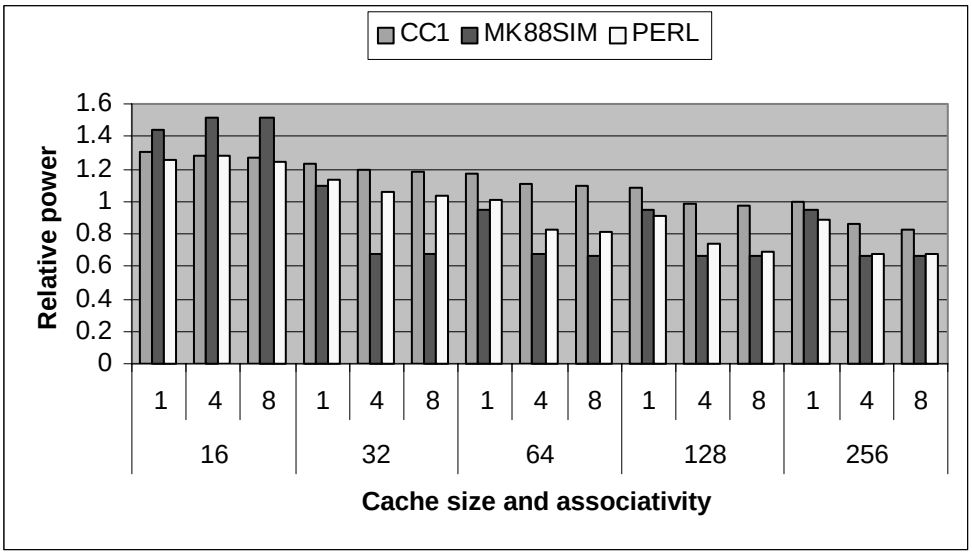


Figure 2-2: Ratio of Power consumption

Figures 2.1 and 2.2 present relative numbers; i.e., when the ratio is greater than 1, it means that the T$ based system loses in comparison with an I$ system (takes more time or wastes more power). As one can observe, some applications such as MK88SIM can take advantage of small trace caches with high associativity. Other applications, such as CC1, can take advantage of the trace cache structure only if a relatively large trace-cache is being used. When looking at the power, the situation is even worse. Here, with a small trace cache, most applications consume significantly more energy than on an I$ machine.

Since we want to reduce the size of the trace cache while preserving its advantages, let's take a closer look at the behavior of the CC1 application. We start looking at the trace cache hit rate it achieves. Figure 2.3 presents the impact of the trace cache size and its set-associativity (from 1-way up to 8-way) on the overall trace cache hit rate. As anticipated, the trace cache becomes more efficient with larger size and higher associativity. Note that Figure 2.3 presents the size of the trace-cache in terms of number of traces. Since traces are decoded, and assuming a trace can contain up to 16 instructions, a cache that can contain 256 traces needs to be as large as 50-60K bytes depending on the size of a decoded instruction.

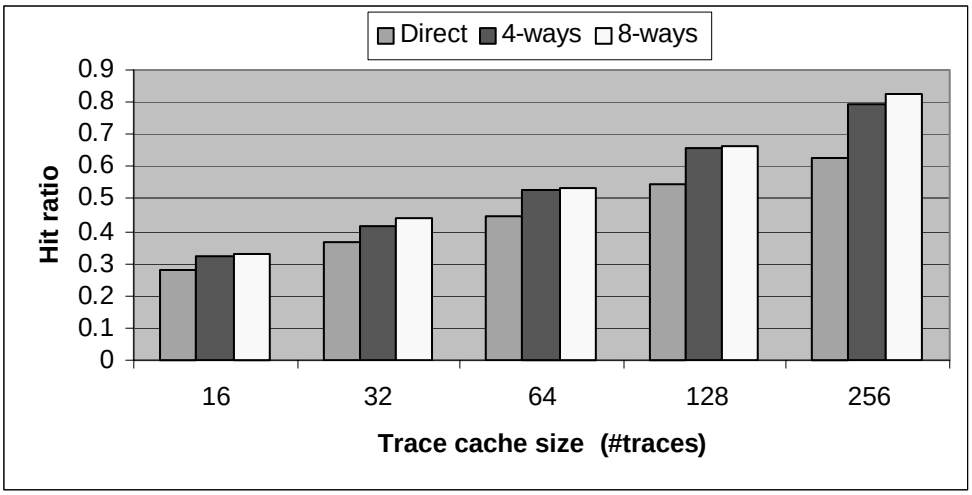


Figure 2-3: Trace cache hit rate of CC1 application

Looking at figure 2.3 it is clear that the main reason CC1 cannot take advantage of a small trace cache is that the number of builds would be very high then. A build of a trace that does not preserve locality costs power, causes replacement of a trace that might have better locality, and does not help the overall performance of the machine.

