IRWIN AND JOAN JACOBS
CENTER FOR COMMUNICATION AND INFORMATION TECHNOLOGIES

Usage of Trace Cache for Predicting Power Saving Opportunities

I. Sorani, A. Kolodny and A. Mendelson

CCIT Report #708
November 2008

DEPARTMENT OF ELECTRICAL ENGINEERING
TECHNION - ISRAEL INSTITUTE OF TECHNOLOGY, HAIFA 32000, ISRAEL

CCIT REPORT #708 November 2008

Usage of Trace Cache for Predicting Power Saving Opportunities

I. Sorani, A. Kolodny and A. Mendelson

Abstract

The use of Trace Caches is a well known technique to overcome the problem of limited instruction fetch bandwidth in modern computer architectures. A Trace Cache stores instructions dynamically, based on the order of their execution, rather than the in the order they appear and are compiled within the source program. Trace Caches were found to be especially effective for very wide machines as they improve instruction fetch bandwidth.

Previous research projects propose the use of relatively short traces to increase the probability of the correct execution of the traces, as well as to optimize Trace Cache memory space utilization. In this research, we investigate the possibility of using the Trace Cache to achieve a reduction of power consumption within the design.

Power saving is an important aspect of modern architectures and many different algorithms have been proposed towards this goal. One of the proposed power saving techniques is dynamic tuning of processor's resources, by turning off, for example, those units and resources that are not in use. Implementing this technique requires a monitoring unit that is able to predict such down-time and provide information about the resources, as potentially needed by the execution. We propose to use a Trace Cache for that purpose, as traces, particularly lengthy ones, contain information about future instruction execution. Therefore, we believe we can use this information to optimize the execution of long and frequent-used traces, and suggest optimizations that can lead to major power savings.

To benefit from dynamic optimization, we need to be able to predict instruction sequences early enough in order to allow changes to the trace, allowing sufficient lead time to make the changes. Therefore, we focus on predicting long instruction sequences that will enable prediction of the processor needs for longer execution time. We will explore the feasibility of making dynamic optimizations based on the information about these sequences. We propose an algorithm for the dynamic tuning of processor's resources, based on the information stored within the Trace Cache. In our proposal, we aim to achieve this with minimal performance penalty and additional hardware requirements.

This work provides an overview of Trace Cache proposals and enhancements over existing solutions, including an overview of several power reduction techniques. We propose a novel opportunity for using long traces as a potential to reduce processor power consumption by analyzing the behavior of instructions that are executed from the Trace Cache. Finally, we discuss the possibilities of using the proposed algorithm.

Abbreviations

IPC	Instructions per Cycle
CPI	Cycles per Instruction
TC	Trace Cache
IC	Instruction Cache
PC	Program Counter
FP	Floating Point

Conventions and Definitions

For the reader's convenience, we define terms and conventions used in this work.

Instruction fetch - The phase of an instruction cycle in which an instruction is read from memory.

Static instruction order (program order) – The order in which instructions are stored at the Instruction Cache, based on their placement by the compiler.

Dynamic instruction order – The order in which instructions are executed by the In-order processor as required by the program.

Basic Block – A group of non-branch instructions ending with a branch.

Trace - A sequence of basic blocks (possibly with repeated blocks) that contains no internal control flow and a single entry point. A trace usually starts at an address of a branch target, and ends when one of the termination conditions is met.

Trace termination conditions - The factor that most directly controls the quality and utility of created traces. In general, traces are composed of an integral number of basic blocks. We do not allow breaking basic blocks or splitting them between different traces, except for very exceptional cases such as a single very long basic-block. Other criteria can be the number of instructions or of branches and control instructions.

Trace line - The space in a trace-cache required to store a maximal-size trace (maximal number of instructions).

Trace Cache - A mechanism for increasing the instruction fetch bandwidth by storing traces of instructions that have already been fetched.

Trace Sequence – A sequence of several trace lines, not necessarily placed one after another at the Trace Cache, but fetched one after another from the Trace Cache without being interrupted by a fetch from the Instruction Cache.

Long Trace Sequence – A trace sequence which is built from a large number of instructions (e.g. more than 100) and therefore from a large number of trace lines.

Trace Prediction - A mechanism for predicting the trace line that will be executed next.

Trace Sequence Prediction - A mechanism for predicting a trace sequence that will be executed in the future (several trace lines that will be executed in the order in which they were predicted). Sequence Prediction is successful if we can predict n instructions (n is a

predefined number) and all of the predicted instructions were actually executed in the predicted order.

Trace Cache Configuration – is represented by two numbers (*n,m*). Where *n* is the maximal number of instructions that can reside in the trace (trace line size), *m* is the maximal number of branches allowed in the trace.

Dynamic Tuning – A process of modification in order to adjust the processor's resources to the current demands of the running application.

Power Consumption refers to the electrical energy that must be supplied over time to an electrical device to maintain its operation. Power consumption is usually a function of the power needed to perform the intended function of the device plus additional "wasted" power that is dissipated as heat.

1. Introduction

In the past the primary goal of general purpose design was performance, regardless of the power it may consume. Many research activities were conducted to improve the performance of superscalar architectures, such as enhancements to the front-end engine and execution engine. The execution engine was mostly enhanced by making the processor wider, using instruction parallelism techniques, adding more resources (e.g. functional units, registers etc.) and by having deeper speculation. Despite all the hardware enhancements, there are several problems that reduce the effectiveness of those enhancements. One of them is instruction fetch, due to the execution of long non contiguous instruction sequences that are stored in a different order at the Instruction Cache. Having reached a "power wall" we now face another problem - Power Consumption - which increases with the addition of more hardware resources. Power Consumption includes power wasted on hardware that is not being used. Eventually the power problem can be translated to performance problem as its solution often requires lowering the frequency.

One of the proposed hardware based optimizations to improve the effective fetch bandwidth was the Trace Cache [10],[13],[14]. Conventional Instruction Caches are not capable of fetching multiple blocks per cycle, because instruction sequences are not always in contiguous locations. Trace Caches overcome this limitation by caching the dynamic instruction stream into structure called "trace", so instructions that are kept non-contiguously at the Instruction Cache appear contiguous (based on their execution order) in the Trace Cache.

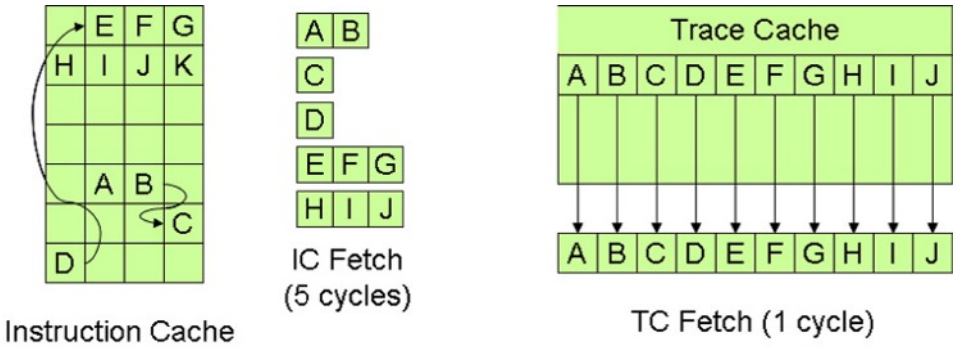


Figure 1: Trace Cache vs. Instruction Cache

Each line of the Trace Cache stores a trace of a dynamic instruction stream. A trace is a sequence of at most *n* instructions and at most *m* basic blocks starting at any point in the dynamic instruction stream. A trace is specified by a starting address and branch vector (a sequence of up to *m-1* branch outcomes that describe the path followed).

A line of the Trace Cache is filled as instructions are fetched from the Instruction Cache. If the same trace is encountered again in the course of executing the program, it is fed directly to the decoder. Otherwise, fetching proceeds normally from the Instruction Cache.

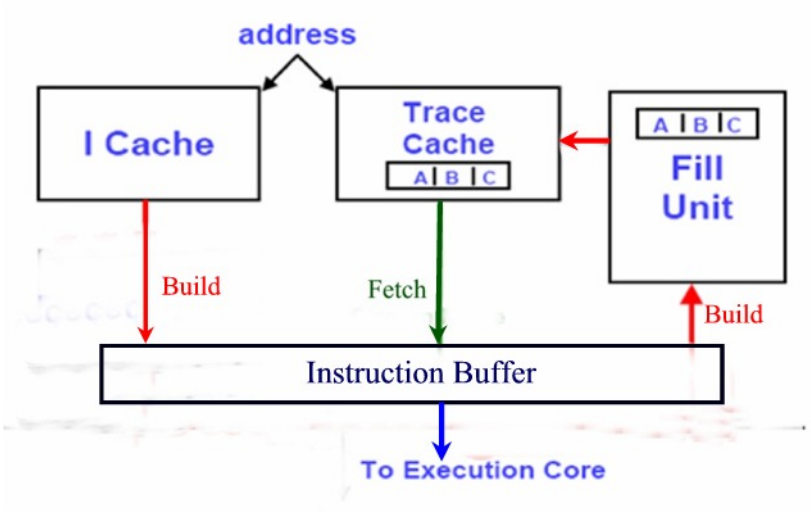


Figure 2: Trace Cache – Build and Fetch paths

Trace Caches are effective because most programs exhibit properties of temporal locality and easily predicted branch behavior. The downside of the Trace Cache is that it stores a lot of redundant data[14].

A Trace Cache was found to be most effective if its size is not limited, however small Trace Caches are more practical. Small Trace Caches are efficient in terms of power, since they are more predictable and have shorter access time. Most of the researches use Trace Caches with a trace line that contains up to 16 instructions and up to 3 branches.

Several papers have proposed using “hot traces” (traces that have been used repeatedly) to overcome the problem of the limited size of small Trace Caches and as a basis for various trace cache optimizations. The optimizations include, for example, reducing the power consumption of the core and gaining better performance given a limited-size cache [4],[14]. One of the proposed filters is a Sample Filter that improves the behavior of a small Trace Cache in terms of coverage and hit rate while the power of the fetch stage is reduced [2].

To benefit from continuous instruction sequences that are stored in the Trace Cache, we should be able to fetch traces from there without accessing the Instruction Cache between traces. To do this it is necessary to predict what the next trace will be. A straightforward method, and the one used in[14], is to predict all multiple branches within a trace simultaneously. Then, armed with the last PC of the preceding trace and the multiple predictions, the fetch unit can access the next trace. Another approach is next trace

prediction [8]. A next trace predictor treats the traces as basic units and explicitly predicts sequences of traces. It collects histories of trace sequences and makes predictions based on these histories.

We have presented the instruction fetch problem and the Trace Cache as one of its solutions. The second problem, as stated above, is processor power consumption. This has become one of the great challenges in designing high performance processors. The rapid increase in complexity and speed that comes with each new CPU generation causes greater problems with power consumption and heat dissipation. Traditionally, these concerns were addressed through semiconductor technology improvements, such as voltage reduction and technology scaling. Recently the focus on Power Consumption has increased and it is being taken into consideration in the very early stages of microarchitecture design.

Trace Cache was found to be very power efficient in case such as P4[6], where it saves decode energy and PARROT [15] when used for power optimizations.

Many different algorithms have been proposed for saving power, including turning off units when they are idle. Several researches proposed dynamic tuning of resources of a general purpose processor, according to the needs of the executed program [1][7] [11].
Bahar et al [1] proposed reducing power dissipation by adjusting the issue queue and execution units to the varying needs of the program. Ponomarev et al [11] proposed dynamically adjusting the sizes of the issue queue (IQ), the reorder buffer (ROB) and the load/store queue (LSQ) based on the periodic sampling of their occupancies.

The main idea of dynamic tuning is to reduce or close resources when the program does not require the full performance of the processor. In this case the processor can enter the low-power mode. When the program switches behavior again and requires more resources, the processor returns to normal operation. This technique is based on the observation that full resources are required only for a portion of the program's execution. For dynamic tuning we need a mechanism that will monitor the needs of the program. The monitoring scheme must be simple since the solution cannot be more power hungry than the problem. Also we do not want to sacrifice performance. Another requirement from the monitoring unit is the ability to predict changes at the processor's resources sufficiently ahead of time. Changes, such as disabling and enabling hardware resources, may take time. To achieve power reduction without lowering performance, we would like to identify places where some resources are not needed for a "long time". "Long time" is defined as enough time to disable the resource, benefit from power reduction while the resource is disabled and not consuming power, and enable the resource just before it is required without losing

performance. We can discuss time in terms of number of instructions that can be executed during this time. Therefore prediction of a processor's requirements enough time ahead can be translated into predicting enough instructions ahead.

In this work we check the feasibility of an architectural enhancement to reduce power consumption. The proposed enhancement is based on a solution that was originally developed for performance improvements. Until now Trace Cache was used mainly to increase instruction fetch bandwidth, we propose using it for power reduction purposes as well.

The main idea of our proposal is to use Trace Cache as monitoring unit for dynamic tuning of the processor's resources at run time. This can be done based on the dynamic information that resides in the Trace Cache. As long as we can predict instruction sequences coming from the Trace Cache that will be executed in the near future, we get all the information that we need. Knowing the next instructions that will be executed gives us knowledge about the processor's requirements and gives us sufficient time to prepare the processor for future requirements. For example, if the Trace Cache predicts that we will not have Floating Point (FP) instructions for a time that is long enough to disable the FP unit, save enough power and re-enable it for the next FP instruction, we can go ahead and disable it. As we need prediction for long time ahead we'll focus on prediction of long instruction sequences. Because our proposal is based on using data from the Trace Cache, the prediction and dynamic tuning will be relevant only for instructions coming from there. The prediction can be done by using a Trace Predictor and keeping additional information for each Trace Cache line.

The baseline of this work is existing processors with a Trace Cache, while we don't try to improve the Trace Cache, its predictor or its performance. Therefore we don't want to change the existing Trace Cache but use existing hardware with minimal additions. This proposal is orthogonal to changes that might be done to improve the Trace Cache and also relevant for future and improved systems that use a Trace Cache.

This work checks the feasibility of the above idea and proposes directions for future work.

Organization of this work:

The rest of this work is organized as follows.

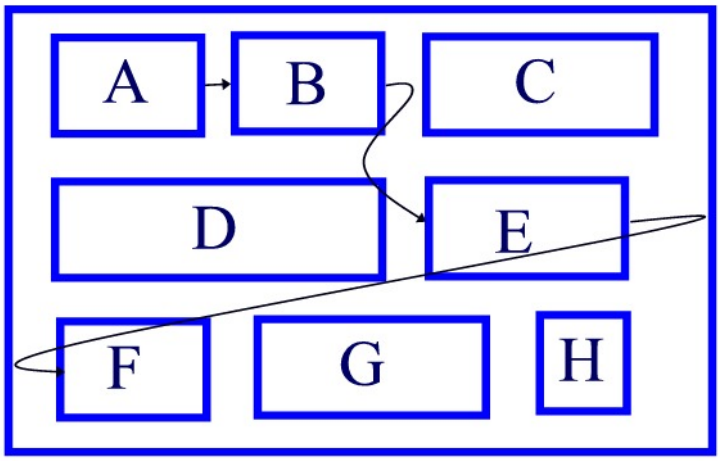
Section 2 presents an overview of related works and background for this work. The simulation environment used is presented in section 3. Section 4 describes basic

observations. The proposed algorithm and feasibility studies are described in Section 5. Section 6 discusses the possibilities of using the proposed algorithm and presents ideas for further improvements. In section 7 we conclude and propose ideas for future work.

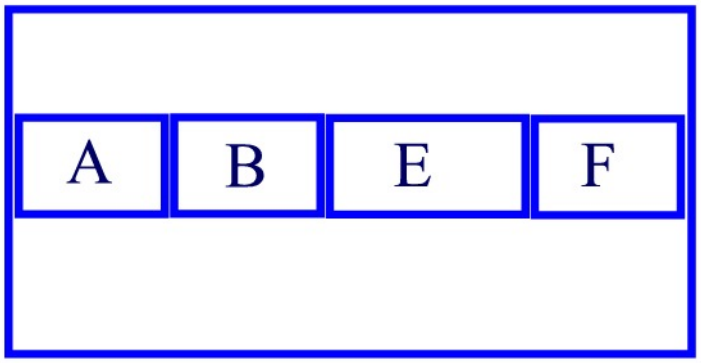
2. Overview of Previous works / Background

Trace Cache

As the issue width of superscalar processors increases, the importance of instruction fetch bandwidth increases as well. Several software (compiler optimizations) and hardware solutions were proposed in order to increase the effectiveness of the fetch mechanism. In this work we focus on a hardware solution called Trace Cache. Trace Cache keeps instructions in their dynamic order (traces) rather than in their static compiled order as they are stored in Instruction Cache.



