Effective use of trace caches

Oren Katzengold
Effective use of trace caches

Research Thesis

Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science in Computer Science

Oren Katzengold

Submitted to the Senate of the Technion - Israel Institute of Technology

Adar, 5766 HAIFA MARCH 2006
The Research Thesis Was Done Under the Supervision of Dr. Avi Mendelson and assoc. Prof. Assaf Schuster in the Faculty of Computer Science

THE GENEROUS FINANCIAL HELP OF THE TECHNION IS GRATEFULLY ACKNOWLEDGED

I sincerely thank my advisors, Dr. Avi Mendelson and assoc. Prof. Assaf Schuster for all the help and support during my long research.

I wish to thank Dr. Avi Mendelson for bringing up the thesis topic, for coming up with creative solutions for all the problems in the way and for always being supportive and patient.

I wish to thank assoc. Prof. Assaf Schuster for his technical help, fresh ideas and for showing different point of views.

I thank the DSP lab and Eran Isler for all the help with computing power for this research.

I thank the Technion staff, the secretaries Hava Shamir, Dorit Asa, Mika Shapiro and Yardena Kolet for being so helpful and patient all along the way.
Content

1. Abstract .....................................................................................................................1
2. Introduction ...............................................................................................................2
3. Out of Order Execution Architecture ........................................................................6
4. Trace Cache ...............................................................................................................9
   4.1. Introduction .........................................................................................................9
   4.2. Trace caches design parameters .......................................................................11
   4.3. Trace predictor ..................................................................................................12
   4.4. Basic observations .........................................................................................13
5. Simulation Environment ...........................................................................................15
   5.1. Simple scalar .................................................................................................15
   5.2. The trace cache ..............................................................................................16
   5.3. The Filters ......................................................................................................17
   5.4. The Hints .......................................................................................................20
   5.5. Condor – testing environment .......................................................................21
6. Profiling ...................................................................................................................23
   6.1. Introduction .....................................................................................................23
   6.2. The profiler structure .....................................................................................24
   6.3. Collecting the information .............................................................................27
   6.4. SimpleScalar profile info ..............................................................................29
   6.5. Creating build hints ......................................................................................30
7. Dynamic Filters .......................................................................................................31
   7.1. Introduction .....................................................................................................31
   7.2. The filters which were tested ..........................................................................32
   7.3. Results ...........................................................................................................38
   7.4. Comparing to trace predictor .......................................................................53
8. Hints ........................................................................................................................54
   8.1. Introduction .....................................................................................................55
   8.2. Implementation of hints ................................................................................58
   8.3. Type of Hints ..................................................................................................60
   8.4. Choosing hints ...............................................................................................62
   8.5. Results ...........................................................................................................65
9. Conclusions and Future Research ...........................................................................77
Appendix A: Benchmarks ...........................................................................................79
References ...................................................................................................................84
List of Figures

Figure 3-1 - A model of P6 (Intel) OOO machine ________________________________7
Figure 4-1 – Compared results of machines: with no TC, with TC but no Tpred, with both TC and Tpred. The results are distributed by trace cache size (in sets) ____________13
Figure 5-1 SimpleScalar Simulator with T-cache _______________________________17
Figure 5-2 - A focus of T-cache with filters __________________________________19
Figure 5-3 - A focus of the hinting system for traces ___________________________21
Figure 6-1 - the data structure of the profile image _____________________________27
Figure 7-1 – CPI results of the different programs as a parameter of trace cache size (the values are relative and can’t be compared between programs) ____________________40
Figure 7-2 – comparing an aggressive usage threshold to a soft one as it inflicted in different cache sizes (cache sizes are measured in number of sets) ________________42
Figure 7-3 – The distribution of CPI results in the soft usage case (the values are relative and can’t be compared between programs) ____________________________43
Figure 7-4 - The distribution of CPI results in the aggressive usage case (the values are relative and can't be compared between programs) __________________________44
Figure 7-5 – the different filters performance distributed by cache sizes (in sets) ____45
Figure 7-6 – comparing the different filtering techniques as they stand by themselves, distributed by cache sizes (in sets) ___________________________________________________________________________47
Figure 7-7 – comparing the inner trace cache techniques when used with a usage filter, distributed by cache sizes (in sets) ___________________________________________________________________________48
Figure 7-8 – comparing the number of instructions fetched from the Instruction cache in all filters combinations distributed by cache size (in sets) _____________________49
Figure 7-9 - comparing the performance of all filters combinations distributed by cache size (in sets) ___________________________________________________________________________50
Figure 7-10- actual part of the instruction code fetched out of the instruction cache, distributed by cache size (in sets) ___________________________________________________________________________51
Figure 7-11 – Compared results of machines: with no TC, with TC and dynamic filters, with TC and Tpred. The results are distributed by trace cache size (in sets) ________53
Figure 8-1 - scheme of the hints making process _______________________________59
Figure 8-2 – performance results of SPEC programs with different trace cache sizes to help choose candidates for hints experiments (the results are relative and can’t be compared between programs) ____________________________66
Figure 8-3 – improvement achieved by adding hints to the code as it appears in the different tests, distributed by programs ____________________________67
Figure 8-4 – performance of the cut & merge tests with and without hints, distributed by programs (the results are relative and can’t be compared between programs) ________69
Figure 8-5 – performance of the kill tests with and without hints, distributed by programs (the results are relative and can’t be compared between programs) ____________69
Figure 8-6 – performance of the LRU tests with and without hints, distributed by programs (the results are relative and can’t be compared between programs) 70