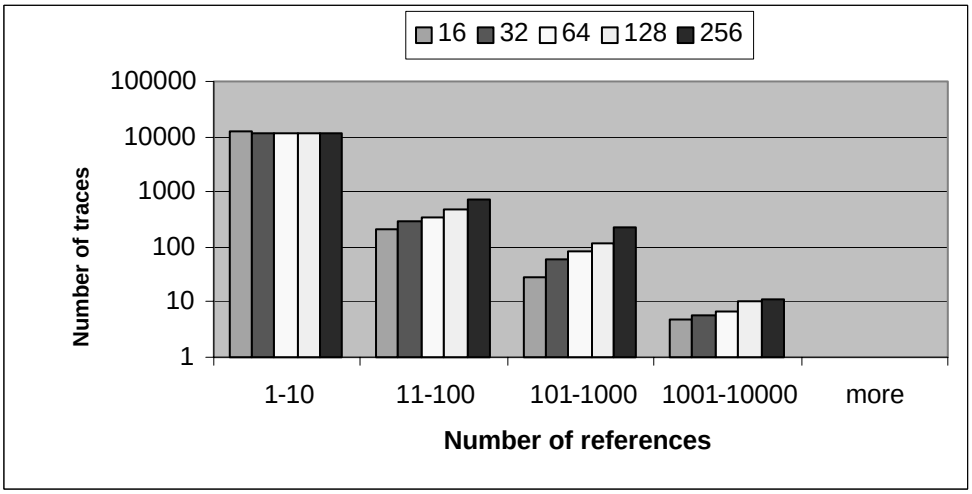


Figure 2-4: Distribution of trace reference count (log scale)

Figure 2.4 presents the distribution of the trace reference counts in logarithmic scale. Numbers in this chart represent the number of traces with specific range of reference count while trace resides in the trace-cache. For example, the leftmost set of bars corresponds to traces being accessed 1-10 times before their replacement from the

trace cache. We are using logarithmic scale due to the fact that the number of traces not showing locality of references highly dominates.

One could see that only very small number of traces are being used over and over. This is not surprising, since many mechanisms in modern computer architectures are based on the observation that a very small portion of the code is executed most of the time. It should not be surprising also to see (Table 2.2) that the majority of executed instructions in the whole program come from traces that are executed frequently. This classical Pareto behavior follows the locality of reference principle, underlying the design of all cache architectures.

Thus, if we could keep in the trace-cache only traces presenting high locality, we could save the build power, improve the trace-cache utilization and thus improve the overall power and performance of the trace-cache system, in particular when relatively small caches are being used.

Trace usages	1-9	10-99	100-999	1000-9999	more
Instructions fetched	146153	1016249	10898118	61657357	180089565

Table 2.2 – number of instructions fetched from each group of traces
(simulated assuming an infinite trace cache)

We have observed that trace distribution numbers are highly sensitive to the cache size and associativity (data is not shown here to save space). When a small and/or direct mapped trace cache is used, the lifetime of traces is reduced and more traces are being used only once before being replaced from the T$. When an infinite T$ is used, the number of traces which are being used only once is significantly reduced.

3 Classification

In order to improve the utilization of the trace cache, this paper suggests to use profile based techniques, which requires a new trace classification. This section aims to describe this new approach.

The analysis of trace appearance patterns in the dynamic instruction execution stream can be explained with the help of the following metaphor, comparing traces to

airline passengers. If traces were passengers, then one could classify them into four types described below and illustrated in Figure 4.1:

1. ***rare traveler*** – A trace which gets activated very few times during its lifetime within the trace cache (or even during the whole program run). We observed that most traces actually belong to this class.
2. ***seasonal tourist*** –A trace that shows a lot of activity during few time-intervals, but is inactive in general.
3. ***frequent flyer*** – A trace that gets activated repeatedly throughout the program, and tends to be re-executed extensively each time it appears in the trace cache.
4. ***air crew*** – A trace that gets executed so often, that it may reside in the trace cache all the time.