(a) Instruction Cache



(b) Trace Cache

Figure 3: Storing a noncontiguous sequence of instructions

Figure 3 shows the structure of the Trace Cache versus the Instruction Cache.

A Trace Cache consists of traces. Each trace inside the Trace Cache is limited to n instructions (the trace cache line size) and m basic blocks, which mean m-1 branch instructions (branch throughput). A trace is specified by a base address (address of the first instruction in the trace) and the directions of its branches.

Trace Cache operation can be best understood by an example. Figure 4 shows a program's control flow graph (CFG), where each node is a basic block, and the arcs represent potential transfers of control (e.g. branches). In this figure, arcs corresponding to branches are labeled to indicate taken (T) and not taken (N) paths. The sequence ACE represents one possible trace which holds the instructions from the basic blocks A, C, and E. This would be the sequence of instructions beginning with basic block A, where the next two branches are not taken and taken, respectively. These basic blocks are not contiguous in the original program, but would be stored as a contiguous block in the Trace Cache. A number of traces can be extracted from the CFG. Four possible traces are:

1: ABC
2: ACE
3: CEA
4: EF
5: BBC
6: BBB

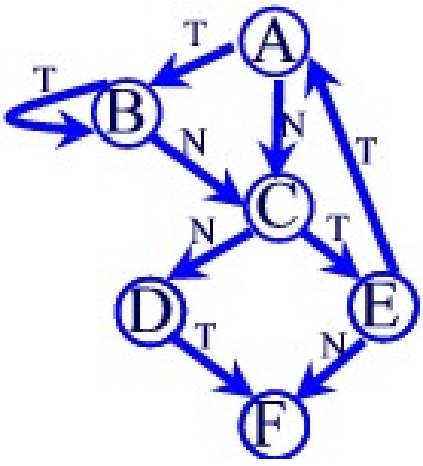


Figure 4: Example of control flow graph

Of course, many other traces could also be chosen for the same CFG, and, in fact, a trace does not necessarily have to begin or end at a basic block boundary, which further increases the number of possibilities. Also, note that in a Trace Cache, the

same instructions may appear in more than one trace. For example, the blocks A, C and E appear more than once in the above list of traces. However, the mechanism which builds traces always creates traces that begin and end on basic block boundaries.

Previous works deal with Instruction Cache hit rate and branch prediction accuracy, but there are a few additional problems that Trace Cache tries to solve:

- Branch throughput – the number of instruction fetched during one cycle depends on the number of branches that can be predicted in a single cycle (on average every fifth instruction is a branch).
- Noncontiguous instruction alignment - the continuity of the code can be broken by an unconditional jump, which makes it difficult to fetch more instruction in the same cycle.
- Fetch unit latency – latency might increase while trying to address the above problems.

These problems can be solved using both software and hardware approaches. On the hardware side we find techniques like the branch address cache, the collapsing buffer or the trace cache. On the software side, we find instruction scheduling techniques and code reordering approaches like the software trace cache.

The Trace Cache mechanism deals with the problems stated above as it allows fetching of nonconsecutive basic blocks coming from different instruction cache lines (noncontiguous instruction alignment). It captures the dynamic instruction stream: instruction sequences and the branch directions which lead to them. If the same starting instruction will be selected again in the future and if we predict the same branch outcomes, the whole instruction trace can be fetched from the Trace Cache. We can fetch several basic blocks without additional processing thus avoiding the fetch unit latency. When there is no trace that meets the demands the instructions will be read from the regular Instruction Cache.

2.1.1 Trace Cache Build and Access

Traces are built during fetch of instructions from the Instruction Cache. The instructions are inserted into a buffer in the order they are accessed, till the termination conditions are met. A directions vector is built in parallel in order to recall the branch directions which were used in the trace. Eventually the trace is copied from the buffer to the Trace Cache and saved there (usually this is done by Fill Unit –

see Figure 5).

The trace can later be accessed either solely by its base address or by a combination of the base address and its branch directions vector. The Trace Cache may choose to allow two traces with the same base address to be saved at the same time. In this case, a combination of the base address and its branch directions vector will be required to access the trace.

When a trace is fetched from the Trace Cache the PC stops influencing the instruction fetch stage (the PC is still calculated for recovery reasons). As a branch instructions in the trace resolves its conditional expression it is compared to the directions vector of the trace to see if the trace took the branch correctly. When a misprediction is detected, the machine needs to roll back to the mistaken branch and look for a new trace (or build it) from that point.

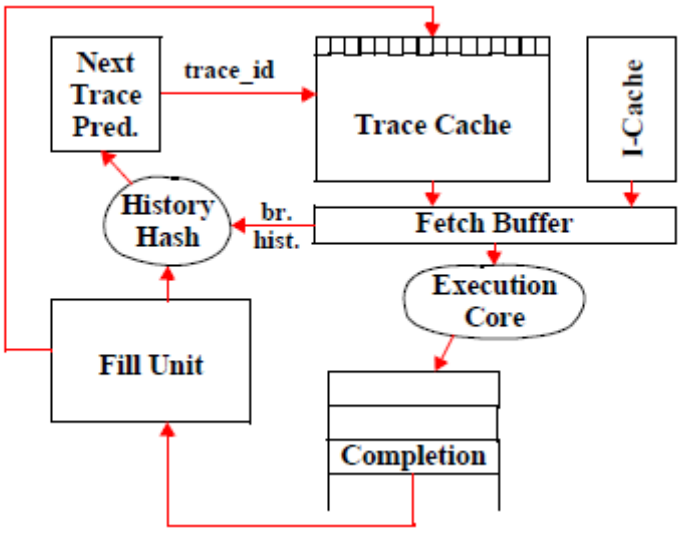


Figure 5: The conventional Trace Cache – block diagram

Several design parameters should be considered as part of the definition of the build and access mechanisms, as they can affect Trace Cache functionality:

- **Termination conditions**, or when should the building stop. Aside from the obvious cases of reaching the instruction and branch limits for each trace (maximal number of instructions and branches), there could be some other situations in which the machine would do better by not continuing the trace. When an indirect branch is used, the trace cache mechanism is not able to check the correctness of the target address. If no mechanism is added to handle such cases, the trace must end. Another

example is function calls, where it might be wise for a new function to start at a new trace, thus increasing the chances of this trace to be reused. Another idea could be to try and finish traces at a branch instruction when possible (a basic block). Doing so may create traces that will have better usage, although it may result in more unused data areas in the Trace Cache. Some Trace Caches may choose not to continue a trace after a taken backward branch (in order to have only one copy of a loop body), or to continue building only if the remaining trace will be able to contain a whole loop body (to create loop unrolling).

- **Multiple traces with same base address**. When saving more then one trace for a base address the Trace Cache allows more flexibility. However, it may become more complex to access the trace. Therefore we should decide whether to allow multiple traces with the same base address and how many of those are allowed.

- **Adding traces to the Trace Cache**. A trace is being built in a buffer during instruction fetch. The buffer can transfer traces to the Trace Cache when it finishes building them, but they might not be valuable traces, as they are based on accessed, not committed, instructions. Copying the traces after they commit could give the machine better confidence in the trace. Doing so would require a more complex control unit. A different option to increase the confidence of the saved traces could be adding some filters in order to have only selected traces in the cache (will be discussed later).

- **Trace Cache access**. In most works dealing with Trace Cache a specific trace is searched in the Trace Cache when accessing it. In case we have multiple traces with the same base address we will access the Trace Cache with both the base address and the branch direction vector. This requires predicting the next few branches. Such prediction could be done by a regular branch prediction mechanism designed to predict few branches forward based on its own predictions. Another possibility is a special trace prediction mechanism that tries to predict the next trace as a whole according to previous traces which were used (will be discussed later).

2.1.2 Trace Cache advantages and limitations

The Trace Cache approach relies on two principles:

Temporal locality – the property that instructions which have been recently used will be used again in the near future. Therefore if the Trace Cache consists of repeatedly used traces we stand to gain in the instruction fetch stage.

Branch behavior - most branches tend to be biased towards one direction, which is

why the accuracy of branch prediction is usually high. Therefore having a trace with the most common branch prediction will benefit us most of the time, and the loss from mispredictions will be relatively negligible.

It is possible to make the Trace Cache power efficient on CISC machines (e.g. Intel's Pentium 4 [6]) by saving decoded instructions. We might choose to save decoded instructions in the Trace Cache, since addresses are not effective within a trace, and we can use the trace even if the actual instruction code changes. By doing so, such machines are able to save much time and power when fetching from the Trace Cache by not having to decode the instructions once more.

Example: In P4 by Intel, there is a Trace Cache that can hold up to 12K uops (decoded instructions), the decoding itself takes 2 stages in P4, and when fetching from the Instruction Cache (L2) only one instruction can be fetched per cycle (as opposed to 3 uops in the Trace Cache). Fetching more than one CISC instructions is difficult due to the fact that the instructions come in different lengths; the uops are RISC-like instructions and come in a fixed size, thus by fetching from the Trace Cache the P4 need not decode the instructions and can fetch more instructions per cycle.

In order to measure Trace Cache performance, metrics other than the common metrics of hit rate and IPC have been defined. These are fragmentation, instruction duplication, efficiency, indexability, and retirement rate [12]. Since performance is the reason for having a Trace Cache in the first place, IPC must be the metric of choice in determining the best configuration.

The limits of the Trace Caches are presented by:

- **Duplication** – a measure of how efficiently the "unfragmented" storage in the Trace Cache is used. Duplication is a consequence of the method of indexing the Trace Cache and is really an intended side effect. In a Trace Cache, the instruction address is used along with branch prediction information to identify a trace, so a given block may begin a trace and also appear as an interior member of many traces in the Trace Cache.
  Examples for duplication (according to Figure 6):
  - Traces that start from same address (A): ACG, ACD
  - Same building block in several traces (C): EAC, ACG
- **Fragmentation** – a measure of storage utilization which describes the portion of the Trace Cache that is unused because of traces shorter than the maximal number of instructions in the TC line. It is essentially wasted storage.
  Examples for fragmentation (according to Figure 6) are ACG, ACD.
- **Indexability** – miss rate due to searching for an address that doesn't start a trace, although it is an interior member of the trace.
  Examples for indexability (according to Figure 6) are D, G.

Those metrics mostly reflect the problem in utilization of the Trace Cache's memory space.

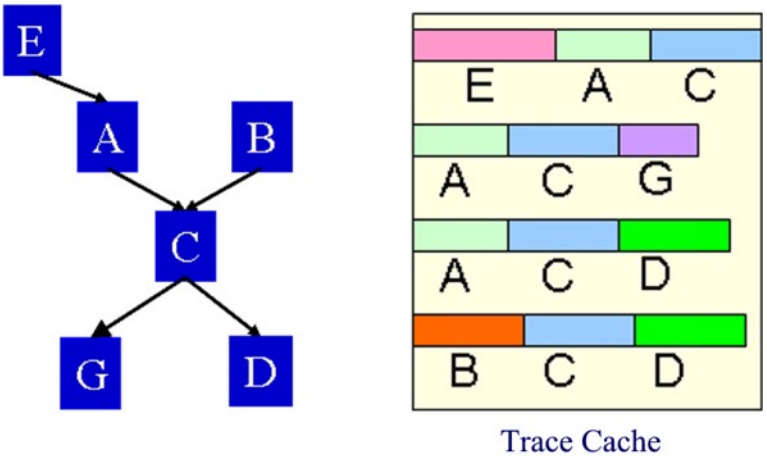


Figure 6: Trace Cache Limitations

The common use of the Trace Cache is to increase instruction fetch bandwidth and resolve additional fetch related issues, as stated above. We propose a different

direction, which is to use the Trace Cache for power reduction, based on the information that already resides in the Trace Cache or can be collected from it.

Trace Cache line Predictor

The best mode of work would be achieved by fetch from the Trace Cache only and without fetch from the Instruction Cache. Therefore it's important to define a mechanism for fetching the next line from the Trace Cache. A Trace Cache line can be identified by both the base address and branch direction vector. In order to predict the next line we need to predict the next few branches. Such prediction could be done by a regular branch prediction mechanism designed to predict a few branches forward.

A number of methods for fetching multiple basic blocks per cycle from Instruction Cache have been proposed, based on multiple branch prediction. The predictor generates multiple branch predictions while the Trace Cache and Instruction Cache are accessed. The fetch address is used together with the multiple branch predictions to determine if there is a trace in the Trace Cache that matches the predicted sequence of basic blocks and if there is a hit. Alternatively, the Trace Cache can be designed to signal a hit if the fetched trace only partially matches the requested path. During each fetch cycle the predictions made by the branch predictor are used to select which blocks within the trace that was fetched from the cache will be issued to the core. If a trace cache line contains ABC, and the predictor predicts ABD, blocks A and B are supplied. If the predictor selects AC, as can happen with some control flows, only A is supplied. This technique is called partial matching [4]. The use of partial matching allows requests to hit in the Trace Cache more frequently and results in better usage of the Trace Cache.

Another option is a special trace prediction mechanism designed specifically to work with Trace Caches. It tries to predict the next trace as a whole according to previous traces which were used [8].
The trace predictor differs from a multi branch predictor in the way it relates to a trace as the basic unit rather than the branch instruction. The trace predictor explicitly predicts the next trace according to the traces which were fetched before. It collects histories of trace sequences and makes predictions based on these histories. This is similar to conditional branch prediction where predictions are made using histories of branch outcomes.
The identifiers of previous traces (base address plus a branch directions vector)

represent a path history that is used to form an index into the prediction table. The last *N* trace-identifiers are hashed via a hashing function while the results are saved in a special history buffer (*N* hashed IDs). When trying to predict the next trace, the hashed IDs are combined by an index generator to form an index. This index is used to access the prediction table. A two bit counter is attached to each entry to provide confidence, for a successful completion of a predicted trace the counter is increased by 1, for a mistake it is decreased by 2. When the counter reaches zero the trace is removed from the prediction table and the actual trace will replace it.

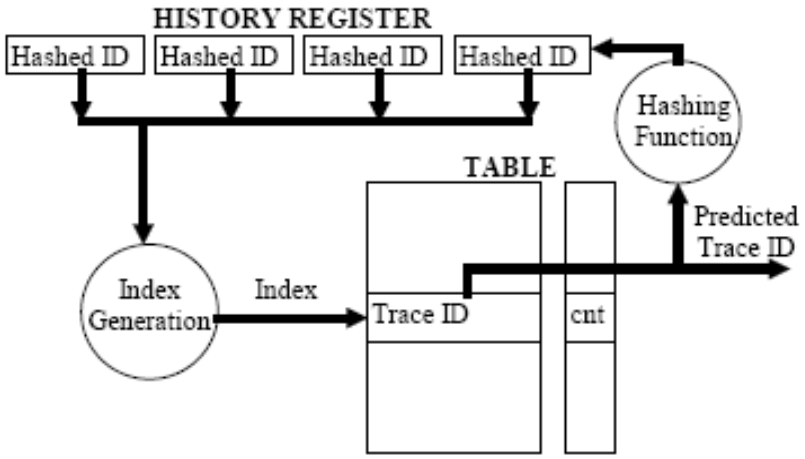


Figure 7: Next Trace Predictor

Trace Cache Filter

It has already stated above that the Trace Cache can contribute more than an increase of instruction fetch bandwidth. It can also contribute to reducing power consumption by keeping the instruction traces in decoded format and consuming power only when building a trace. It has been proposed by Rosner et al.[14] that the functionality of the Trace Cache can be divided into trace-building, which builds and fetches the traces, and trace-bookkeeping, which maintains the Trace Cache and tries to avoid unnecessary rebuilds. While the Trace Predictor improves trace-building functionality, the Trace Cache filter affects the functionality of the trace-bookkeeping and tries to improve the Trace Cache organization and utilization.

It was observed by Rosner et al. [14] that:

- The majority of the traces are rarely used. Many traces are built only once.

- The majority of the executed instructions come from the more frequently used traces.
- A good replacement mechanism has a significant effect on reducing the build-rate (affects both performance and power.)

Those behavioral patterns indicate that filtering techniques based on the frequency of trace usage can reduce the trace-build rate. Therefore, filtering can increase fetch bandwidth and reduce power consumption. The filter can increase the utilization of the trace cache by preventing the infrequently used traces from polluting the trace-cache and keeping the "good" traces. This way we'll have "good" traces ready in the Trace Cache for fast delivery and we'll prevent expensive rebuilds.