Figure 8-7 – comparing performance of all the techniques as they stand on their own distributed by programs 71

Figure 8-8 – comparing the combinations of all suggested techniques, distributed by programs 72

Figure 8-9 – results of tests running the vortex program with and without hints, distributed by inputs 73

Figure 8-10 – results of tests running the perl program with and without hints, distributed by inputs 74
List of Tables

Table 5-1 SimpleScalar Parameters. _________________________________________15
Table 7-1 – the performance of the inner trace cache technique presented relatively to the aggressive usage filter results used on the same cache size ______________________46
Table 7-2 – c & m technique improvement on top of a usage filter ______________48
Table 7-3 - number of full inlined functions fetched from the TS (geomean) ______52
Table 8-1 - profiling results of perl, distributed by inputs (part A) ____________75
Table 8-2 - profiling results of perl, distributed by inputs (part B) ____________76
1. Abstract

Trace cache was found to be very effective in improving the performance of wide machines and reduce the active power of mainly CISC architecture. Unfortunately, in order to achieve high efficiency of the trace cache, large number of traces needs to be maintained in the trace-cache, resulting in significant increase in area and power leakage. This work aims to improve the power efficiency of small trace caches by keeping mainly "good traces".

Our solutions focus on sorting the traces either dynamically or statically, helping the trace cache to better decide which traces to keep.

We tested two approaches; one calls for filtering technique to remove bad traces from the trace cache while the other uses the compiler which by using a profiling image chooses the good traces in advance and hints the trace cache of them.

In the filtering techniques we tried blocking inconsistent traces and use dynamic feedback in order to estimate the already saved ones. In the hints concept, a special pragma instruction was added in order to allow the compiler pass hints to the trace cache.

The paper shows that the use of our filtering techniques can improve significantly the overall performance of small trace caches. The paper also shows that if the proposed filters are used in conjunction with a machine without a trace predictor, it can outperform similar systems with Trace cache and trace predictor.

The system can even be improved if compiler generated hints are added to improve the quality of the proposed filter by additional 5%-6%.
2. Introduction

In modern computer processor architecture, when machines have the ability to execute instructions out of order, there is a high demand for parallelism of the machine stages, or as it is often referred to as instruction-level parallelism (ILP).

The trace cache [1][2][3] is one aspect of serving the ILP concept; the main idea is to make the instruction cache save the instruction code not as continuous memory blocks but paths of instructions in the order they are fetched. The instructions are fetched from the regular memory or cache, while doing so a trace of instruction is being built; the trace cache, by saving those made traces, is able to prepare continuous code to be fetched the next time it is needed. More on the trace cache will be elaborated under chapter 4 in this thesis.

When moving toward processors using ILP, trace caches were proved to be very important for both performance and power when used in wide modern processors; this is mostly thanks to the fact that they increase the number of effective instructions being fetched in each cycle (in average).

However, it is suggested in many works that not all the traces have the same contribution to the overall performance of the trace cache [4][5][6][7], one can use the following characteristics for traces of a program:

**Hot traces**: traces which are often being used; these traces are fetched a lot and during short periods of time, therefore they are likely to be reused often. There is a good chance that most of the times they will be needed they will already be available in the trace cache.

**Cold traces**: these traces are seldom used; most probably they will be fetched only a few times and then the next time they will be used again, if at all, will be a long period of time afterward. The total number of times they are used is not relevant but only the frequency they are fetched.

**Good traces**: so far only the frequency of the traces used was discussed, the machine could either build them or fetch them. However the machine can be wrong and the trace might cause a branch miss prediction. Good traces are such that tend to be correct, that is the trace fully commits in the vast majority of times it is used.

**Bad traces**: if a trace tends to cause branch miss prediction it is considered a bad trace, it is most probably a fault made by the prediction mechanism, if exists, or the machine situation (traces available in the trace cache) if no such mechanism is used.
These definitions create two scales, a hot – cold one and a good – bad one, how many traces are considered in each category is a subjective definition emerged from the situation, in order to keep the discussion general the groups will remain countless for now. It should be mentioned that "bad" it a relative concept to a machine and its prediction technique. A trace becomes bad once the machine executes it when it shouldn't, if the machine hadn't done so the trace wouldn't cause miss prediction.

Good traces, on the other hand, suffer from a different problem; when a trace is good it will be good for all machines as long as the machine will choose to fetch it. However it is uncertain whether a better trace exists or not, i.e. there could be a better combination of traces which will produce better performance. It requires different heuristics and strong profile analysis to choose the best traces and even then there are no guarantees.

Four combinations can be formed from the last trace characteristics, good and hot traces, good but cold, bad and cold or bad and hot.

**Good and hot traces**, as expected, should be the most valuable traces, it should be beneficial for the machine to have them available in the trace cache; some researches take it one step forward and make these traces subject for dynamic optimizations [7][8].