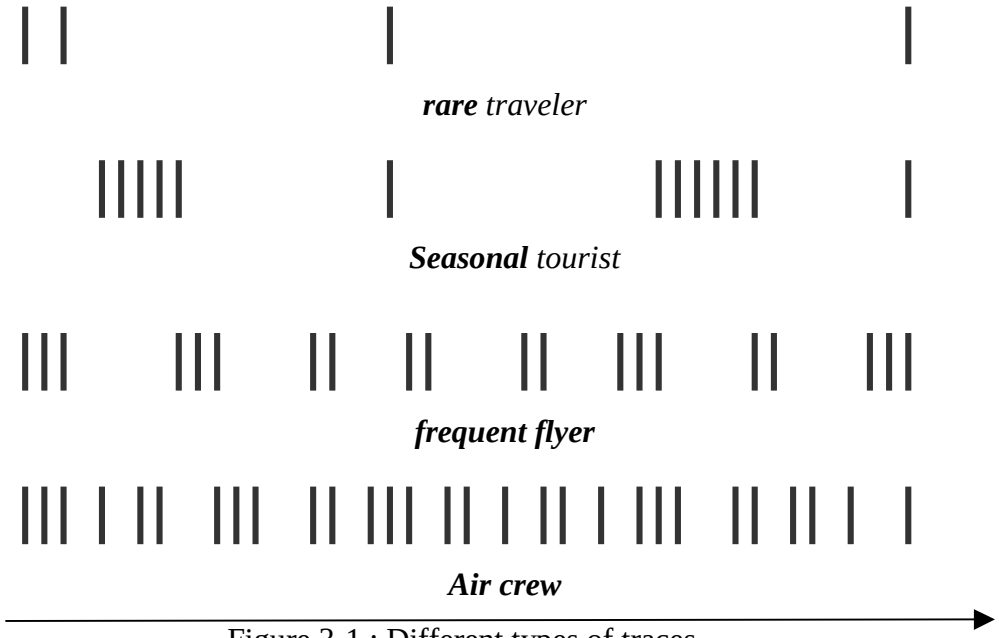


Figure 3-1 : Different types of traces

Given this classification, at run-time it is clear that *rare* traces need not be built or kept since their overall cost (in terms of time and energy) is higher than fetching the instructions directly from the instruction cache. It is also clear that *air crew* traces should be kept in the trace cache as long as possible, since their reusability is very high.

Handling the *frequent flyers* and the *seasonal tourists* is less obvious and will be discussed later on.

We were looking for a classification method that will be depend on the program behavior rather than on the system (trace cache) parameters. Thus, we choose the following heuristics: A sampling interval of 10000 trace accesses is arbitrarily defined (any interval of the order of several times the cache size would be appropriate). Within each sampling interval, the number of accesses for each trace is counted during simulation. If a trace is executed more than 10 times in an interval, the interval is considered "***active***" for this trace, and a counter *active_intervals_num* for this trace is incremented. Whenever an active interval follows an inactive interval, this is considered a "**switching**" of activity status, and a counter *activity_switches_num* for this trace is incremented. At the end of the profiling run, all traces that have a low *active_intervals_num* are attributed to be of type ***rare*** (in our experiments we used a low threshold of *total_number_of _intervals*/1000; i.e., we consider a trace to be rare if it active less then a fraction of 1000 of the total number of intervals). Traces with a high *active_intervals_num* are attributed as type ***air crew*** (in our experiments we used a high threshold of *total_number_of _intervals*/10). Remaining traces are marked as ***seasonal*** if their switching between inactive and active status is low (we used *activity_switches_num* < *(2/3)* active_intervals_num*). Finally, all the remaining traces are ***frequent flyers*** because they switch fairly often between active and inactive status.

Results of profiling runs on the benchmark programs are shown on Figure 3.2. Even without fine-tuning the classification thresholds, the results show that rare traces dominate in all applications. The classification is completely independent on trace-cache configuration though (only objective program behavior does matter). We have verified that the classification is also not very sensitive to input data. Therefore, a profiling run with a one set of input data can generate useful trace-type identification for use later on in run-time optimization.