The trace population can be divided into two groups: the "hot traces" and "cold traces". "Hot traces", traces that have been used repeatedly, are a small number of traces responsible for the majority of the instructions executed. Most of the traces are "Cold traces" that are rarely executed. It follows the "80/20 principle": a small fraction of the static code is responsible for most of the dynamic code.

A "good" Trace Cache should have a replacement algorithm that helps the cache population consist mainly of "hot traces". It should also have a mechanism that prevents "cold traces" from entering the cache. "Cold traces" might cause the eviction of "hot traces," thus reducing the hit rate of the cache and increasing the number of builds, which waste power. Another advantage of keeping "hot traces" is that their number is usually limited, so they can fit well into a small Trace Cache and allow most of the code to be executed from the Trace Cache.

Various SW- and HW-based filtering mechanisms have been proposed using the "hold/cold traces" principle [2][9][14] .
Kosyakovsky et al. [9] proposed software-based profiling to identify "hot traces". The traces are classified into four types according to the frequency they have been accessed during the sampling interval. Based on those classifications the SW supplies hints to hardware about which trace should be built and stored in the Trace Cache and which should be executed from the Instruction Cache. This technique requires ISA changes (add hints).

Rosner et al. [14] presented a HW approach which separates the Trace Cache into two mutually exclusive parts. Filter Trace Cache (FTC) and Main Trace Cache (MTC). The FTC is used for filtering (monitors traces behavior), while the MTC stores traces that have been selected by the filtering process ("hot traces").

Instructions are fetched from the instruction cache, decoded, and built into traces. Built traces are entered into the FTC. Useful traces are moved to the MTC for longer term storage. Other traces are discarded from the FTC. The decision to discard or promote a trace is made based on a filter that implements some heuristic. The heuristic is based on the number of times this trace was accessed. The assumption is that a useful trace will continue to be accessed repeatedly, therefore it gets promoted to MTC. Otherwise, it's discarded on the grounds that it is likely to be evicted from the MTC shortly after being promoted and before being used again.
The downsides of this proposal are that each access to the Trace Cache requires access to both parts and leads to higher power consumption, while only part of the Trace Cache (MTC) is used to store "hot traces", making the Trace Cache effectively smaller.

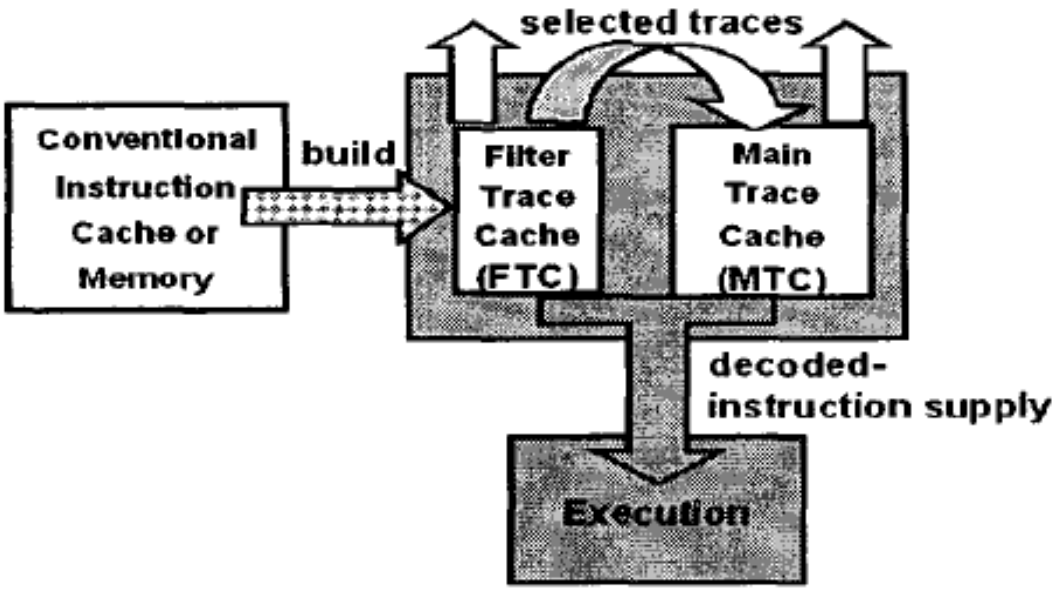


Figure 8: FTC and MTC filtering system

The Sampling Filter, as proposed by Behar et al. [2], is based on the observation that most of the builds are of "cold traces". Those traces have a small chance to be accessed again before being evicted from the cache, while "hot traces" reside fairly longer in the cache and are occasionally disturbed by the "cold traces".
Normally every trace that is built is inserted into the cache. The Sampling Filter is an entrance filter that would prevent "cold traces" from entering the cache. It allows only a fraction of the potential traces to be built and enter the cache. This filter uses a

statistical approach and selects traces to be inserted into the cache randomly on periodic basis (without prior knowledge and without performing any bookkeeping on the traces). Traces that are not selected are discarded. Using this approach we reduce the number of "cold traces" in the Trace Cache.
The filtering algorithm can be tuned with different sampling rates (the rate at which traces are sampled and inserted into the cache).

The advantages of this proposal are that this filter requires minimal hardware for the implementation, it improves the Trace Cache utilization by "hot traces" and reduces the number of trace builds and the power associated with those builds.
The downsides of this proposal are a longer learning (adjustment) period and that for each sampling rate there are some access patterns that might be destructive.

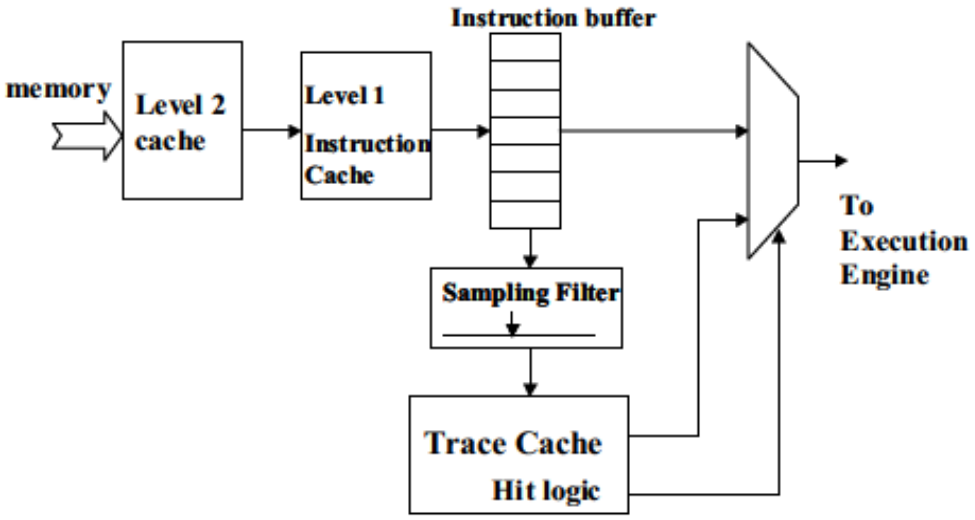


Figure 9: Trace Cache with Sampling Filter

Power Reduction Mechanisms

Performance has been a main goal to achieve in superscalar architecture. This is mainly achieved by exploiting instruction-level parallelism. Performance improvements often translate into having multiple execution units that can accommodate a large variety of instruction mixes. In order to support different execution capabilities and to achieve a significant speedup, the conventional instruction path needs to include wide issue buffers to allow a larger number of in-flight instructions and wide instruction paths to feed all available execution units. All those require more hardware, which leads to higher power consumption.

Traditionally, performance concerns have taken priority over energy costs or power consumption. Power efficiency has been addressed mainly at the technology level, through lower supply voltages, smaller transistors, better packaging, etc.

Lately there is a growing focus on reducing the power consumption as we are approaching a “power wall”. Power dissipation of microprocessors is becoming an important concern for designers. Low-power design has become an active area of research, but the power problem should also be addressed at the microarchitectural level.

Modern superscalar microprocessors are designed to achieve the best performance for large number of targeted applications (benchmarks), resulting in the permanent allocation of resources to maximize performance across a wide range of applications. It has been observed that there is wide variation in processor resource usage among various applications. In addition, the execution profile of most applications indicates that there is also wide variation in resource usage from one section of an application’s code to another. High-end configurations also tend to have high energy consumption partly due to power consumption of unused modules. The ideal would be to identify the right configuration which optimizes the energy consumed per each region of code.

Several works have presented changes at the microarchitectural level for control and scaling of resources to address the issue of power consumption.
The technique called Pipeline Balancing (PLB) proposed by Bahar et al.[1], dynamically tunes the resources of a general purpose processor to the needs of the program by monitoring performance within each program. The goal is to determine the changing needs of each program and tune processor resources to the program with the aim of reducing power dissipation. The PLB algorithm is quite simple. There is a monitoring unit that monitors the issue and execution needs of the program. When the program does not require the full issue and execution capabilities of the processor, the issue width of the processor is reduced, some execution units are disabled and it enters a low-power mode. When the program switches behavior again

the processor returns to normal operation.

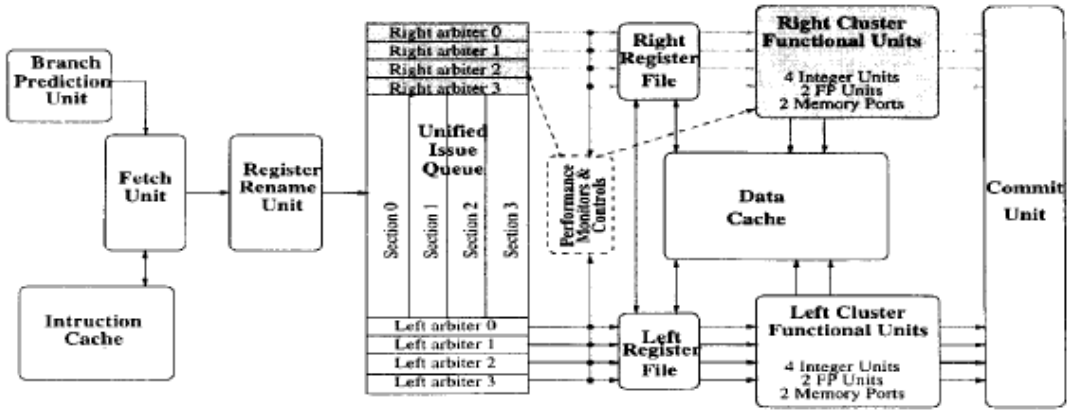


Figure 10: Pipeline organization of the processor

The basic assumption of the PLB algorithm is that past program behavior indicates future program needs. It predicts the future resource needs of the program based on past program characteristics such as issue IPC (for integer and FP instructions). These values are measured over a fixed sampling window. The requirement of the PLB is that the monitoring scheme must be simple, since the solution cannot be more power hungry than the problem, and PLB cannot sacrifice performance to reduce energy. PLB saves energy by modifying the activity rate in the issue queue and execution logic when the program doesn't require them. Implementing this technique requires dedicated HW (counters, comparators, FSM).

Ponomarev et al. [11] proposed changing the "one-size-fits-all" philosophy in allocating datapath resources as it results in resource over-commitment. They proposed a mechanism that dynamically adjusts the sizes of the issue queue (IQ), the reorder buffer (ROB), and the load/store queue (LSQ) based on the resource demands of the executing program.

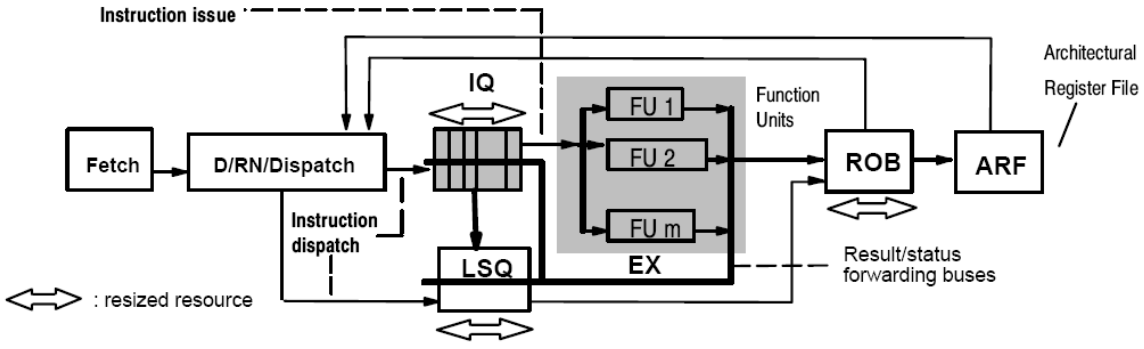


Figure 11: Superscalar Datapath

The resources of those components are partitioned and the number of active (i.e., powered up) partitions are chosen dynamically by tracking the actual demands of the program. The IQ, the LSQ, and the ROB are controlled independently. Resources can be downsized if their occupancies are low by deallocating one or more partitions and turning them off. As the resource demands of the application go up, deactivated partitions are turned back on to avoid any impact on performance. Downsizing is done based on periodic sampling of the occupancies of each component. Upsizing, on the other hand, is done more aggressively in order not to harm the performance. The hardware requirements for this mechanism are simple. The proposed approach attempts to track closely the dynamic demands of an application and therefore tries to allocate "just the right amount of resources at the right time" to conserve power. In this work again the decision is based on the past behavior. This technique has several constraints: it requires dedicated hardware that can be partitioned, transition period is required for pointer adjustment.

Another technique based on run-time profiling was proposed by Iyer et al.[7]. It is based on detection of hotspots - several small critical regions of code in which a program spends most of the execution time.

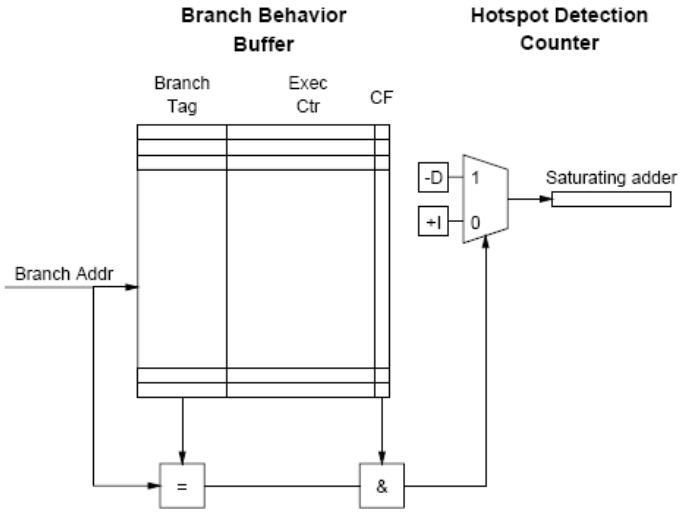


Figure 12: Hotspot detection hardware

This technique is based on identifying these hotspots in a program at run-time, characterizing each hotspot in terms of its power usage and arriving to the energy-optimal configuration of resources for the processor. The hotspots are identified by

finding the most frequent branches.
To determine the optimum configuration for each hotspot power register was used. It maintains power statistics for the four units of the processor that were defined as most power-hungry: floating point ALU, integer ALU, register file and instruction window. There is predefined number of possible configuration and the goal is to find an optimum configuration for each hotspot. When a hotspot is detected the power register is sampled each 1024 instructions and then the configuration is changed. After the optimum configuration is found, it is stored inside the hotspot table. The table contains one entry for each hotspot. The next time the same hotspot is encountered, the optimal values can be taken from the table.
This technique requires additional dedicated hardware used for "hotspot" detection and power profiling. Using "hotspots" for power optimization reminds our proposal, as we are looking for regions of code that are frequently executed.