**Good and cold traces** are traces that although causing no penalties are rarely used, in most cases they are bad for the trace cache because they take valuable space, however when the cache is big enough they should be kept.

**Bad and cold traces** are harmful, these traces cause the machines to pay in both performance and power by creating miss predictions as well as occupy valuable cache space; however being cold they don't cause enough damage to justify being dealt with, except maybe not saving them in the trace cache on the account of being cold.

**Bad and hot traces** are the real problem; these traces manage to trick the fetching mechanism into being fetched a lot although tending to cause branch miss prediction. These traces hit the machine in a weak spot and in order to avoid the damage, some mechanism need to be added for the machine in order to discover such situations.

Keeping the hot and good traces rather then the bad ones is important to guarantee the maximum benefit from use of trace cache. This issue can be approached in either hardware or software. Our research focuses on hardware based solutions. Software techniques, although may be important, are out of the scope of this work.
This thesis offers two solutions to that problem; the first one suggests to dynamically manage the trace cache in order to keep as many good traces in the trace cache, and either remove or not save the bad ones. The second solution deals with hints which are planted in the instruction code by the compiler in order to help the trace cache manage its traces better.

The first solution uses two mechanisms; one is a cache like table, located outside the trace cache (not an integral part of the trace cache) that works as a filter preventing non persistent traces from getting into the trace cache. This table counts the number of times the traces are being built, and only when a single trace was used enough times (given an initial threshold) it is transferred into the trace cache to be saved there.

The second mechanism is an internal one (a part of the trace cache), this mechanism tries to remove traces which tend to be wrong (bad traces) out from the trace cache. There are two types of such cleaners; one removes those bad traces from the cache completely by applying a counter to each trace in the trace cache. These counters measure the times in which the trace was completely committed as opposed to the times it caused a miss prediction. The other one suggest not to "punish" the entire trace, instead, the traces are cut at the branch that went wrong (caused the miss predictions) leaving in the cache only the part of the trace that committed properly, thus having shorter but "safer" traces; then allowing the traces to be merged later with the following trace being fetched in order to form a better and longer trace (short traces are a waste of cache space as well as bad for the ILP concept).

Our second solution aims to transfer knowledge the system gathers either at compilation time or at runtime to the cache managing algorithm. We propose to do this by using profiling techniques in order to statically form "hints" for the trace cache. The use of hints can be widely exploited for many different algorithms. In this work we examine an architecture where a special trace cache is devoted for those traces which are marked as "good traces", the rest of the traces are maintained by a regular trace cache which deals with non hinted traces. We will show that the proposed new technique can improve the overall performance and reduce the number of trace builds.

Previous works are discusses in each part according to relevancy. The rest of the paper will be organized as follow: Section 3 will go over basic out of order terminology, section 4 will elaborate on the trace cache concept. Section 5 shows the parameters of the simulator used for this work, as well as special units which were added. It will also present
the working environment on which the tests were performed. Section 6 will discuss the use of profiling and will show a detailed description of the profiler we've built. Section 7 presents the dynamic filters and shows the performance results they provide and section 0 will deal with the hints and their performance results.
3. Out of Order Execution Architecture

When trying to support ILP, the machine must execute a few instructions simultaneously; the problem is that not all instructions take the same amount of clock cycles as well as dependencies between instructions which makes some instructions wait for others to be executed.

The above reasons pushed the idea of having instructions executed not in the order they are presented in the code; in such machines instructions are executed whenever they have all their data available and there is an available unit to support their operation.

The instruction code still comes in a sequential way, therefore most out of order (OOO) machines are divided to a front-end and a back-end; the front-end works in order, fetching the instructions from the memory and preparing them to be executed, while the back-end is the OOO part in which the machine executes any instruction it is able to.

Since in many cases it is important for the machine to know what part of the code it has finished executing (interrupt, debugging, branch instructions), and since it is easier to support, most machines work with an in-order completion i.e. the instructions are committed (finished) in the order they were fetched.

We will use the P6 OOO machine by Intel [33] as a model to discuss the different parts of an OOO machine.

**Front-end**

The front-end is the in-order part of the machine, in this part the machine fetches instruction from the cache and moves them to the decoding unit, if a branch instruction appears the machine continue fetching according to a branch prediction mechanism.

When possible, more then one instruction can be fetched from the cache, depending on the machine's abilities.

The decode stage prepares the instructions to be executed, in the P6 example using a CICS architecture the instructions are translated into RISC like instructions called Uops. The registers in the instructions are renamed to physical registers using a table called RAT (register allocating table), the decoded instructions are inserted into a special instruction buffer called ROB (reordering buffer) as well as to an executing queue waiting to be executed. There are two such queues, one for arithmetic operations called RS (reservation stations) queue and one for memory operations (load, store) called MOB (memory reordering buffer).
The ROB is a cyclic buffer which holds the instructions fetching order. Instructions are inserted into the ROB after being decoded and they are removed from it after being committed. The committed instructions indicate to the user the part of the program which is done, the instructions in the ROB in any given time indicates instructions being processed by the machine (although some of them may just be waiting there).

The ROB also provides the machine with a pool of physical registers (each entry is a physical register), instruction results are saved in the ROB entries until the instruction commits.