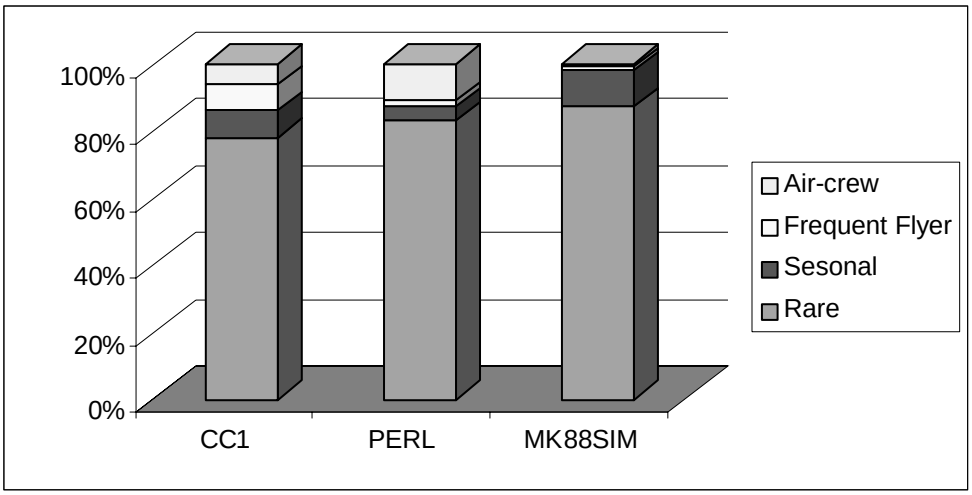


Figure 3-2: Distribution of different trace types in different applications.

4 Profile based filtering

Given the classification above, we tried several trace-management heuristics to "filter out" selected traces. In the figures below we will use the following working modes:

- **No filtering:**

Working with "regular" trace cache, where all traces are built, stored and executed from the trace cache; they may be later overwritten using standard LRU algorithm. This together with the system with no trace cache serves as a baseline for comparison.

- **"Smart" *(hit-based)* filtering:**

In this option, traces are not stored in the trace-cache unless they experienced at least one hit during the profiling run (after being built and saved in the trace cache). This method differs from the profile based methods described above, since it relates not only to particular program behavior, but to specific Trace Cache parameters as well.

Some (explainable) "anomalies" could be observed while using this policy. For example, the larger the Trace Cache, the less traces are filtered out, which is appropriately reflected in measures we get from simulation.

- **Combining all kinds of filtering**

  Here, both *rare*, *seasonal*, and traces that had no hits in profiling runs are excluded from the trace cache in performance simulation runs. The purpose is to filter as much as possible, with maximal optimization being expected.

4.1 Basic measurements

Figures 4.1 , 4.2 show the effect of the above kinds of filtering on trace cache hit rate and number of trace builds. Obviously, hit rate grows with cache size and associativity.

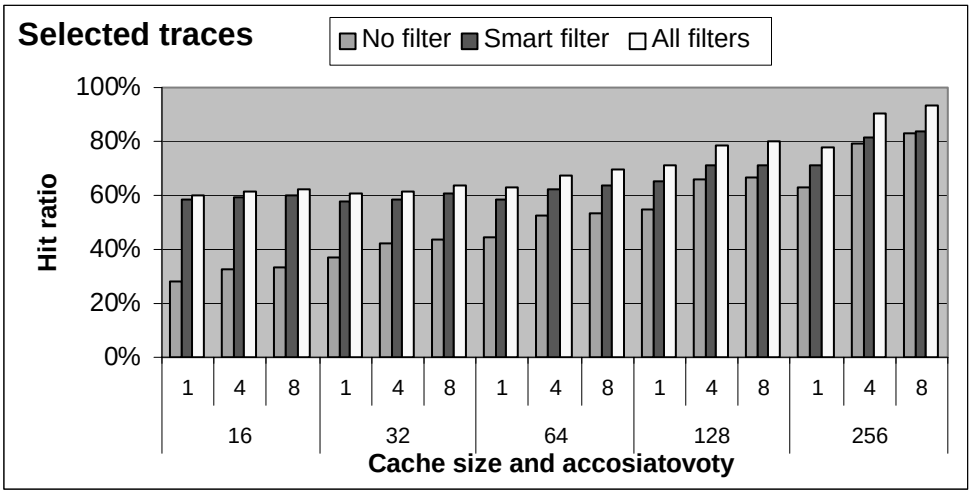


Figure 4-1 trace cache hit rate using 3 filtering options (CC1)