The main idea of the above works is to propose microarchitectural solutions to make the processor more energy efficient. They propose scaling the processor's resources dynamically and adjusting them to the demands of executed code in order to have a more energy-optimal configuration of resources for the processor. Those changes should be done with minimal impact on performance. As the decision is done during run-time it requires a monitoring unit that should track changes in the code. The works stated above use different monitoring units and propose different dynamic changes of various resources in the processor.
In all the above proposals additional hardware is used for monitoring the executed code. In this work we want to propose another microarchitectural solution to the power problem based on dynamic scaling of a processor's resources during run-time. We attempt to reduce the power consumption while the performance is retained, based on existing hardware. As stated before, a Trace Cache can contribute to power reduction of the FE (by storing decoded instructions). We want to propose additional way for power reduction and use the Trace Cache as a monitoring unit for dynamic scaling of processor's resources.
A Trace Cache stores the instructions in their execution order. A good Trace Cache will have most of the executed code in execution order. Therefore it might have the data about future demands of the processor. If we can derive and use this data wisely we can fit the processor's resources to the exact demands of the executed application. We can attempt to identify the best configuration for the processor to optimize the energy consumed per instruction for each region of code. All the data we

need about application behavior can be derived from the Trace Cache. Therefore for processors that already use the Trace Cache no additional hardware is needed for monitoring. The optimizations that can be done based on the data stored at the Trace Caches will be discussed later.

3. Simulation Environment

The performance numbers presented below are based on an extended version of the sim-outorder simulator from the SimpleScalar tools set 3.0d [3]. The configuration used for all simulations is the default configuration of Simple Scalar. The main parameters are listed below in Table 1.

We use 10 benchmarks from the Spec2000 benchmark suite [5] with different inputs (total 20 benchmarks). The benchmarks and inputs are listed in Table 2.

The SimpleScalar simulator was augmented with a detailed model of the Trace Cache along with the Next Trace Predictor and Sampling Filter as implemented by Michael Behar [2].

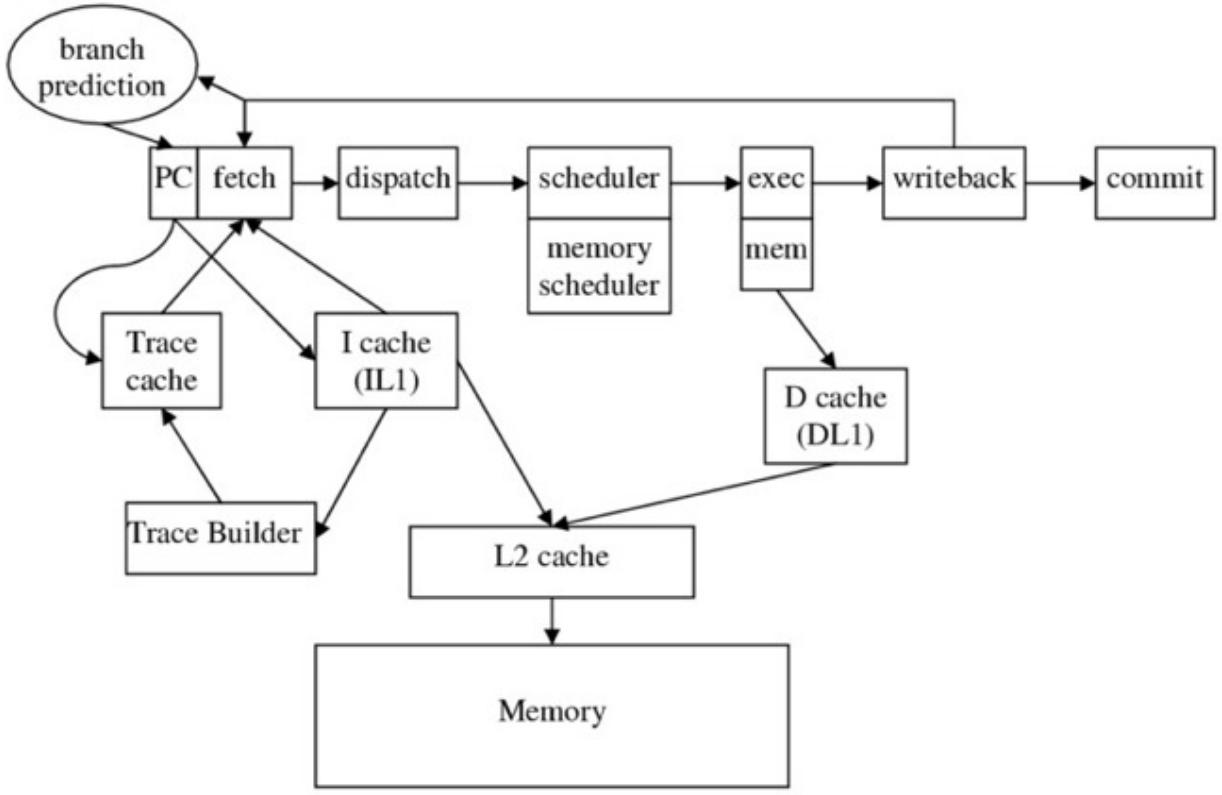


Figure 13: SimpleScalar Simulator with Trace Cache

Execution Engine	
Decode, Issue, Commit width	4
Functional Units	INT ALU: 4
	INT MUL/DIV: 1
	FP ALU: 4
	FP MUL/DIV: 1
Fetch queue size	4
Register update unit (RUU)	16
Load Store queue (LSQ)	8
Memory	
L1 Data Cache	Size:16KB Associatvity: 4 ways Block Size: 32B Replacement Policy: LRU Latency: 1 cycle
L1 Instruction Cache	Size:16KB Associativity: 1 way Block Size: 32B Replacement Policy: LRU Latency: 1 cycle
Trace Cache	Size:1KB Associativity: 2 ways Block Size: 16B Replacement Policy: LRU Latency: 1 cycle
L2 Unified cache	Size:256KB Associatvity: 4 ways Block Size: 64B Replacement Policy: LRU Latency: 6 cycles
Branch predictor	
Predictor	Bimodal 2K entry
Return Address Stack (RAS)	8
BTB	2K entry, 4 ways
Next Trace Predictor	HYBRID

Table 1: SimpleScalar Parameters

We used the following benchmarks from the SPEC2000 Benchmark Suite [5]:

Benchmark	Input
179.art	A10.img
164.gzip	input.source
164.gzip	input.log
164.gzip	input.graphic
164.gzip	input.random
164.gzip	input.program
175.vpr	Place
175.vpr	Route
176.gcc	166.i
176.gcc	200.i
176.gcc	Integrate.i
176.gcc	scilab.i
181.mcf	inp.in
256.bzip2	input.source
256.bzip2	input.graphic
256.bzip2	input.program
183.equake	inp.in
255.vortex	lendian1.raw
300.twolf	Ref
188.ammp	ammp.in

Table 2: Benchmarks

Trace Cache at SimpleScalar

Several parameters can affect the performance of the Trace Cache:

- Trace Cache line size
- Can we have traces with same base address
- When should we stop building the trace
- Can a trace be aborted in the middle
- How it's being accessed, etc.

In this work we use traces that are constructed in the front-end, so wrong speculative traces are allowed. Each trace can contain up to *m* instructions and *n* direct branches (*m, n* are simulator parameters). 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. Each trace is identified by Trace ID which consist of the address of the first instruction in the trace and a branch direction vector (each bit is the direction of a branch. 0 for not taken and 1 for taken).

The Next Trace Predictor is used for trace prediction and is responsible for supplying the correct trace ID. A Sampling Filter is used to improve the Trace Cache's hit rate. Concurrent access between the Instruction Cache and the Trace Cache (CTC) was modeled.

The simulator can be at one of the following states with regard to the Trace Cache: Build, Read or Trace Fetch.
Each instruction that is fetched from the regular Instruction Cache is entered into the trace builder buffer and attached to a trace. In this situation the machine is in Build state. 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 (as we use small Trace Caches the penalty associated with misses would be too high in sequential mode).
When a trace ends for any of the reasons mentioned above (reached end conditions) it is sent to the Trace Cache without checking if it fully commits. At this point the filter may still prevent it from being actually saved. The simulator moves from Build state to the Trace Fetch state.
During the Trace Fetch stage the current program counter along with branch directions vector is checked in the Trace Cache for a hit. If there is a hit, a trace line is supplied, otherwise instructions are fetched from the Instruction Cache or the Memory and the simulation moves to the Build state. The Trace Cache continues to feed the fetch unit until the trace ends or until there is a misprediction in the pipeline. This is called a Read state.
The Trace Cache gets feedback from the writeback stage about whether the branches were correctly predicted. It gets feedback from the commit stage to report which traces have fully committed. In order to support these feedbacks each instruction should have an indication of the trace it belongs to. In case of branch misprediction during the Read stage there is a recovery process and the simulator moves to the Trace Fetch state. In case of misprediction the part of the trace until the branch that caused the misprediction will be executed and the rest will be discarded.

The Trace Predictor is updated at the commit stage with the information if the trace prediction was correct (trace was fully committed) or incorrect and updates the predictor counters for this trace accordingly.

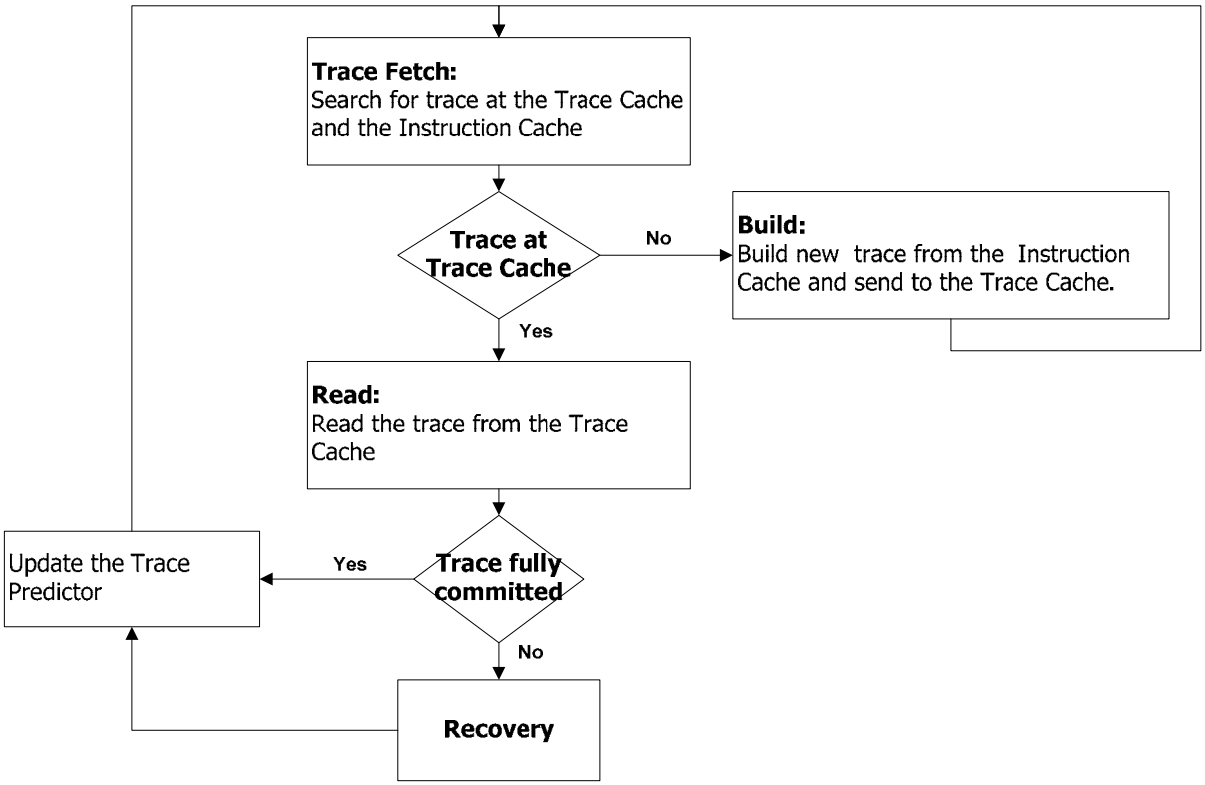


Figure 14: Access to the Trace Cache

Figure 14 shows different states with regard to the Trace Cache and transitions between them.

4. Basic observations

In this work we want to propose a novel usage for Trace Cache. Until now the main usage of the Trace Cache has been to increase the instruction fetch bandwidth. The usage we want to propose is based on the statistical information that already resides there and can be retrieved with minimal addition of HW. The Trace Cache keeps instructions in their execution order. In each fetch of a Trace Cache line we receive up to *n* instructions (where *n* is the size of Trace Cache line) that will be executed next. In other words, in each trace line fetch we can predict up to *n* instructions and their characteristics. We propose using this prediction for dynamic tuning of the processor's resources and for regulating the Power Consumption of the processor.

Our proposal is based on having a prediction for sequences of instructions that come from the Trace Cache; therefore we need to check that enough instructions come from the Trace Cache (based on given configuration) to make this proposal significant.
In order to simulate basic observations we used a 16:3 configuration of the Trace Cache. This configuration allows up to 16 instructions in the trace and up to 3 branches in the trace.

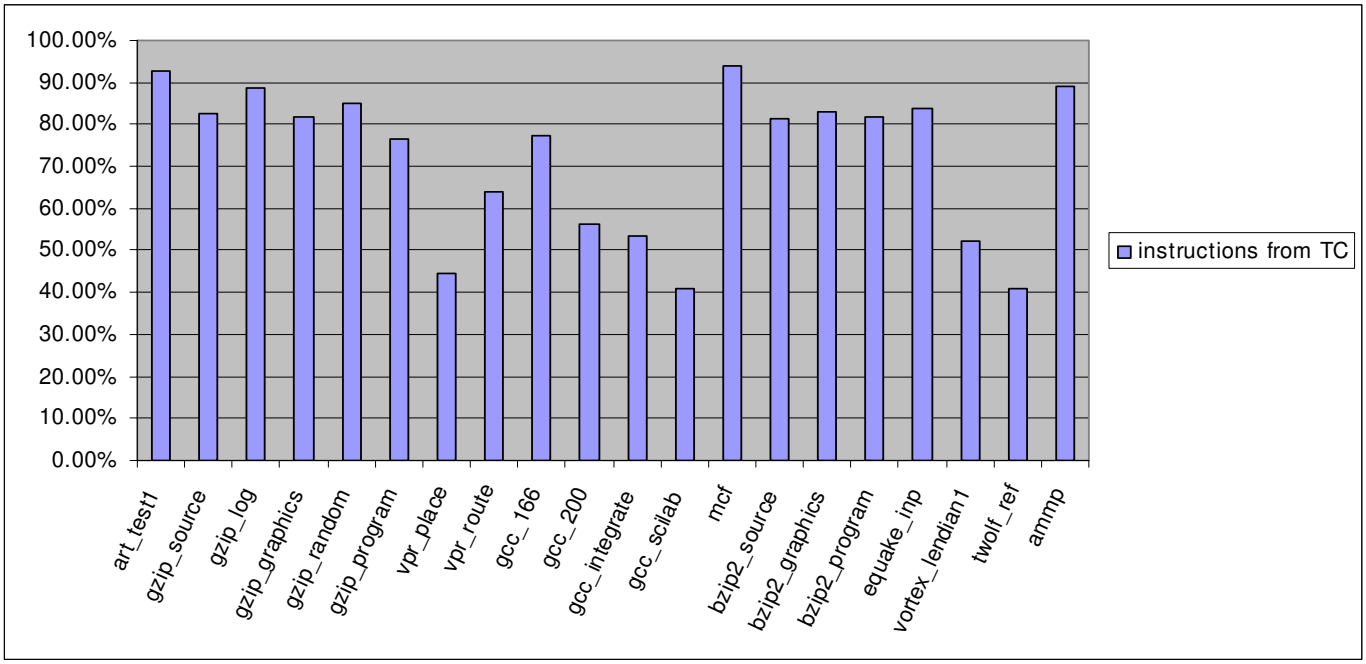


Figure 15: percentage of instructions coming from TC vs. IC (16:3)

Figure 15 shows the percentage of instructions that come from the Trace Cache, while the rest come from the Instruction Cache based on the 16:3 configuration. We can see that for most of the benchmarks the majority of the instructions come from the Trace Cache and not

from the Instruction Cache, or at least we have a significant amount of instructions coming out of the Trace Cache during the program.
If there is a miss at the Trace Cache the instructions will be fetched from the Instruction Cache. In this case the instruction fetch bandwidth will be lower and we may want to move to a low-power configuration in order to reduce the Power Consumption instead of using the full capabilities of the processor.

During instruction fetch from the Trace Cache we can fetch up to *n* instruction (where *n* is the size of the trace line), therefore while fetching a trace we expect to predict the characteristics of up to *n* instructions ahead. This data will be relevant only if the instructions that reside in the trace will be actually executed. We define "trace execution quality" as the ratio of executed to stored instructions in the trace. The prediction about the trace information will be relevant only if the trace execution quality is high. Otherwise we can't change the processor's resources based on such a prediction.