**RS**

The RS is a pool of uops which haven't been executed yet, for each uops it saves its attributes as well as the value of the input data. Each operand has an indication whether it is ready (its value is available), when a value is not available it means that the operand waits for another uops to be executed, when the needed data will pass via the WB bus to
the ROB the RS unit will bypass it and write the value in the operand space turning it to "ready".

When all values are available ("ready") the uops can be executed and the machine will do so once there is an available executing unit, a special unit called dispatcher chooses which of the ready uops will be executed.

Executed uops are written to the ROB and deleted from the RS; if the RS is full it will stall the decoder.

**MOB**

Manipulate the load and store operations, if possible, it allows OOO among memory operations. Each memory uop is allocated a new entry in-order, address is updated when known.

Memory dependencies can't be fully resolved statically, therefore in most modern processors load uops may pass loads/stores but store uops are executed in order. Store instructions do not write their data into the memory but only when they retire (commit), therefore a store instruction is considered done (finished the execution stage) once it has an address and a value.

**Back-end**

The machine executes in parallel arithmetic operations as well as memory operations based on the information of the RS and MOB units. When an instruction is finished it writes its result (if exist) into its entry in the ROB.

Instructions retire from the ROB in order, any entry may commit after the one before it did. An instruction commits by writing its data to the "real" (architectural) registers or to the memory.
4. Trace Cache

4.1. Introduction

As the issue width of superscalar processors increases, the importance of instruction fetch bandwidth increases as well.

In order to support wider machines the machines' ROBs (Reordering Buffers) are made larger so they could hold more instructions for the executing units to select from, the number of executing units increase as well in order for the machine to be able to execute more instructions per cycle; that all leaves the fetch bandwidth as a performance bottleneck.

While instruction cache hit rate and branch prediction accuracy has long been an issue for fetch bandwidth, some more factors should be considered:

- Branch throughput – if only one branch can be predicted in a single cycle then only one basic block can be fetched during a cycle; given the rule of thumb that on average every fifth instruction is a branch that might be a problem.
- Noncontiguous instruction alignment – when having a taken branch or a jump instruction the continuity of the code is broken and it is proven to be hard for most fetch mechanisms to continue fetching instruction in the same cycle.
- Fetch unit latency – the need for higher branch throughput and noncontiguous instructions alignment might increase the fetch unit latency (by creating more miss-predictions or making the recovery phase take longer since there are more instructions in the pipe); ways must be found in order to minimize the latency impact.

The trace cache [1][2][3][2] tries to solve these problems by capturing dynamic instruction sequences; each cache line in the trace cache is a snapshot or a trace of the dynamic instruction stream.

Each trace is limited to n instructions (the cache line size) and m basic blocks (which means m-1 branch instructions). A trace is specified by a base address (address of the first instruction in the trace) and directions of its branches (since the trace do not keep the addresses of the branches, an indirect branch always terminates the trace).

If the same trace will be required again it will be fetched from the trace cache and when there is no trace to meet the demands the instructions will be read from the regular Instruction cache.
The trace cache approach relies on dynamic sequences of the code being reused, that is thanks to:

- Temporal locality – the fact that programs tend to spend most of their running time using a small part of the static code.
- Branch behavior – branch instructions tend to behave the same during a program run (that is the reason for the branch prediction high accuracy).

The trace cache works in two phases: build and fetch. When a trace is to be added to the trace cache it first needs to be built from the regular instruction cache, the dynamic instructions are inserted into a buffer in the order they are accessed, while doing so a directions vector is built as well in order to recall the branch directions which were used in the trace. Addresses are not available when fetching instructions from a trace, therefore for each place where the trace might end an address needs to be added. This address can be used by the fetch mechanism to know where to continue fetching from; if the trace ends with a branch instruction it could be wise to save two addresses (taken and not taken), another option is to save only the address of the last read instruction and let the BTB find the taken address if needed.

Eventually the trace is copied from the buffer to the trace cache and saved there, the trace can later be accessed by its base address alone or a combination of the base address and its directions vector depending of whether a trace predictor is used or not. The trace cache may choose to allow two traces with the same base address to be saved at the same time, if so a trace predictor must be used.

When a trace is fetched from the trace cache the PC stops influencing on the instruction fetch stage; the PC is still calculated for recovery reasons while the branch instructions in the trace resolve their conditional expression without causing a jump, after getting a result it is compared to the directions vector of the trace to see if the trace took the branch correctly. When a miss prediction is detected, the machine needs to roll back to the mistaken branch and look for a new trace (or build it) from that point.

CISC machines like Intel's Pentium 4 [9], might choose to save decoded instructions in the trace cache; since addresses are not effective within a trace there is no problem in changing the size of the instruction code. 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.

In P4 by Intel for example, there is a trace cache that can hold up to 12K uops (decoded instructions), the trace cache is the L1 instruction cache of the machine and it receives a similar hit rate to an 8K to 16K bytes conventional instruction cache.

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 few CICS instructions is hard 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.

4.2. Trace caches design parameters

Few things that need to be noted when using a trace cache which will affect its functionality:

- **How many traces in the trace cache can have the 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.