The hit rate presented on Figure 4.1 refers to the trace cache after the filtering has been applied. Here, the hit rate is defined as the fraction of hits out of accesses to non-filtered traces. This measure shows that the impact of the filtering on small trace caches is very significant, but for these sizes, the smart filter does most of the work. As the capacity of the cache increases, the smart filter becomes less effective, but the profile based one can still contributes.

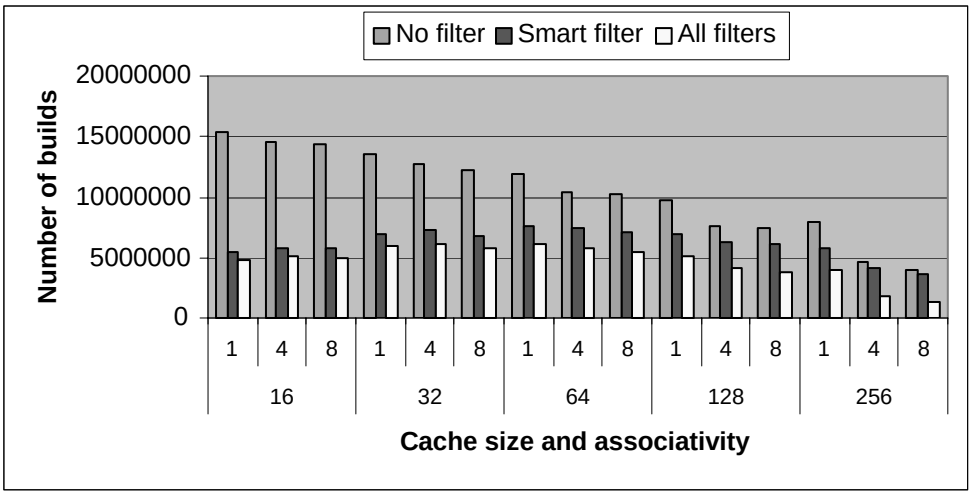


Figure 4.2: number of builds for various trace cahce configurations (CC1)

As could be seen on Figure 4.2 build count is the measure affected very positively by filtering. This is especially pronounced with smaller Trace Cache sizes. One could see that the same trend we saw in Figure 4.1 continues in Figure 4.2, but the amplitude of these effects is much more significant.

4.2 Basing profile on diffeent input

All the simulations we described so far, used the same input for both collecting trace profile data and measuring the impact of filtering. In real life, we should use our technique while obtaining the profile information with one set of inputs and examine the performance on another one. Such an experiment is dealt with in this section.

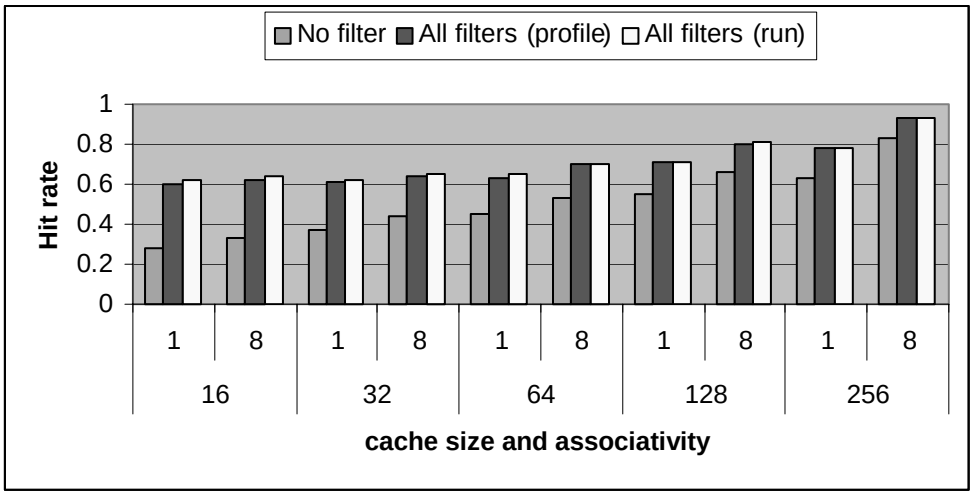


Figure 4-3: Hit rate comparison for different inputs

Figure 4-3 shows a run of CC1 with restricted (but still representative) number of Trace Cache configurations. One could see that the results we are getting are very similar to each other and so, it justifies our assumption that frequent flyer and air crew traces remain the same regardless of the input.