Benchmark	Input	Trace Execution Quality
179.art	A10.img	0.96
164.gzip	input.source	0.95
164.gzip	input.log	0.96
164.gzip	input.graphic	0.94
164.gzip	input.random	0.95
164.gzip	input.program	0.94
175.vpr	Place	0.95
175.vpr	Route	0.99
176.gcc	166.i	0.97
176.gcc	200.i	0.96
176.gcc	integrate.i	0.97
176.gcc	scilab.i	0.96
181.mcf	inp.in	0.96
256.bzip2	input.source	0.92
256.bzip2	input.graphic	0.97
256.bzip2	input.program	0.97
183.equake	inp.in	0.98
255.vortex	lendian1.raw	0.99
300.twolf	Ref	0.93
188.ammp	ammp.in	1

Table 3: Trace Execution Quality for TC with configuration 16:3

(Trace Execution Quality = ratio between number of instructions executed from Trace Cache line and number of instruction that are actually stored at the Trace Cache line).

Table 3 shows that the trace execution quality of a Trace Cache with configuration 16:3 is very high. This result shows that we have high confidence that when fetching a certain line from the Trace Cache most of the instructions from this line will be executed. Therefore using a prediction about characteristics of all instructions that are stored at this line is relevant.

In our proposal we aim to predict long instruction sequences and therefore we're interested in trace utilization. In the case of fully utilized traces we can predict information for as many instructions as the Trace Line size. For a basic configuration it means that in case of high utilization each time we fetch a new Trace Cache line we can predict data for about 16 instructions ahead.

Benchmark	Number of instructions
art_test1	12.34
gzip_source	12.79
gzip_log	13.79
gzip_graphics	11.58
gzip_random	12.10
gzip_program	11.99
vpr_place	12.10
vpr_route	11.66
gcc_166	12.41
gcc_200	11.55
gcc_integrate	12.13
gcc_scilab	11.38
mcf	12.77
bzip2_graphics	11.74
bzip2_program	11.94
equake_inp	12.35
vortex_lendian1	12.59
twolf_ref	13.34
Ammp	11.50

Table 4: Average number of instructions executed from TC line for 16:3 configuration

Table 4 shows the average numbers of instructions executed from Trace Cache lines for different benchmarks. As trace execution quality is high (almost 1) those numbers also represent trace utilization for a 16:3 Trace Cache configuration. According to this

measurement we see that in each trace fetch we can actually predict information about twelve instructions only (on average) that will be executed in the future.

Large Traces

We want to be able to dynamic changes to the processor's configuration such as disabling and enabling units based on the prediction that we make. Disabling or enabling a unit can take a non-trivial amount of time, depending on the HW implementation (not in the scope of this work).

We define "prediction window" as a window of time during which executed all the predicted instructions and only them. In order to make a change at the processor, we need the prediction window to be big enough to allow it. Predicting more instructions ahead of the execution enlarges this window. (E.g. In case we can predict that no FP instructions will be executed in the near future we can suggest disabling of FP unit. We will benefit from this change if the prediction window is long enough to allow disabling of the FP-unit, execution under new power saving configuration and enabling FP unit back.)

Actually the time can be represented by the number of instructions that can be executed during this time; therefore "prediction window" can be represented by the number of instructions that are executed during it.

We want to be able to predict more than twelve instructions ahead (the average shown above). As we predict one Trace Cache line each time we can try to achieve bigger prediction window by increasing the size of Trace Cache line.

We checked if a bigger Trace Cache line can increase the prediction window. We chose to check a 50:10 Trace Cache configuration, meaning up to 50 instructions at the line and up to 10 branches. We need to check if all the basic observations are still relevant. First of all we want to check that we still have enough instructions coming from the Trace Cache vs. Instruction Cache and that using Trace Cache for prediction is relevant. As stated above, the following measurements are based on configuration 50:10.

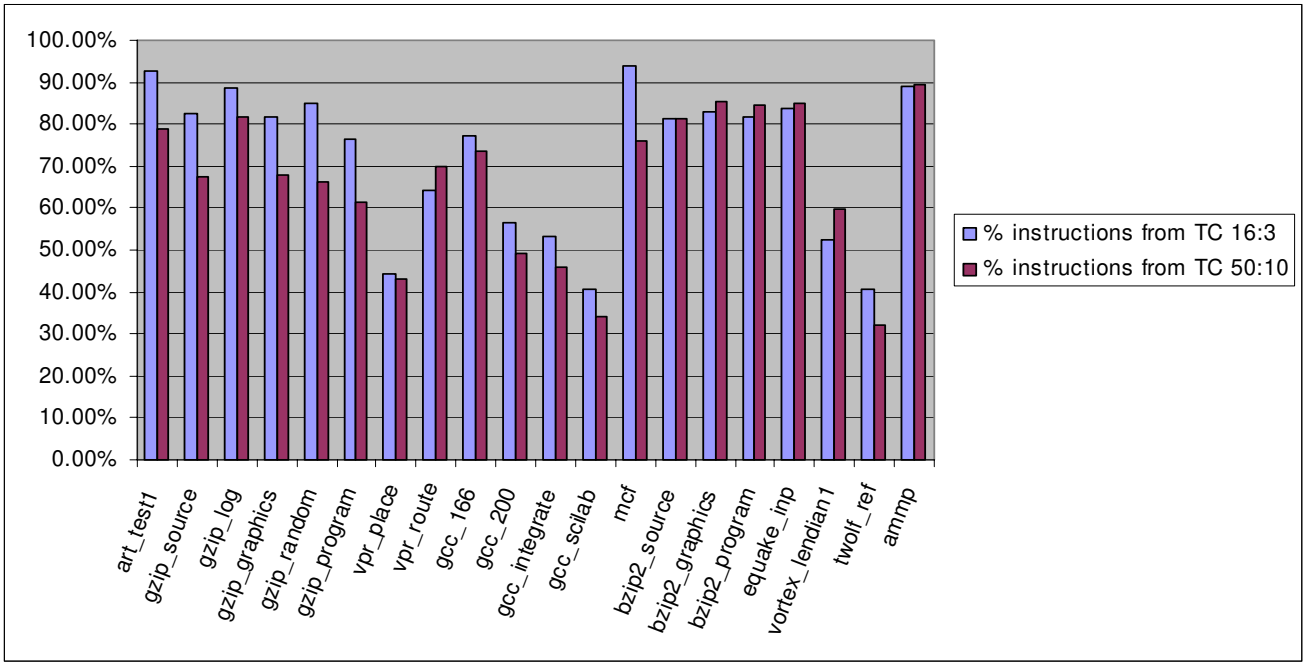


Figure 16: percentage of instructions coming from TC

This measurement at Figure 16 shows that when we enlarge the Trace Cache line most of the instructions still come from the Trace Cache (more than 50% of the instructions for most of the benchmarks). Therefore a predictions based on a 50:10 Trace Cache configuration are relevant. We still need to check if we succeeded in actually enlarging our prediction window. The percentages of instructions coming from the Trace Cache are high, but in most of the cases the numbers are lower than for a 16:3 Trace Cache configuration. Other parameters that needs to be checked, in order to decide whether predictions based on a 50:10 Trace Cache configuration are relevant and long enough, are trace execution quality and trace utilization.

Table 5 shows the trace execution quality measurements (the ratio of executed to stored instructions in the trace) for a 50:10 and 16:3 Trace Cache configurations.

Benchmark	Input	Trace Execution Quality 16:3	Trace Execution Quality 50:10
179.art	A10.img	0.96	1
164.gzip	input.source	0.95	0.88
164.gzip	input.log	0.96	0.91
164.gzip	input.graphic	0.94	0.93
164.gzip	input.random	0.95	0.96
164.gzip	input.program	0.94	0.87
175.vpr	place	0.95	0.91
175.vpr	route	0.99	0.98
176.gcc	166.i	0.97	0.96
176.gcc	200.i	0.96	0.96
176.gcc	integrate.i	0.97	0.95
176.gcc	scilab.i	0.96	0.95
181.mcf	inp.in	0.96	0.9
256.bzip2	input.source	0.92	0.92
256.bzip2	input.graphic	0.97	0.97
256.bzip2	input.program	0.97	0.9
183.equake	inp.in	0.98	1
255.vortex	lendian1.raw	0.99	0.99
300.twolf	ref	0.93	0.98
188.ammp	ammp.in	1	0.99

Table 5: Trace Execution Quality for TC with different configurations

Comparing the two Trace Cache configurations, we see that in most of the cases the trace execution quality is lower for 50:10 Trace Cache configuration than for 16:3, although it is still high enough. Knowing the execution quality is meaningless without knowing the actual number of the instructions that are stored in the Trace Cache line or executed from each Trace Cache line. We can have a very high trace execution quality, but the trace may be short. In this case we won't be able to have a long enough prediction.

Benchmark	Number of instructions 16:3 TC configuration	Number of instructions 50:10 TC configuration
art_test1	12.34	45.30
gzip_source	12.79	25.64
gzip_log	13.79	32.14
gzip_graphics	11.58	24.76
gzip_random	12.10	29.75
gzip_program	11.99	22.71
vpr_place	12.10	26.60
vpr_route	11.66	17.76
gcc_166	12.41	23.83
gcc_200	11.55	20.58
gcc_integrate	12.13	21.75
gcc_scilab	11.38	18.29
mcf	12.77	33.13
bzip2_graphics	11.74	24.63
bzip2_program	11.94	24.45
equake_inp	12.35	29.18
vortex_lendian1	12.59	25.46
twolf_ref	13.34	27.47
ammp	11.50	26.88

Table 6: Average number of instructions executed from TC line

Table 6 shows the average numbers of instructions executed from a Trace Cache line for different benchmarks. It compares between the two configurations. As trace execution quality is high (almost 1) for both configurations, those numbers also represent the trace utilization. The above measurements show a serious drop in utilization of the Trace Cache with larger lines, meaning that the efficiency of the Trace Cache has decreased too. We also see that by increasing the trace line by a factor of 3 (from 16 to 50) we achieved an average increase in predicted instructions by only a factor of 2.

By increasing the trace line size we increased the number of instructions we can predict in each trace fetch, although the instruction increase was not as beneficial as expected. On the other hand, the change caused lower trace execution quality, lower Trace Cache utilization and therefore decreased Trace Cache performance.

Another drawback of changing the Trace Cache configuration is the lack of flexibility. We prefer to be able to add our solution to any existing system with a Trace Cache, regardless of its configuration. Trace Cache configuration might be a very difficult and meticulous decision affected by different parameters that we can't and don't want to be aware of.

The conclusion is that increasing the Trace Cache line size will not help increasing the

prediction window in the desired way: having significant increase without hurting the existing configuration and performance.

Therefore we're looking for solution that will give us prediction based on the Trace Cache regardless of its configuration.

5. Long Trace sequences

In order to benefit from dynamically changing the processor's configuration we need to be able to predict the needs of the processor ahead of time. Such prediction will allow us to make the change and benefit from it in terms of reducing power consumption. We use the number of instructions that are executed during this time as representative of the elapsed time, and therefore we would like to predict as much instructions as possible.
In the previous chapter we saw that there is a potential to use the information from the Trace Cache to predict the processor's needs, as the majority of the instruction come from the Trace Cache and not from Instruction Cache. (This can also be improved as Trace Cache is being improved.)
We showed that enlarging the Trace Cache line allows us to increase the prediction window, but not enough. On the other hand, it hurts Trace Cache utilization and performance, and on the other hand, we prefer to have a solution that is configuration independent.

Another way to predict characteristics about long instruction sequences from the Trace Cache is to use more than one trace. The main idea is to use those traces as long as they are fetched one after another from the Trace Cache (but do not necessarily reside in consecutive lines in the Trace Cache). If we are able to predict such a sequence and fetch information from several Trace Cache lines together, we can handle them as a very long instruction sequence coming out of the Trace Cache. We can use the combined information from all those lines as a prediction for dynamically scaling the processor's resources in the near future, while those instructions are executed. In this case we don't need to change the existing configuration of the Trace Cache, and therefore don't hurt its utilization, performance and system flexibility. The important thing to remember is that we don't need to fetch the instructions from the Trace Cache, which may require time. All we need to do to predict the future behavior is to fetch the Tag which keeps the information about the instructions at the line. This "fetch" is much less time and power-consuming.

We define a Trace Sequence as a sequence of instructions coming only out of the Trace Cache (without fetching from the Instruction Cache). A Trace Sequence can consist of one or more Trace Cache lines.
But the questions are:
- Do we have Trace Sequences that are long enough to allow sufficient prediction?
- Do such sequences include enough instructions to make this proposal relevant?

In order to answer those questions we checked the percentage of instructions that are executed from trace sequences with different length.

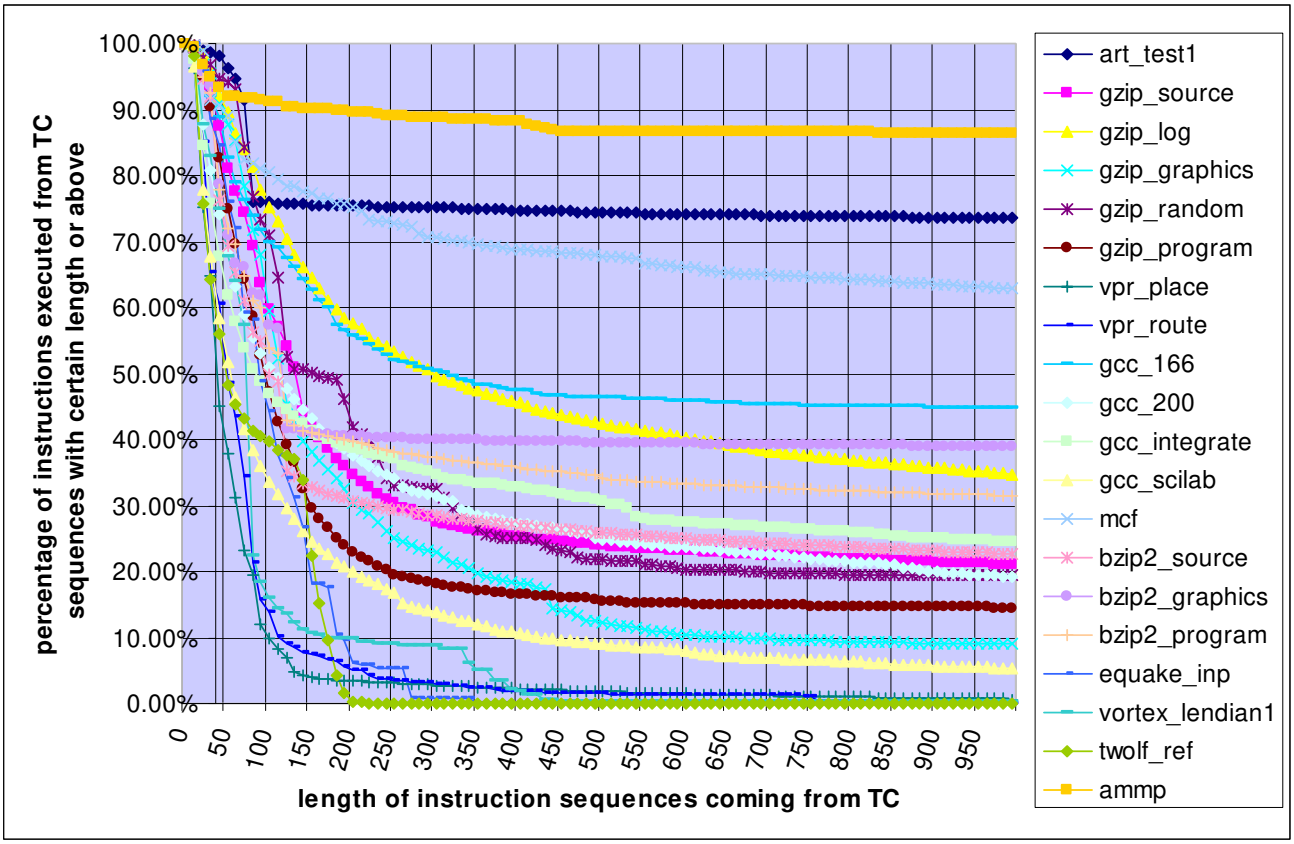


Figure 17: Distribution of instruction sequences coming from TC

Figure 17 presents distribution of instruction sequences. We can see for each benchmark what percentage of instructions executed from the Trace Cache (X axis) belongs to sequence with certain length (Y axis). In other words, each dot (x,y) in the Figure 17 represents the following data: y% of instructions executed from the Trace Cache belong to trace sequences with length x or more.