- **When will the trace be saved** – the buffer can transfer traces when finished building them however they might not be valuable traces; copying the traces after they commit could give the machine better confidence in the trace. Doing so will also call for a bigger buffer as well as a more complex control unit. A different option to increase the confidence of the saved traces could be done by adding some filters in order to have only selected traces in the cache.

- **When should the building stop** – aside from the obvious cases of instructions and branches limitation for each trace, there could be some other situations the machine will do better by not continuing the trace. When an indirect branch or jump is used, the trace cache mechanism is not able to check the target address correctness, either such option need to be added or trace must end in such cases.

A more subjective example is function calls, where it might be wise for a new function to start at a new trace, thus making the trace have an even better chance of being reused. Another idea could be to try and finish traces in a branch instruction when possible (a basic block), doing so may create traces that will have better usage however it may cause for more unused data areas in the trace cache.
Some trace caches may choose not to continue a trace after a taken backward-branch (in such way to have only one copy on 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); such ideas were presented in [4][8].

- **How to access the trace cache** – in most works dealing with trace cache a specific trace is asked from the trace cache when accessing it, doing so calls for 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 option is a special trace prediction mechanism that tries to predict the next trace as a whole according to previous traces which were used.

  In our work we chose to build traces in such a way that at most a single trace can start at a certain base address and no trace predictor is used. Doing so made the trace selection simpler and also allowed the performance to be independent of the quality of the predictor. Such fetch mechanism does not suffer from the complexity of a prediction mechanism. Later on in the paper we will compare the use of our proposed trace cache with a trace cache that allows multiple traces to begin at the same address (and using trace predictor). We will show that our trace is comparable to the more complicated structure.

- **Can a trace be aborted in the middle** – in order to support such quality, addresses of inner branches must be saved, however it could allow the machine to use partial matching when fetching a trace.

### 4.3. Trace predictor

As mentioned before in many cases the machine would prefer to access the trace cache knowing exactly which trace it wants to fetch, one such mechanism is the trace predictor as presented in [10].

The trace predictor differs from a multi branches predictor in the way it relates to a trace as the basic unit rather than the branch instruction. The trace predictor predicts the next trace according to the last N traces which were fetched.

A trace is indicated as a base address plus a directions vector. The last N traces (address plus directions) 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 in which the next trace is presented (if the entry is valid).

A 2 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 a new trace will replace it.

### 4.4. Basic observations

Although not mentioning the simulation environment yet, we wish to show here the performance of out of order machines using trace cache in comparison to a machine without such a component. The simulation environment will be discussed in part 5.

![Figure 4-1 – Compared results of machines: with no TC, with TC but no Tpred, with both TC and Tpred. The results are distributed by trace cache size (in sets)](image)

Figure 4-1 shows performance results of three machines; the first machine uses a trace cache with a trace predictor and allows traces with the same base address to be saved in the trace cache at the same time. The second machine also uses a trace cache; however it fetches traces only according to a base address (no trace predictor). The last machine does not use a trace cache at all but it has double the size an instruction cache.
The trace caches are rather small; their sizes go from 4K to 16K (plus a 256K one), while the instruction cache of the regular machine uses 256K (128K in the machines which use trace cache). The instruction cache while being bigger is also considered slower (3 cycles for first access in comparison to 2 of the trace caches) and is organized in 4 ways. The trace caches have associativity of 4 as well.

The results represent relative CPI of 13 different benchmarks. It can be clearly seen by Figure 4-1 that using a trace cache was always advised (with the exception of the big trace cache of 256 traces); even when using the smallest trace caches, both machines using trace cache managed to give better performance as opposed to the machine with the bigger instruction cache.

When comparing the machines using trace cache, the trace predictor manages to give much better results rather than the simple trace cache which fetches according to a base address. The improvement when using the trace predictor goes from 13% up to about 20% in the trace cache of 64 traces (16 sets * 4 ways).

In the big trace cache of 256K the comparison to the regular instruction cache is not a fair one since the total size of both caches is much larger. This trace cache is brought here in order to compare the machine using a trace predictor to the machine that doesn't. The machine with the trace predictor keeps improving while the regular machine receives its worst performance; this behavior is one of the main motivating reasons for applying filters in the trace cache and will be explored further in chapter 7.

We will show later, in chapter 7, that when using dynamic filters in the primitive trace cache its performance improves a lot; we will compare then the same machines once more and show how the advantage of the trace predictor disappears.
5. Simulation Environment

5.1. Simple scalar

The simulator used in this study was derived from the SimpleScalar 3.0 tool set[11][12]. We used the Sim-Outorder mode to simulate an Out-of-Order model of an 8-wide machine. The model was enhanced in order to simulate a trace cache; then a hinting mechanism and some dynamic trace filters were added.