4.3 Performance and Power consumption Evaluation of filtered Trace-cache.

Here we enhance the method described in section 2 to enable evaluating the impact of the filtered trace-cache architecture on the overall performance and overall power of the system. We use the definition of $\boldsymbol{CPI_{I\$}}$, $\boldsymbol{CPI_{T\$}}$, $\boldsymbol{B_{time}}$ and ***#B*** as in section 2. However, when a filtered trace cache is used, some of the instructions are fetched from the T$ and some - from the I$ directly to execution. Therefore, we need to use the parameter ***Thit*** to indicate the fraction of instructions that have been fetched from the T$.

$$\boldsymbol{cycles\_NO\_T\$} = \boldsymbol{N} * \mathrm{CPI}_{\mathrm{I\$}} \qquad (7)$$

$$\boldsymbol{cycles\_filter\_T\$} = \boldsymbol{N}*\boldsymbol{Thit}*\boldsymbol{CPI_{T\$}} + \mathrm{N}*(1-\mathrm{Thit})*\mathrm{CPI}_{\mathrm{I\$}} + \mathrm{B\#}*\mathrm{Btime} \qquad (8)$$

$$\boldsymbol{exec\_time\_ratio} = \frac{\mathrm{CPI}_{\mathrm{T\$}}}{\mathrm{CPI}_{\mathrm{I\$}}} * \boldsymbol{Thit} + (1-\boldsymbol{Thit}) + \frac{\#\mathrm{B}}{\mathrm{N}*\mathrm{CPI}_{\mathrm{I\$}}} * \mathrm{Btime} \qquad (9)$$

As we can see the improvement in performance of the filtered trace cache over the system without the trace cache depends on the probability to hit the trace cache and the amount of builds we can save in this process.

Calculating the new power consumption in the front-end is similar to the calculation of the performance. Here

$$\boldsymbol{Power\_NO\_T\$} = \boldsymbol{N} * P_{I\$} \quad (10)$$

$$\boldsymbol{Power\_filter\_T\$} = \boldsymbol{N} * \boldsymbol{Thit} * \boldsymbol{P_{T\$}} + \boldsymbol{N} * (1 - \boldsymbol{Thit}) * \boldsymbol{P_{I\$}} + \#\boldsymbol{B} * \boldsymbol{P_B} \quad (11)$$

$$\boldsymbol{Power\_ratio} = \frac{1}{\boldsymbol{P_{I\$}}} * \left(\boldsymbol{P_{T\$}} * \boldsymbol{Thit} + \boldsymbol{P_{I\$}} * (1 - \boldsymbol{Thit}) + \frac{\#\boldsymbol{B} * \boldsymbol{P_B}}{\boldsymbol{N}} \right) \quad (12)$$

Figures 4.4 and 4.5 show the effect of filtering on performance and power using these equations. Comparing these numbers with Figure 2-1 and Figure 2-2 shows the significant contribution of our proposed new technique. For example, looking at the challenging CC1 application, execution times are reduced by 20% and more, using a moderate size cache of 128 traces. Power is also similarly reduced.