Examples to explain the graph:

- For the ammp benchmark about 90% of the instruction sequences coming from the Trace Cache are very long instruction sequences. Over 85% of the instruction sequences are longer than 1000 instructions.
- For the twolf_ref benchmark about 40% of the instruction sequences are longer than 100 instructions, but only 20% are longer than 150 instruction. There is small amount of instruction sequences with length 200 or above.

The data presented in Figure 17 is presented in the table below in more detailed manner, but with focus on certain length numbers. It presents percentage of instructions at instruction sequences with lengths over 30, 50, 100, 200, 300, 500 or 1000 executed from the Trace Cache.

Benchmark	Input	% instructions in sequences with over 30 instructions	% instructions in sequences with over 50 instructions	% instructions in sequences with over 100 instructions	% instructions in sequences with over 200 instructions	% instructions in sequences with over 300 instructions	% instructions in sequences with over 500 instructions	% instructions in sequences with over 1000 instructions
179.art	A10.img	98.79%	96.29%	75.88%	75.45%	75.12%	74.43%	73.63%
164.gzip	input.source	93.70%	81.16%	59.66%	34.69%	27.53%	23.93%	21.09%
164.gzip	input.log	96.09%	89.69%	75.07%	57.59%	49.66%	42.34%	34.56%
164.gzip	input.graphic	95.39%	87.71%	59.36%	30.35%	22.85%	12.26%	8.96%
164.gzip	input.random	96.91%	94.04%	71.02%	41.94%	32.50%	21.73%	19.36%
164.gzip	input.program	90.52%	74.81%	47.17%	22.97%	18.02%	15.41%	14.51%
175.vpr	Place	64.86%	37.95%	9.87%	3.44%	2.77%	1.82%	0.66%
175.vpr	Route	65.27%	48.54%	13.76%	5.19%	2.96%	1.49%	0.08%
176.gcc	166.i	91.90%	82.65%	69.90%	55.64%	50.49%	46.33%	44.90%
176.gcc	200.i	80.88%	68.51%	50.99%	37.51%	31.37%	24.84%	19.30%
176.gcc	integrate.i	75.82%	61.89%	46.98%	38.64%	34.55%	30.67%	24.54%
176.gcc	scilab.i	67.77%	51.76%	33.56%	19.93%	13.75%	9.10%	5.29%
181.mcf	inp.in	91.37%	89.11%	80.56%	75.11%	70.54%	67.83%	63.00%
256.bzip2	input.source	92.11%	69.20%	49.87%	30.78%	28.49%	25.90%	22.71%
256.bzip2	input.graphic	93.47%	70.88%	57.22%	40.48%	40.02%	39.55%	38.99%
256.bzip2	input.program	92.50%	71.99%	53.87%	39.79%	37.22%	34.02%	31.50%
183.equake	inp.in	88.61%	76.10%	44.14%	6.00%	0.78%	0.00%	0.00%
255.vortex	lendian1.raw	82.94%	67.76%	15.88%	9.74%	8.76%	0.23%	0.15%
300.twolf	Ref	64.39%	48.27%	39.81%	0.15%	0.12%	0.03%	0.00%
188.ammp	ammp.in	94.99%	92.11%	91.23%	89.70%	88.69%	86.59%	86.49%

Table 7: Distribution of instruction sequences

For example:

- 75.88% of instruction sequences in the art benchmark that were executed from the Trace Cache have 100 instructions or more, 73.63% of the instruction sequences have 1000 instructions or more. The set of the traces with more than 100 instructionst includes the set of sequences with over 1000 instructions.
- In the equake benchmark, 44% of instruction sequences that were executed from the Trace Cache have 100 instructions or more, but only 6% have 200 instructions or more.

We can see that most of the benchmarks have a high percentage of instruction sequences coming from the Trace Cache with over 100 instructions in the sequence.
There are several benchmarks that have high percentages of very long instruction sequences (over 1000 instructions): art, gcc_166, mcf, ammp.

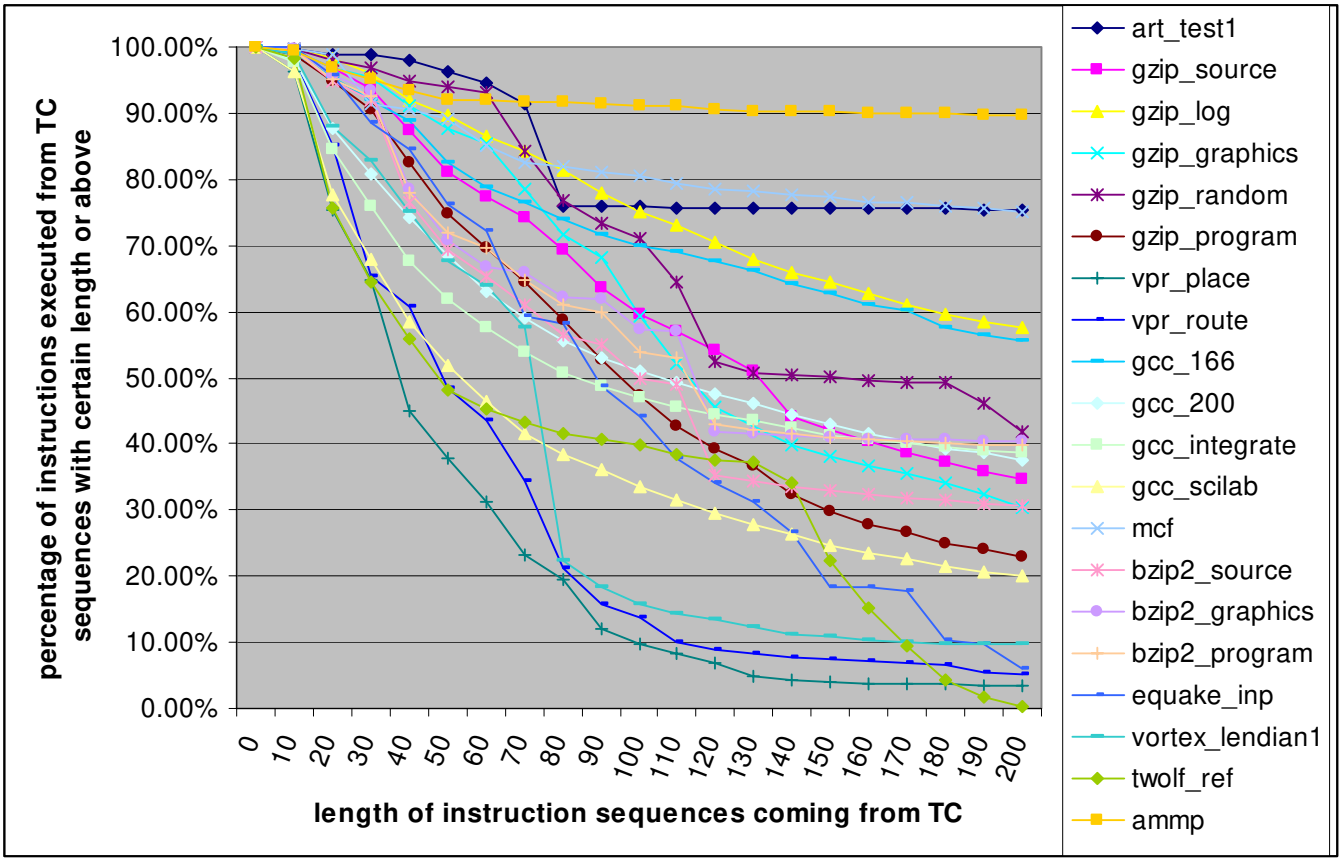


Figure 18: Distribution of instruction sequences coming from TC (zoom till 200)

The below table presents summary of the above measurements - percentage of executed instructions that came as long trace sequences from the Trace Cache. We defined here "long sequence" as sequences with at least 100 or 200 instructions. The measurement is done relatively to all executed trace sequences.

Benchmark	Input	% instructions in trace sequences with over 100 instructions	% instructions in trace sequences with over 200 instructions
179.art	a10.img	75.88%	75.45%
164.gzip	input.source	59.22%	34.47%
164.gzip	input.log	75.07%	57.59%
164.gzip	input.graphic	59.37%	30.34%
164.gzip	input.random	71.02%	41.95%
164.gzip	input.program	47.24%	23.03%
175.vpr	Place	9.87%	3.43%
175.vpr	Route	13.73%	5.19%
176.gcc	166.i	69.54%	54.84%
176.gcc	200.i	51.02%	37.52%
176.gcc	integrate.i	46.99%	38.67%
176.gcc	scilab.i	33.50%	19.90%
181.mcf	inp.in	80.56%	75.11%
256.bzip2	input.source	86.70%	80.76%
256.bzip2	input.graphic	57.22%	40.48%
256.bzip2	input.program	54.03%	39.83%
183.equake	inp.in	45.41%	6.47%
255.vortex	lendian1.raw	15.91%	9.72%
300.twolf	Ref	39.75%	0.14%
188.ammp	ammp.in	91.23%	89.70%

Table 8: Long Trace Sequences at the Trace Cache

We can see in Table 8 that many instructions that come from the Trace Cache reside in long sequences, and therefore their behavior can be potentially predicted and used as input for dynamical tuning of processor's resources.

Another idea to increase the prediction was to check the lengths of the instruction sequences coming from the Instruction Cache. Those instructions interrupt sequences of instructions from the Trace Cache. We thought that they can be ignored, if short enough,

and we can see the program or part of it as very long sequence of instructions coming from the Trace Cache. The simulations (Figure 19) showed that although most of the instructions executed from the Instruction Cache belong to short sequences, there are still a number of long instruction sequences executed from the Instruction Cache that can't be ignored.

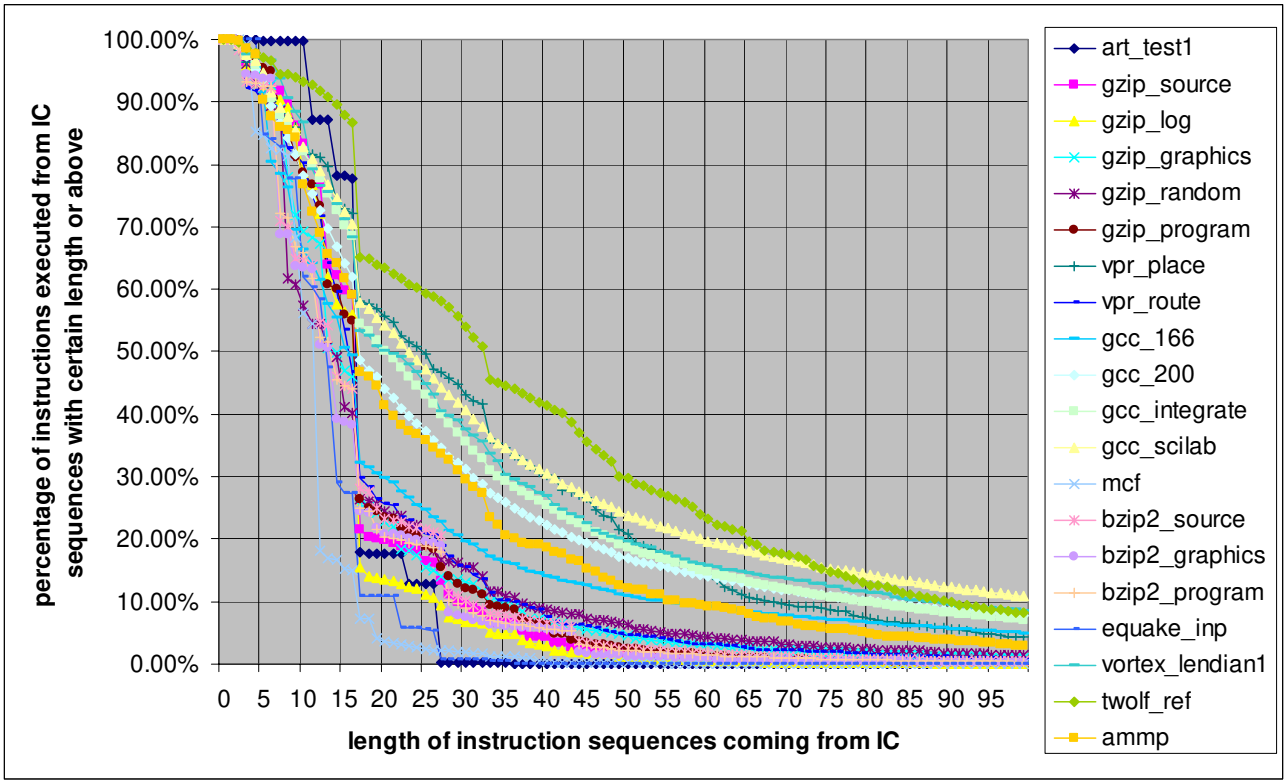


Figure 19: Distribution of instruction sequences coming from IC

Figure 19 presents the distribution of instruction sequences coming from the Instruction Cache for different benchmarks. Each point (x,y) presents the following data: y% of instruction sequences that were executed from the Instruction Cache are with length x or more.

For example: In the gcc_scilab benchmark about 10% of instruction sequences executed from the Instruction Cache are more than 100 instructions long.

Comparing Figure 19 with Figure 17 we can see that most of the instructions in long sequences are executed from the Trace Cache and not from the Instruction Cache.

Proposed architecture

In this work we propose making a small change to the existing Trace Cache to add additional information to each trace and to build a Trace Cache with trace characterization.

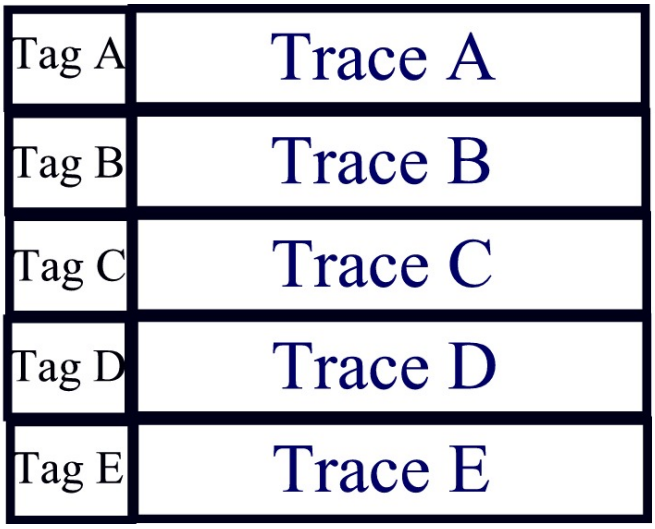


Figure 20: Trace Cache basic structure

The current architecture of the Trace Cache already includes additional information for each trace, called a Tag. The Tag includes the trace id, the trace size and other trace related information. We propose adding information about instructions inside the trace, called trace characterization, to the Tag. As we're changing the Tag only, the change we propose won't affect Trace Cache parameters, it will only increase the Tag. This information can be derived from the instructions within the trace during trace build. Having this information summarized in the Tag will allow us to have a special fetch during trace access that will fetch only the information needed for dynamical tuning without needing to fetch all the instructions. The content and the amount of additional information depend on the type of tuning that we want to make (will be discussed later).

5.1.1 Algorithm for Dynamic Tuning based on the Trace Cache

In this chapter we define the algorithm that can be used for dynamical tuning of processor's resources.

1. Retrieve the information about future requirements for the resources, based on the Trace Cache. The information will be retrieved from the Tags.
2. Analyze the retrieved information and propose an optimal configuration.
3. Compare the proposed configuration with the current configuration.
4. Change the current configuration if needed.

In this work we will focus on #1 and propose options for #2.

Another issue that should be addressed is when this algorithm should be performed. We should define a limit: the number of instructions or traces that we predict ahead. The algorithm should be performed again before all the predicted instructions are executed to recalculate the processor's needs according to the next set of instructions and to apply a new configuration if needed. There is no need to perform the algorithm for each new trace access, as long as we have data for a long time ahead. In addition to performing the algorithm according to the number of instructions that have yet to execute, we propose to do it for each misprediction, as the configuration that was predicted is not relevant anymore.

We should be aware that there are cases in which we won't be able to predict enough instructions according to predefined limit, for example if the next trace is not in the Trace Cache. Therefore we propose to perform the algorithm based on the number of instructions in the trace sequence that remain to be executed and not based on a constant time or instruction interval.

There is no need to have feedback on whether we followed the predicted path. This should work, as we use same predictor for fetching the instructions, so as long as we didn't have a misprediction the path should be the same.