The basic configuration parameters that were used in the simulations are presented in Table 5-1:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Queue</td>
<td>8</td>
</tr>
<tr>
<td>Branch prediction</td>
<td>Gshare</td>
</tr>
<tr>
<td></td>
<td>History 13</td>
</tr>
<tr>
<td></td>
<td>Table 8K</td>
</tr>
<tr>
<td>BTB</td>
<td>4x512 entries</td>
</tr>
<tr>
<td>Instruction L1</td>
<td>Block size: 64</td>
</tr>
<tr>
<td></td>
<td>Sets: 128</td>
</tr>
<tr>
<td></td>
<td>Associativity 4</td>
</tr>
<tr>
<td></td>
<td>Replacement LRU.</td>
</tr>
<tr>
<td></td>
<td>Access time: 3</td>
</tr>
<tr>
<td></td>
<td>For the same block: 1</td>
</tr>
<tr>
<td>Trace cache</td>
<td>Block size: 256</td>
</tr>
<tr>
<td></td>
<td>Sets: 4 – 256 (tested)</td>
</tr>
<tr>
<td></td>
<td>Associativity 4</td>
</tr>
<tr>
<td></td>
<td>Replacement LRU with / without filters (tested)</td>
</tr>
<tr>
<td></td>
<td>Access time: 2</td>
</tr>
<tr>
<td></td>
<td>For the same block: 1</td>
</tr>
<tr>
<td></td>
<td>1 trc per base address</td>
</tr>
<tr>
<td>L2 Unified cache</td>
<td>Block size: 64</td>
</tr>
<tr>
<td></td>
<td>Sets: 4096</td>
</tr>
<tr>
<td></td>
<td>Associativity 4</td>
</tr>
<tr>
<td></td>
<td>Replacement: LRU</td>
</tr>
<tr>
<td></td>
<td>Access time: 14</td>
</tr>
<tr>
<td>ITLB</td>
<td>Page size: 4K</td>
</tr>
<tr>
<td></td>
<td>Sets: 16</td>
</tr>
<tr>
<td></td>
<td>Associativity 4</td>
</tr>
<tr>
<td></td>
<td>Replacement: LRU</td>
</tr>
<tr>
<td></td>
<td>Miss penalty: 30</td>
</tr>
<tr>
<td>I-ALU</td>
<td>16</td>
</tr>
<tr>
<td>I-DIV</td>
<td>4</td>
</tr>
</tbody>
</table>

More information on the Simplescalar simulator can be found at the simple scalar website [13].

We will elaborate on the fetch mechanism, with a glance to the memory sub system (in the aspect of instruction caches) and the branch prediction mechanism.
The simulator, when not using a trace cache, fetches instructions until one of the following accrues: the instruction queue is full (it fulfilled the machine's capacity), there is no more room for instruction in the next pipe stage (RUU is full), there is a branch instruction predicted to be taken or more branches then the machine can predict while fetching (in our case we do not allow prediction while fetching therefore any branch instruction will cut the fetching sequence), there was a change of cache lines.

When the machine receives a cache hit for the first time there is a latency of 3 clock cycles, any other access to the same cache line will not cause any penalties.

We used Gshare 2 level branch prediction with a table of 8192 entries (2 K-Bytes); the table is accessed using the XOR function between the history and the branch address in order to get an index.

The prediction is available in the fetch stage and is not considered to create a penalty (that is aside from stopping the fetching sequence), after a branch is resolved the result is sent to the branch prediction mechanism at the writeback stage.

5.2. The trace cache

Each instruction being 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 status. When a trace ends (too many branches, too many instructions, a function call/return, indirect jump) it is sent to the trace cache (without checking if it fully commits) where the filters may still prevent it from being actually saved, this part will be explained later. If there are 2 trace caches, one is oriented for hinted traces, in such a case a special logical unit decides where to send the trace to (according to the last hint that was received).

The trace cache can get feedback from the writeback stage in the form of a report whether a branch was mistaken, and feedback from the commit stage to report which traces have fully committed. In order to support these feedbacks each instruction should have an indication for which trace it belong to, we assume this is possible in our machine via special buffer that contain the tags of the traces that are in the machine and pointers from the ruu (Simplescalar version of the ROB).

The trace cache can either use this feedback or not, depending on the mechanisms it uses as will be explained later as well. When trying to fetch after a trace was finished both caches are searched (trace cache and Instructions cache) in any case of both hitting the trace cache gets priority (the trace cache is accessed by a base address alone), if there are two trace caches, unless overriding the hinted traces get priority.
Figure 5-1 shows the simple scalar basic pipe, and how the different caches are integrated, the feedback lines are not drawn in this schematic.

5.3. The Filters

As part of the machine description we will explain how the filters and hint mechanism are implemented. These sections are kept in a basic level in order to be understood at this point; however they will probably become clearer after reading part 7 describing the filters and part 0 dealing with hints.

There were two different types of filters used in this work, one is a usage filter which is located outside the trace cache (not a part of the cache) and the second is an inner trace cache mechanism which filters the traces that have already been saved in the cache. The usage filter follows the reappearance behavior of traces; when a trace is being used often enough the filter will let it through. The mechanism inside the trace cache works more like a cleaner, unlike the usage filter it does not try to prevent traces from getting into the cache but removes traces which are considered not useful.
Figure 5-2 describe the trace cache and its filters. The usage filter (1) is basically a table which contains tags, in our tests we used a table twice the size of the trace cache it filtered for. Each entrance (2) in the table represents a trace; as such it must contain a base address, a directions vector for the branches, number of branches and a usage counter. The table is organized as a cache (sets, ways) and is built the same way as its trace cache (same associativity), when a trace is first inserted to the table its counter is set to 1, every time the filter receive an already existing trace it will add one to the counter of that trace. Traces should be evicted from the usage table the same way as they would have in the trace cache (LRU or new trace with the same base address), trace that is re-inserted will start a new counter.