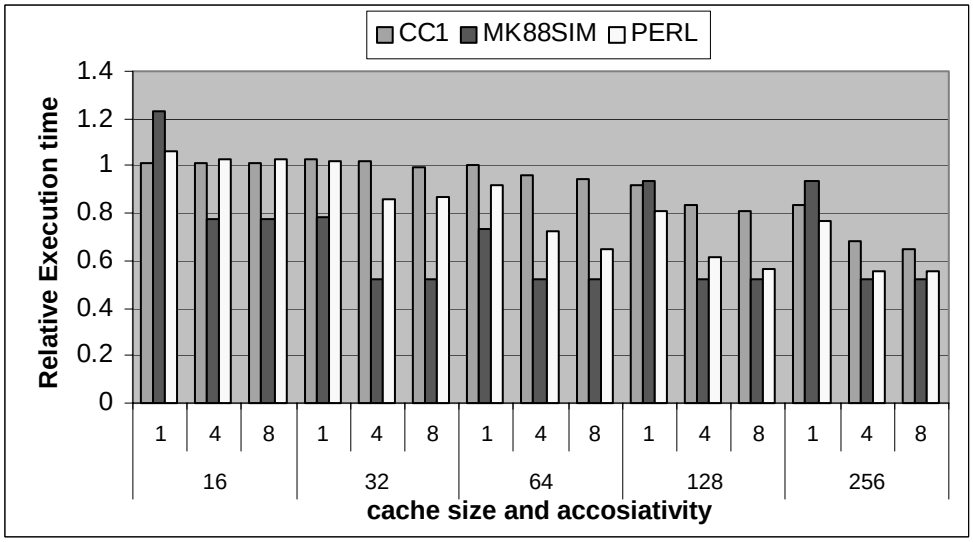


Figure 4-4: Relative execution time of a filtered T$ vs. I$

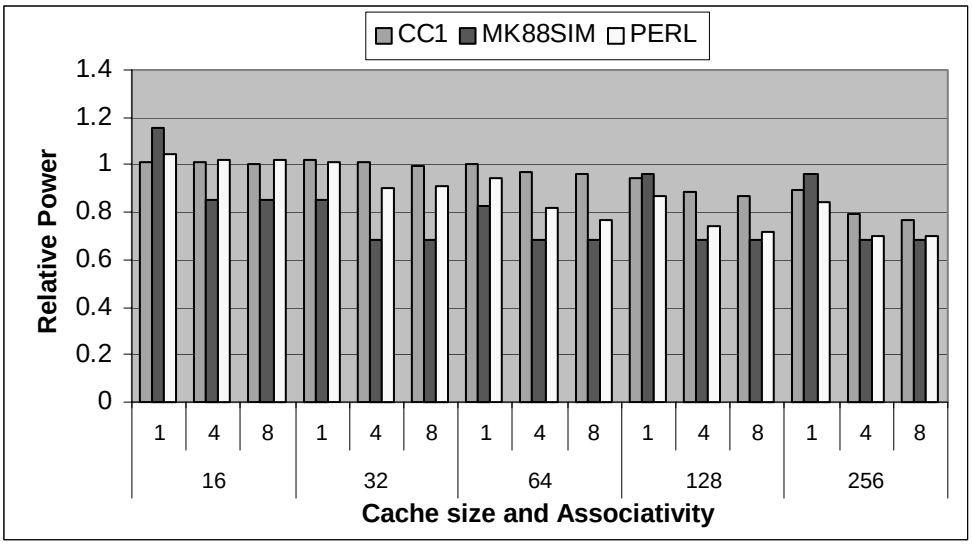


Figure 4.5: Relative power consumption of a filtered T$ vs I$

To summarize the performance and power comparison, one should have noticed that the simple model we are using does not take into consideration many important design parameters such as the fact that smaller traces are faster and may consume less power than larger caches. This is especially true if we count for the leakage power which is proportional to the area of the cache. So, believe that for many systems, the capability to reduce the size of the trace cache may bring even more benefit than we have presented here.

5 Conclusions and remarks

This paper made two important contributions: it improved our understanding on how traces behave within the trace-cache and proposed a new technique (which is based on profiling information) to improve the performance and power consumption of trace-cache architectures. Trace caches are suffering from sub-optimal area utilization. Different works [8][1] pointed on duplication of basic blocks within the trace-cache and suggested different techniques how to avoid it. In this work, we focus on a different phenomenon. We found that the majority of traces never being used again after being built and saved. We found out that these traces are the major source for trace cache inefficiencies. Since building and executing from the trace cache is much more expensive than just executing the same code directly from the instruction cache (we benefit only

from the reuse of the trace), we propose the use of profile based mechanism that can indicate which traces should be built (and put into trace cache), and which traces should be executed from the instruction cache instead.

This paper shows a new direction for trace-cache optimizations and estimates its potential for improving the power and performance of trace cache systems that suffer from high miss pressure. This pressure can result from large footprint of the program in respect to the physical size of the trace cache being used.