Prediction of long trace sequences

We see that we can benefit a lot from predicting long trace sequences and their content. As mentioned above, such sequences mostly consist of several Trace Cache lines in the order that they should be fetched and executed (but not necessarily in the order in which they are stored inside the Trace Cache).

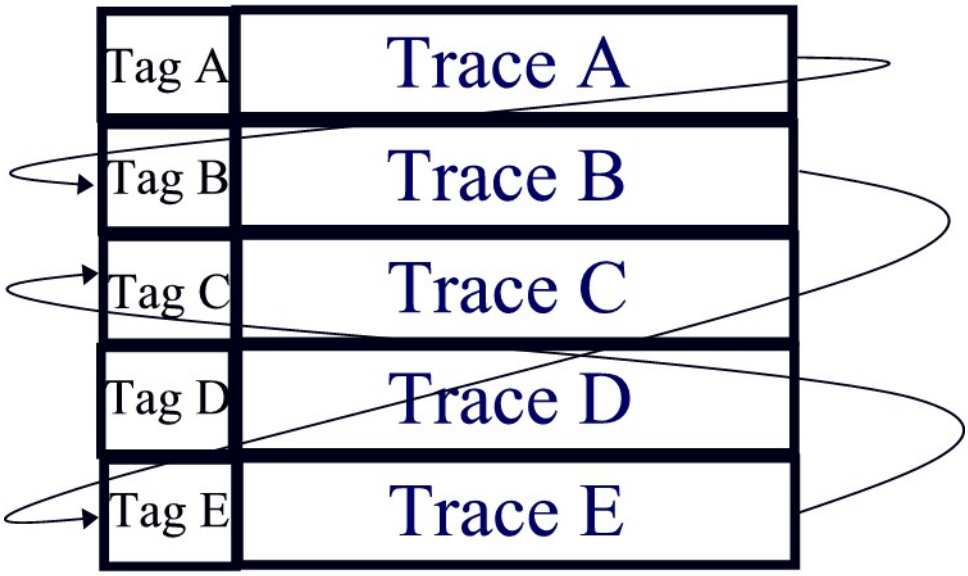


Figure 21: Example of a Trace Sequence

Figure 21 presents an example of a trace sequence ABEC which is combined of 4 traces. A, B are consecutive traces, but B and E, E and C are non-consecutive.

In order to benefit from those sequences we should be able to know the information from all the lines enough time ahead. Preferably before the instructions from the first trace of the sequence are fetched and executed, and not at the time that the actual fetches occur.
As mentioned before, what we actually need in order to perform dynamic tuning is not the prediction of the actual instructions in the trace sequences, but prediction of information about the trace sequence. There are several options to achieve this:

- A trace that starts a long trace sequence should keep relevant information for the whole sequence.
- Predict all (or enough) traces ahead and take the relevant information from each trace.

For the first option we can use the regular Trace Cache line prediction mechanism to predict sequence information by predicting one trace each time. The change that we need for this

option is that each trace should be able to keep enough information needed to make a decision for which trace sequence it may start. This solution has several drawbacks: We need to add more space to the Trace Cache to each line for additional information (may require a lot of space). We need to decide what information should be saved in case a trace can start several long sequences or be part of several trace sequences.

For the second option less information can be saved per each line, but we need to have a new prediction mechanism – the Sequence Information Predictor. It should be able to predict the next line before it's actually needed for execution. More than that, it should be able to predict several lines ahead until we have enough information or until prediction is unavailable.

We tried to simulate the second approach. We built a Sequence Information Predictor based on the Trace Predictor. The algorithm for the Sequence Information Predictor is described below in Figure 22.

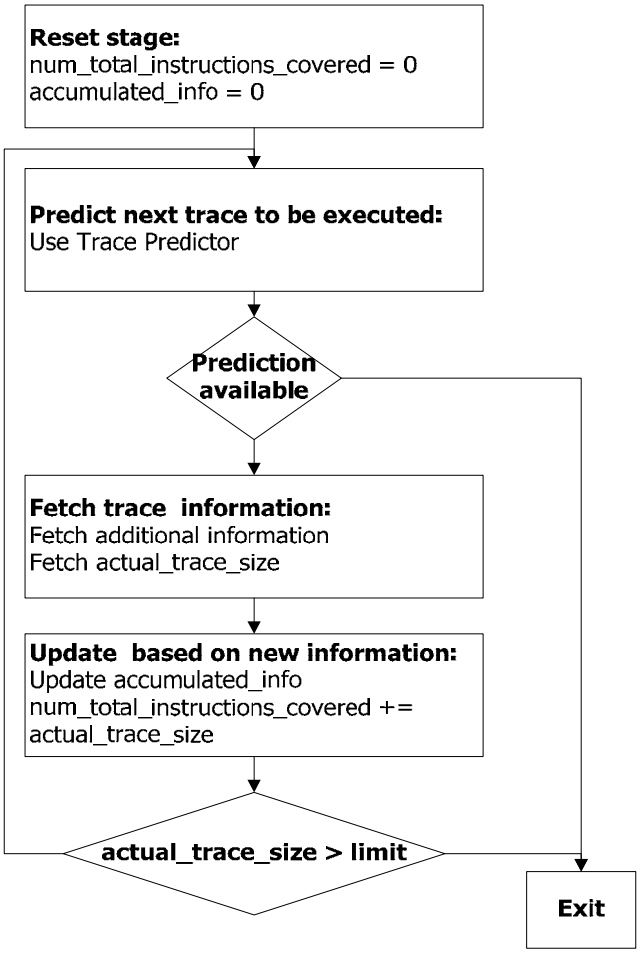


Figure 22: Algorithm for Sequence Information Predictor

Two main changes need to be done to apply this algorithm:

- Trace Tags should have additional information. This requirement is easy to be implemented and requires minimal change.
- The Trace Predictor that we use should be changed. It is required to be activated in advance, before the previously predicted trace was executed or an event fetched and before we got feedback. This requirement is more complicated and it requires an additional changed version of the Trace Predictor. The implementation is described below.

In our proposal the Trace Predictor is activated independently in two different versions:

- The Trace Predictor is activated as before to find the next trace for execution. Following this instructions are fetched for execution.
- A changed version of the Trace Predictor is activated. It tries to predict as trace lines ahead as possible. During this activation only Tags are fetched without actual fetch of instructions. It's important to emphasize that even for short traces (like 16:3 configuration) the Trace Cache is not accessed frequently, and therefore we have time for smarter algorithms that will calculate and change the configuration.

The above algorithm also requires definition of a limit. A limit is the number of instructions that is enough to make a future prediction and change the processor's configuration. Its definition is not in the scope of this work. For feasibility studies we defined it as 100 instructions. During the simulation we used "sliding window" approach. Trace Sequence Predictor was activated each time Trace Predictor was activated (each time we access new trace we predict 100 instructions ahead).

5.1.2 Change of the Trace Predictor – implementation details

In regular Trace Predictor implementation there are two tables (Simple Scalar implementation):

- Ongoing-table - keeps all the traces and an indication whether they are still valid in the processor.
- History-table - keeps predictions (trace ID) and counters for prediction correctness. This table is used only for Trace Predictor purposes.

Each trace that is fetched is added to the Ongoing-table. The trace is kept there until it is committed or recovered (due to misprediction). During this time the trace is indicated as valid. The trace becomes invalid after it's committed or removed in case of recovery. The Ongoing-table is used not only for the Trace Predictor (e.g. instructions fetch from the trace), and therefore we should keep it correct, especially the valid indications.

Each trace that is executed without recovery updates the History-table which is used for the Trace Predictor. A counter relevant to the trace (identified by trace-ID) is updated according to whether the trace had a correct or incorrect prediction. In case of a new trace a new prediction is added to the History-table.

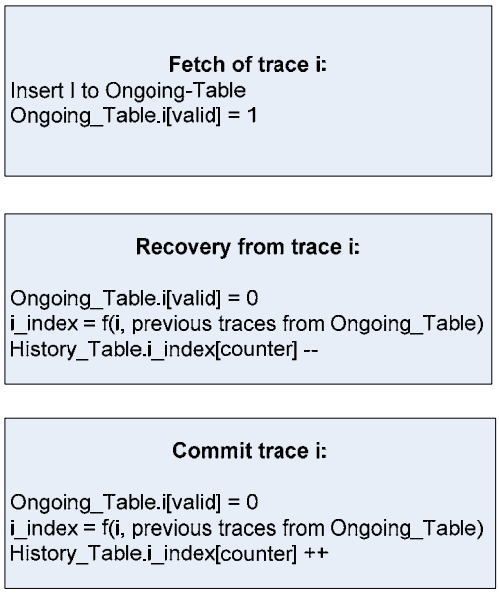


Figure 23: Table's update policy

The Trace Predictor calculates the index to the History-table based on the trace address and previous traces (next trace predictor). The calculation is done based on the traces in the Ongoing-table (even if the previous traces already committed and are not valid at the table). According to this index the Trace Predictor accesses the History-table and checks whether the counter for this trace indicates a good prediction. If the counter is high, the predictor returns a correlated trace address, otherwise there is no prediction.

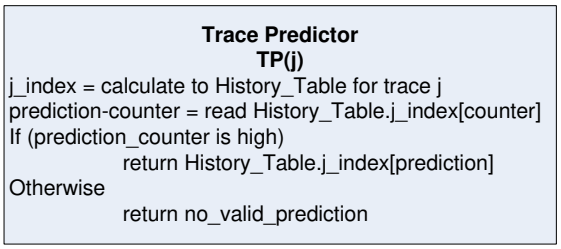


Figure 24: Trace Predictor for j (j last executed trace)

The changed Trace Predictor aims to predict more that one trace. Therefore as long as we want to predict another trace we simulate as if we fetched a trace and try to predict next one. All predicted traces are added speculatively to the Ongoing-table (without valid bit set, without changing table parameters) and we let the predictor work as usual. We don't update the Ongoing-table's parameters (e.g. valid, num-of-entities, head), so those speculative traces won't be visible for regular execution, they will be visible only for the Trace Predictor as the "path" to next trace. When the trace is actually fetched it will override its previous and speculative entry in the Ongoing-table by updating all necessary parameters.

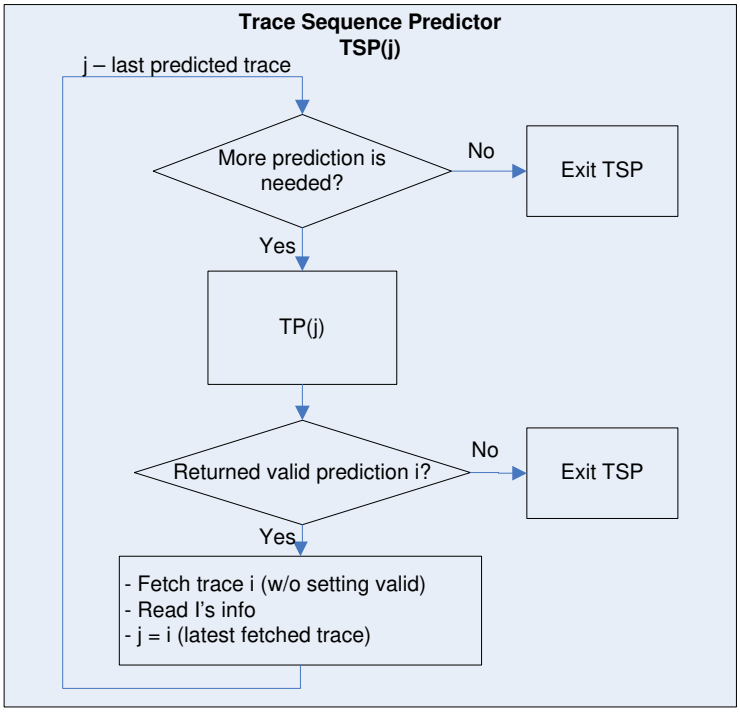


Figure 25: Trace Sequence Predictor starting from trace j

Filter for long trace sequences

We see from the results of the simulations above (Table 8) that although we have many long trace sequences, we also have many short trace sequences (the complement of long sequences). Prediction based on short trace sequences will not give Power Reduction but may harm performance. For example if we make a configuration change according to short sequence prediction we may choose to disable some unit that we may need in the near future. In this case instead of power gain we lose performance. We can gain a lot from using prediction in case of long trace sequences but we can also lose a lot from doing changes based on short trace sequences.
We would like to be able to predict long trace sequences only and base dynamic changes of the processor's resources on those predictions only. We tried to find a way to filter short trace sequences and predict only long trace sequences.

There are several origins for short trace sequences. Sometimes we just don't have prediction for next trace. Sometimes short trace sequence can be created by a misprediction in the middle of long sequence.

We simulated some basic filters that require minimal additional change.
Most of the simulated filters are based on two counters that are added to each Trace Cache line Tag. The first counter was increased each time this trace started a short trace sequence (short_seq_cntr). The second counter was increased each time the trace started a long trace sequence (long_seq_cntr).
Update of the counters: Each time we start new sequence (not necessarily for each new trace) we keep the trace ID of the trace that started it. When the sequence is stopped (we encountered fetch from Instruction Cache or misprediction) we update one of those counters. The counters are updated according to trace ID and number of instructions executed so far, depending on whether the sequence that the trace started was short or long.
The definition of a long trace sequence depends on limit requirements. In our simulations we defined a long trace sequence as a sequence with more than 100 instructions. Those counters were used later at the fetch of trace sequence information to decide whether they have a potential to start a long trace sequence.
We tried several filters, as presented below, which are based on the described above counters:

- The continuity has potential to be long if
  (long_seq_cntr - short_seq_cntr) > 0

- The continuity has potential to be long if (long_seq_cntr - short_seq_cntr) > 1
- The continuity has potential to be long if long_seq_cntr > 0
- The continuity has potential to be long if long_seq_cntr > 1

An additional filter that we tried was based on the new Trace Predictor for several trace lines (as described above). If the Trace Predictor can predict enough traces after this one in order to have a long trace sequence, then this trace is a good candidate for starting a long trace sequence. This filter covers only one source for short trace sequences – no available prediction for long sequences. This filter can be improved by taking into consideration "mistakes" in previous predictions, and whether this long sequence became eventually short. Other filters should cover all the origins of short trace sequences.

The results from simulations of the described above filters are presented at Table 9:

Benchmark	Input	Ench. Trace Predictor	long_cntr - short_cntr> 0	long_cntr - short_cntr> 1	long_cntr > 0	long_cntr > 1
179.art	a10.img	82.93%	22.13%	8.22%	66.36%	43.41%
164.gzip	input.source	57.02%	18.19%	8.06%	46.19%	26.84%
164.gzip	input.log	74.21%	36.20%	21.23%	57.92%	39.19%
164.gzip	input.graphic	34.85%	28.20%	14.69%	56.62%	33.41%
164.gzip	input.random	59.32%	36.58%	19.79%	64.10%	41.13%
164.gzip	input.program	34.37%	17.62%	9.10%	44.85%	24.92%
175.vpr	place	45.17%	12.31%	3.56%	25.75%	7.95%
175.vpr	route	80.63%	9.36%	5.78%	13.95%	6.95%
176.gcc	166.i	50.34%	31.32%	16.08%	47.75%	26.71%
176.gcc	200.i	69.63%	32.44%	17.62%	47.55%	26.45%
176.gcc	integrate.i	71.20%	23.05%	10.60%	38.36%	19.22%
181.mcf	Inp.in	79.82%	20.26%	7.95%	57.97%	36.66%
256.bzip2	input.source	77.90%	59.49%	45.85%	65.69%	49.65%
256.bzip2	input.program	79.93%	53.24%	37.08%	61.86%	43.14%
183.equake	Inp.in	77.34%	44.34%	19.77%	58.33%	25.93%
255.vortex	lendian1.raw	84.57%	15.50%	4.86%	25.78%	7.67%
300.twolf	Ref	84.35%	17.41%	5.65%	38.70%	13.49%
188.ammp	ammp.in	71.40%	18.93%	4.69%	22.23%	4.94%

Table 9: percentage of successful predictions using different filters

Each number in Table 9 presents the percentage of successful predictions of long trace sequences using a certain filter. Successful prediction – the sequence was identified by the filter as a long trace sequence and was executed as a long trace sequence. The numbers are calculated as the ratio between successfully predicted long trace sequences (that were predicted as long trace sequences and were executed as long trace sequences) and all executed trace sequences with more than 100 instructions.