The inner trace cache filters are supported by the feedbacks mentioned before. A trace (3) in the trace cache contains: the instructions (the trace itself), a base address, a directions vector of the branches, number of instructions, number of branches, a counter and the number of the last branch to mistake (inner count for each trace). The last two fields are used by the cut/kill mechanisms. The counter is meant to count branch mistakes, it is a 2 bit counter set for 2; every mistake decreases the counter by 1 and if the trace fully commits, the counter is increased by 1.

There are two options to deal with a trace who's counter reached 0, one it to evict the trace from the trace cache (kill it) and the other is to cut the trace at the branch that caused the mistakes.

When the cutting technique is being used the counter works a bit different, instead of counting mistakes for the trace as a whole it counts mistakes per branch instruction. For implementing it, the "last branch to mistake" field is used; this field remembers each time which branch was the last to cause miss-prediction, when there is no such branch it will be set for zero. The counter will keep on behaving the same way as was described before as long as it is the same branch instruction that makes the mistake. If a new branch is mistaken the last branch field will update and the counter will be reset.

When the counter reaches zero, in the regular killing mechanism the trace will be removed from the cache; in the cutting mechanism all the instructions after the mistaken branch will be removed and the trace will stay in the cache (the counter will off course be reset), this can easily be done by changing the tag fields (number of instructions and branches) and therefore should not take extra time.
In the merge scenario, the trace cache attempts to merge each trace to the trace before. The only exceptions are indirect branches which might jump to different locations each time, once placed in the middle of a trace (where the address isn't saved) the machine won't be able to check if the target address is correct. Our merging mechanism also support some fine tuning features, beside merging everything it is possible to merge only branch instructions (not to merge function calls) or to merge function as well if it is a complete function (dynamic inline).

If the merging is possible, meaning there is enough room in the last cache line to hold the new trace (number of instructions and branches), the new trace will be copied to the last cache line. If the new trace is a trace fetched from the trace cache (not a new built one) the copied trace will remain in the trace cache as well, if no other path will use this trace it will be removed in the normal LRU way. As for the fear from bad merges, since the machine merges everything blindly, the cutting mechanism should be able to fix it later on.

![Figure 5-2 - A focus of T-cache with filters](image)

<table>
<thead>
<tr>
<th>base address</th>
<th>trace direction</th>
<th>branches number</th>
<th>instructions number</th>
<th>Instructions</th>
<th>kill counter</th>
<th>number of mistaken branches</th>
<th>(1) Usage table</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td></td>
<td></td>
<td></td>
<td>(1)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(2)</td>
<td></td>
<td></td>
<td></td>
<td>(2)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(3)</td>
<td></td>
<td></td>
<td></td>
<td>(3)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
5.4. The Hints

The hint instructions are inserted to the code by the compiler in order for the machine to catch them and decide how to use them; Figure 5-3 shows this part of the system.

When fetching a hint instruction the hint decoder (1) figures out what kind of hint was fetched and transfers it to the hints control unit (2), this unit can either bypass the branch prediction mechanism when it is a build hint, or remember the hint for later use if the hint is used for filtering (which trace cache to send it to, not to save the trace at all). When a build hint is used, the hint provides a branches vector meant to bypass the branch prediction mechanism while building the trace. If the machine chooses to use this hint the prediction for the branch instructions during the building process comes from the hints control unit.

The hints are served in order of appearance (first come first served). Hint instructions are not executed by the machine; they are used only to enhance the functionality of the build unit (in our example). Thus, when the trace is built, the hint instruction is not saved as part of the trace, allowing the machine to avoid the overhead of executing extra instructions.
5.5. Condor – testing environment

In many researches and projects the quality of the research is heavily dependent on the available computing power [14]. Such work requires an environment of a High-Throughput Computing (HTC), where a large amount of computational power is available for a long period of time.

Years ago such work was done by a centralized mainframe or a supercomputer, when users had to wait for their turn to use the mainframe and had limited amount of time allocated.

As computers became smaller, faster and cheaper users moved from centralized mainframes to personal computers, the personal computer is slower then a large centralize machine but allows personal access for the user.
For an institution having many personal computers, the total computational power as a whole may rise dramatically, however, because of the distributed ownership individuals can not use that power.

As a matter of fact many personal desktop machines sit idle for a very long period of time while their owners do not use them, hence making the machines computational power go to waste.

Condor [14] is a software project created by the University of Wisconsin-Madison; it is a specialized workload management system for computer-intensive jobs. Like other full-featured batch systems, Condor provides a job queuing mechanism, scheduling policy, priority scheme, resource monitoring, and resource management. Users submit their serial or parallel jobs to Condor, Condor places them in a queue, chooses when and where to run the jobs based upon a policy, carefully monitors their progress, and ultimately informs the user upon completion.

The Condor software system creates an HTC environment by using the desktops machines which aren't in use at the time. The system receives jobs and distributes them to available machines, when a machine stops being available the job can be relocated to a different machine.