In the future we would like to examine the feasibility to classify the different traces at run-time, using hardware mechanisms. We hope that the combination of such hardware with software based profiling techniques can achieve even better results.

References

[1] B. Black, B. Rychlik, and J. Shen, "The Block-based Trace Cache", in *Proceedings of the 26th International Symposium on Computer Architecture*, May 1999.

[2] D. C. Burger and T. M. Austin "*The SimpleScalar Tool Set, Version 2.0*", Computer Architecture News, 25 (3), pp. 13-25, June, 1997

[3] D. Diefendorff, "HAL Makes Sparcs Fly", in *Microprocessor Report*, Volume 13, No 15, Nov. 1999.

[4] D. Friendly, S. Patel and Y. Patt, "Alternative Fetch and Issue Policies for the Trace Cache Fetch Mechanism", in *Proceedings of the 30th International Symposium on Microarchitecture*, December 1997.

[5] D. Friendly, S. Patel and Y. Patt, "Putting the Fill Unit to Work: Dynamic Optimizations for Trace Cache Microprocessors", in *Proceedings of the 31st International Symposium on Microarchitecture*, Nov. 1998.

[6] Q. Jacobson, "High-Performance Frontends for Trace Processors", Ph.D. Thesis, Department of Electrical & Computer Engineering, University of Wisconsin – Madison, Aug. 1999.

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

[8]S. Jourdan, L. Rappoport, Y. Almog, M. Erez, A. Yoaz, and R. Ronen, "eXtended Block Cache", in *Proceedings of the 6th International Symposium on High Performance Computer Architecture*, Jan. 2000.

[9] K. Krewell, "Quicktake: Willamette Revealed", *Microprocessor Report*, Feb. 2000.

[10] S. Manne, D. Grunwald and A. Klauser, "Pipeline gating: Speculation Control for Energy Reduction", in *Proceedings of the 25th International Symposium on Computer Architecture*, pages 132-141, June 1998.

[11] *Merten, M.C.; Trick, A.R.; George, C.N.; Gyllenhaal, J.C.; Hwu, W.W., "A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization"* in ISCA 1999, Page(s): 136 -148

[12] S.W. Melvin and Y.N. Patt, "Performance Benefits of Large Execution Atomic Units in Dynamically Scheduled Machines", in *Proc. of the 1989 Intern. Conf. on Supercomputing*, pages 427-432, 1989.

[13] 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.

[14] M. Postiff, G. Tyson and T. Mudge, "Performance Limits of Trace Caches", in *Journal of Instruction-Level Parallelism* No. 1, Oct. 1999.

[15] A. Ramírez, J. L. Larriba-Pey, C. Navarro, J. Torrellas, and M. Valero, "Software trace cache", in *Proceedings of the 1999 international conference on Supercomputing*, pages 119 – 126, 1999.

[16] A. Ramirez, J.L. Larriba-Pey and M. Valero, "Trace Cache Redundancy: Red and Blue Traces**",** in *Proceedings of the sixth International Symposium on High-Performance Computer Architecture 6*, pages 325-333, 2000.

[17] Droped for anonymousity "*Filtering Techniques to Improve Trace-Cache Efficiency"*

[18] E. Rotenberg, "Trace Processors: Exploiting Hierarchy and Speculation", Ph.D. Thesis, University of Wisconsin, 1999.

[19] E. Rotenberg, S. Bennett and J.E. Smith, "Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching", in *Proceedings of the 29th International Symposium on Microarchitecture*, Dec. 1996.

[20] M.S. Schlansker and B.R. Rau, "EPIC: Explicitly Parallel Instruction Computing", *Computer*, Vol. 33, No. 2, pages 37-45, Feb. 2000.

[21] J. Sahuquillo and A. Pont, "The Filter Cache: A Run-Ttime Cache Management Approach", in *Proceedings of the 25th EUROMICRO Conference,* volume 1, pages 424 – 431, 1999.

[22] T. Pabst, "Intel's New Pentium 4 Processor" in *Tom's Hardware Guide,* http://www.tomshardware.com/cpu/00q4/001120/index.html

[23] M. Upton, "The Intel Pentium® 4 Processor", in http://www.intel.com/pentium4, Oct. 2000.