We can see that even by adding a very simple filter (long_seq_cntr > 0) we can have good prediction of long trace sequences. For example: in the art benchmark 66.36% of executed long trace sequences were predicted as long trace sequences by the (long_seq_cntr > 0) filter.

Also we can see that filtering based on the new Trace Sequence Predictor increases the chance for more accurate prediction of long trace sequences. For example: in the art benchmark 82.93% of executed long trace sequences were predicted by new Trace Predictor as long trace sequences. As stated above this filter can be improved by taking into account history of mispredictions (this can be improved in future work).

Same data also presented in the figure below.

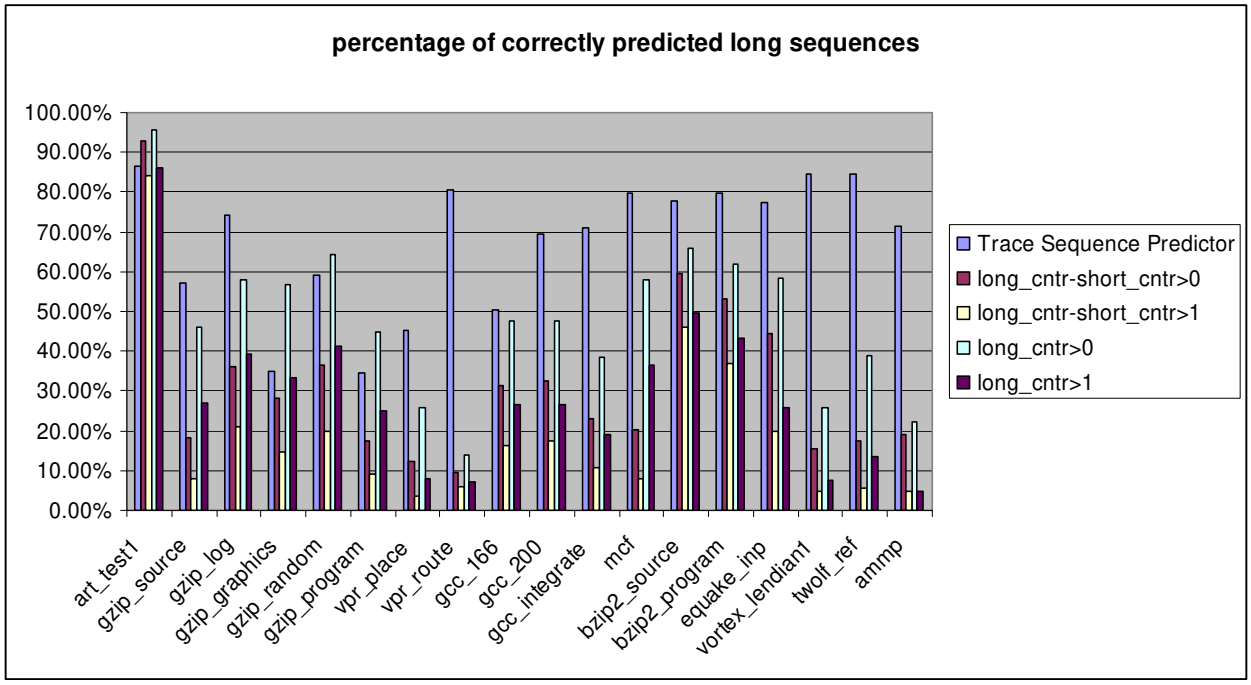


Figure 26: Successful Prediction of Trace Long Sequences using different filters

We can see that using Trace Sequence Predictor predicts successfully most of the long trace sequences.

The problem with using Trace Sequence Predictor is that it has a very high percentage of "bad" predictions, relatively to using the presented above filters. We consider a prediction as "bad" if short sequence was predicted as long. In this case we might have a wrong decision of making changes based on wrong information.

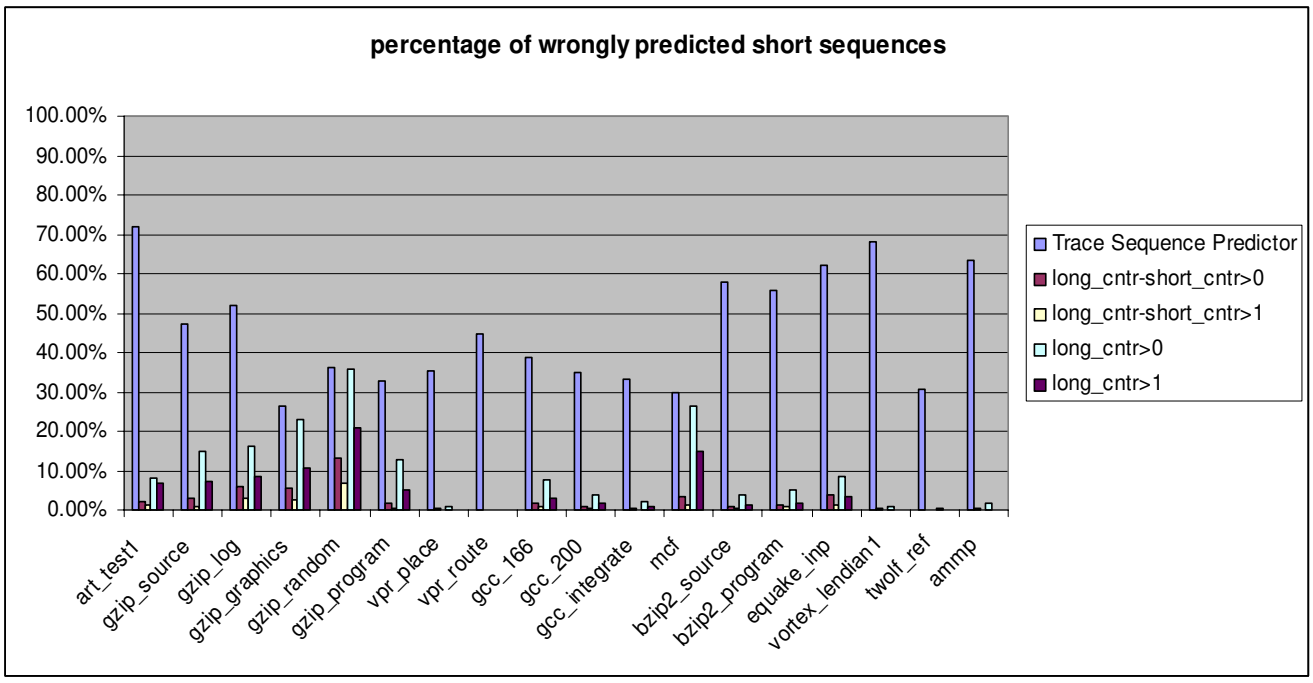


Figure 27: "Bad" Prediction of Trace Long Sequences using different filters

This figure shows that using Trace Sequence Predictor has many "bad" predictions relatively to using other filters.

6. Discussion

Characteristics of long trace sequences

We examined long trace sequences and tried to characterize them. Most of the long trace sequences (that eventually became very long – thousands of instructions) have patterns of repeating instructions (like a loop).
We tried to count those loops to check if we have correlation between the sequence length and having a loop with many iterations. We saw such a correlation for very long trace sequences. Long trace sequences (hundreds of instructions) did not necessarily include loops. We checked loops in terms of trace lines only (counted trace lines that repeated themselves).

One of the examples we presented for dynamic tuning of processor's resources was disabling the FP execution unit if we could predict a long sequence of non-FP instructions. We tried to simulate such a prediction by adding a simple indication to the Tag about whether the trace includes an FP instruction or not. According to our simulations long trace sequences either had many FP instructions or none. Therefore the FP unit can be a good candidate for dynamical tuning.

How to use Prediction of long trace sequences?

We showed that many long trace sequences of instructions can come from the Trace Cache. We proposed a way of accurately predicting information about such sequences. Assuming we can collect all relevant information about the instructions in the trace and maintain it in the Tag, how can we use it? Our proposal was to use this information for power reduction. As the necessary information can be available sufficiently ahead of execution time, we can use the information to disable and enable execution units without lowering performance.

We listed suggestions for power optimizations. Those optimizations can be done using dynamic tuning of processor resources based on the information that can be retrieved from the Trace Cache:

1. FE power down. When we predict a long trace sequence coming from the Trace Cache we can disable units that fetch instructions from the Instruction Cache. We can close parts of the FE and fetch as all coming instructions from the Trace Cache. For this change it's enough to know that we have a long trace sequence coming from the Trace Cache and what the length is. This use does not require any additional information to be kept in the Tag, as the trace size is already part of it. We don't need to know what the instructions inside this sequence are. We can enhance this idea by keeping information or

hints for decode at the Trace Cache and disabling some decode related units as well. If we keep decoded instructions we can also turn off the decoder.

2. Disable/enable FP units. Simulations show that trace sequences either had many FP instructions or none. Therefore the FP unit is a good candidate for being disabled based on long trace sequences' predictions. In order to make it possible we need to keep the information whether the Trace Cache line has FP instructions inside it.

3. "Exotic" instructions. Special hardware that is being kept for "exotic" instructions is a good candidate to be closed while not needed. Similar to FP there are special purpose instructions that use dedicated hardware. Those instructions are not widely used. As long as we don't run appropriate applications we can enjoy power saving while this hardware is disabled.

4. Execution unit balancing. Another option is to collect statistics for each trace about the utilization of different units during execution. Using this information per trace sequence we can configure execution units according to the needs of the program. During this work we didn't measure the benefit of power reduction, but other works showed that pipe balancing can achieve power reduction [1].

5. Memory operations. While identifying many memory operations based on long trace sequence, we can suggest partially disabling units unrelated to execution of those instructions as memory instructions are very time consuming. Another option is to change issue width. More accurate algorithms can be proposed for analyzing such instructions and base dynamic tuning on whether they cause hit or miss.

6. Power Reduction during loop execution. Simulations showed that many very long trace sequences are based on loops. We can recognize large numbers of loop repetitions using Loop Detector. On the other hand we can collect statistics on different unit utilization during the loop execution based on Trace Cache. Dynamic tuning can be based on the above information. Trace Cache will provide information about the resources needed for loop body. Loop Detector will provide the number of repetitions for better prediction of sequence length. Using data from trace sequences we can do this for long loops as well (where loop body consists of several traces).

7. Summary

This work showed a new approach to using Trace Caches. Most of the works proposed using Trace Caches to achieve low latency and high bandwidth during instruction fetch.
In this work we tried to go in another direction and propose using the information that we already have in the Trace Cache for reduction of Power Consumption.
This work is based on the assumption that the information in the Trace Cache is highly accurate (most of the traces are "hot" and if we have a hit at the Trace Cache then we get a correct trace).

Our proposal is to use the Trace Cache as a monitoring unit for dynamic tuning of processor's resources. Trace Caches store instructions in their executed order. If we can predict the instructions that will be executed in the trace then we can adapt the processor's resources to the actual needs of the program.
In this work we do not add a Trace Cache, but rather make small changes. The baseline of this proposal is processors that have Trace Caches.

First of all we checked the feasibility of the proposal, whether Trace Cache can be used as a monitoring unit. We saw that the majority of executed instructions come from the Trace Cache and not from the Instruction Cache (most of the benchmarks showed 70% or above). Therefore making decisions about future execution based on instructions executed only from the Trace Cache has high potential. We checked the quality of trace execution to understand whether the data about instructions in the trace is relevant. For all the benchmarks the trace execution quality was very high (over 0.92). Therefore we can predict future execution based on the instructions stored in the Trace Cache, as there is high confidence that they will be actually executed.
We proposed tuning the processor's resources dynamically based on prediction of what resources will be used. We know that we have this information in the Trace Cache. We showed that this information can be retrieved from the Trace Cache with high confidence. If we can predict which of the processor's resources will be used, we can turn off redundant hardware and save power. Changes based on turning units on or off units take time. In order to make such changes and benefit from them, we need to have information about future execution sufficiently ahead of time. Prediction of one Trace Cache line ahead, as achieved by regular trace line fetch, is insufficient for this purpose. For small a Trace Cache with regular traces we can predict only 12 instructions in average. Increasing the trace line proved to be inefficient, therefore we defined long trace sequences as several traces executed one after the other. We considered sequences with more than 100 instructions as

long. According to our simulations we have enough long trace sequences coming from the Trace Cache. Over 50% of the instructions executed from the Trace Cache come as part of long trace sequences. This shows that if we can predict those long trace sequences we can predict information about most of the instructions sufficiently ahead of time.
We changed the Trace Predictor to be able to predict information about several trace lines ahead (trace sequence). By doing this we found a way to predict information about many instructions that will be executed in the future. Knowing this information allows us to tune the processor's resources accordingly: turn-off redundant hardware and turn-on required hardware.
We can benefit, in terms of power, from knowing about future instructions, but we can also lose performance if we base resource changes on short trace sequences. We tried to "build" some simple filters that would help us to distinguish between long and short instruction sequences and we succeeded predicting up to ~60% of long trace sequences.

We tried to analyze those trace sequences to understand how the information that we can retrieve can be used for reducing power consumption. We proposed several directions for dynamic tuning based on the information that can be retrieved from long trace sequences coming from Trace Cache.

Our proposal is independent of the Trace Cache that is used or its configuration. We tried to base our simulations on the Trace Cache with several improvements in order to achieve better performance. Improvements of the Trace Cache in terms of trace prediction, utilization and ability to cover most of the executed program, will contribute to better results of our proposal as well.

Future work

In this work we proposed a microarchitectural enhancement that aims to reduce Power Consumption based on the Trace Cache. We checked the feasibility of this proposal and presented some ideas that can be implemented based on this enhancement. We didn't simulate most of those ideas and they can be the subject of future work.
More work can be done to improve the long trace sequences filter to achieve a higher percentage of prediction for long sequences and a lower percentage of prediction of short sequences (i.e., to reduce number of times short trace sequences are predicted as long). Wrong prediction of short sequences instead of long has a high penalty in performance.
In this work we didn't simulate the real gain that can be achieved in power consumption and loss of performance and this can be a subject for future work.
One of the ideas that came from analyzing trace sequences is to combine a Loop-Detector

with a Trace Predictor to identify long trace sequences that are based on a loop that is repeated many times. This can be used as a prediction for very long trace sequences. More work can be done in analyzing long trace sequence and finding other proposals for dynamic changes.

We presented the existence of long trace sequences and how to predict them. More work can be done to define what a long sequence is in terms of number of instructions, and how to determine this number. Calculation of this number can be done based on the resource that will be tuned and how long it takes to make the change.

We tried to predict many instructions to have a big prediction window. However we didn't have a definition for mapping the time units to the number of instructions. This mapping may depend on the configuration (execution width) of the processor and might be tuned as we change the configuration.

We proposed defining the configuration based on additional information from instructions that will be executed. We didn't define how to set the configuration or perform the actual changes. All those are subjects for future work.

8. References

[1]. R. Iris Bahar, S. Manne; "Power and energy reduction via pipeline balancing", In Proc. ISCA'01, pages 218–229. June 2001

[2]. Behar, M.; Mendelson, A.; Kolodny, A.; "Trace Cache Sampling Filter", ACM Transactions on Computer Systems (TOCS), 2007

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

[4]. D. H. Friendly, S. J. Patel and Y. N.Patt, "Alternative Fetch and Issue Policies for the Trace Cache Fetch Mechanism"

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

[6]. G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel: "The microarchitecture of the pentium 4 processor", Intel Technology Journal Q1, 2001

[7]. A. Iyer and D. Marculescu, "Run-Time Scaling of Microarchitecture Resources in a Processor for Energy Savings," Proc. Kool Chips Workshop, Dec. 2000.

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

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

[10]. Peleg, U. Weiser; "Dynamic Flow Instruction Cache Memory Organized Around Trace Segments Independent of Virtual Address Line", United States Patent No. 5,381,533, Jan 10, 1

[11]. D. Ponomarev, G. Kucuk, and K. Ghose,, "Dynamic Resizing of Superscalar Datapath Components for Energy Efficiency", in IEEE Trans. On Computers, 55(2), pp 199-213, Feb. 2006

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

[13]. 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

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

[15]. R. Rosner, Y. Almog, M. Moffie, N. Schwartz and A. Mendelson, “PARROT: Power Awareness through Selective Dynamically Optimized Traces”, in PACS’03, Dec. 2003.

[16]. Rosner, R.; Mendelson, A.; Ronen, R.; “Filtering techniques to improve trace-cache efficiency” in International Conference on Parallel Architectures and Compilation Techniques (PACT), Page(s): 37 -48, 2001