When a single program needs to be run many times with different inputs or parameters, Condor provides a convenient environment to easily submit the job and will automatically make all the tests run whenever there is a machine available that meets the job's requirements.

Each job being sent may add requirement regarding the memory amount it needs or operating system or CPU type etc., then all available computers report to the system via Ads of their status and what can they support. The condor system matches jobs to computer also taking into consideration jobs priority, waiting time and executing time.

In this research, due to the enormous amount of tests done and the fact that each test takes over an hour to complete, all tests were send as jobs under condor system running in the Technion - computer science, lab for distributed systems.
6. Profiling

6.1. Introduction

Feedback directed optimization (FDO) [15] is a common way for optimizing a system based on run time behavior. The method calls for running an existing code using typical inputs and collecting statistics. Based on the information collected at those runs, the program is optimized again with special attention to the parts of the code which were often used; these parts may have a good potential of performance improvements – hence making the common case run faster.

Many compilers use FDO after compiling a code, they run the compiled program with training inputs collecting information about the runtime behavior of the program into a profile image; the information is then used by the compiler to regenerate the code, often making the frequently executed paths optimized more efficiently at the expense of the other paths.

The efficiency of program profiling is mainly based on the assumption that the characteristics of the program remain the same under different runs as well.

Many works showed that using a profile image while compiling the code can help improve programs performances; in [15] it is shown how compiler optimizations based on profiling in Compaq's compilation tools for Alpha can improve the results received by a very powerful and complete classical optimizer.

In [16] profiling image is used to improve the value prediction mechanism by hinting in advance on which values predicting is advised. Another work [6] uses profiling to classify traces of instruction code and test trace cache results when it can use the collected knowledge.

FDO can be used by a compiler in order to rearrange the instruction code. The trace scheduling [17][18] uses profiling information in order to compact the program parts that are used more frequently while spreading the less used parts. Another example is the software trace cache [19] where profiling image was used to rearrange the compiled code in such a way that it will cause the branch instructions in the program to tend not to jump.

Some newer researches study the effect of doing profiling dynamically via hardware [20][21], in those cases information is being collected at run time into a special hardware unit that put it all together and analyze it, once specific code is considered by the hardware to be important enough then it is subject alteration and optimization. Similar process is being done in the complex filtering examples [4][7][8][22][23] (those works will be
discussed in the Dynamic Filter part - chapter 7), once going toward dynamic profiling the question need to be dealt with is how deep should the profiling be and then how much is it worth paying for in power or performance.
In our research we used the profile image in order to create hint instructions; future research could try to use dynamic profiling techniques in order to insert hints dynamically into the code.

6.2. The profiler structure

Our profiler is meant for collecting statistics about traces, it collects all the info into a dynamic data structure and when the program ends, either save it to a file or print a summary as will be explain later on.

We will now elaborate on the data structure and its fields for future use.

The database of the profile is made of general info regarding the trace cache and a hash table of traces. Each entry in the table is a pointer to a list of traces; traces are being inserted to the head of the list or updated when already in the list, the hashing is being made according to the trace base address modulo the number of entries in the table.

The profile image structure is as shown in Figure 6-1; the first part (cache info) is a copy of the cache info as it is saved in the cache data structure of the Simple Scalar; the fields: bsize, balloc and usize have no meaning what so ever and only exist for compatibility reasons.

There are two sets of three fields each; every set has a counter which counts how many traces were accessed, and two average fields: one for average number of branches per trace and the other for instructions per trace. The two sets differ by whether the access being considered was for reading (cache hit) or writing.
The structure holds a pointer to the hash table of the traces.

The traces (called trace_node_header) are each made of (in order of appearance):
- A trace (branches directions), a binary vector indicating whether each branch in the trace was taken (1) or not taken (0) starting from the LSB.
- A base address, the address of the first instruction in the trace.
- Number of instructions in the trace.
- Number of branches in the trace.
- Number of times a certain trace was searched in the trace cache but was not found. This field is not relevant for most of this work experiments since in general a trace predictor was not used and traces were accessed only by a base address, in any case such field is supported for future works.
- A pointer to the next trace.
- A list of instances of that same trace – this is a list of nodes for every time the trace was in the trace cache, each node is created when a trace is saved in the trace cache and it is considered open as long as the trace stays in the cache. Once the trace is either removed from the trace cache or altered the node is closed and next time the trace will be in the cache a new node will be made for it. The information being held in each such node will be elaborated later.
- An information table of the branches in the trace – this table has an entry for each branch of the branches in the trace; each entry indicates what the branch instruction number inside the trace is. The second field plays a double role, it is meant to count the number of times the branch was mistaken (the direction used from the directions vector was wrong) this can help learn about weak links in the trace. However when using the Oracle mode (a perfect branch prediction) there is no point for such field, therefore it was used to save each branch next address, doing so can help find overlapping situations and loop unrolling cases.

For every time the trace was built and saved in the trace cache there is an instance of trace_node for that time. These nodes are held in a list of the trace_node_header and being inserted from the head of the list.

Each node has the following information (in order of appearance):
- Number of times it was fetched; not necessarily fully fetched but there was a cache access asking for that trace.
- Number of times the entire trace was committed (its last instruction was committed).
- Number of times the trace had a mistake in one of its branches (had an inner miss prediction).
- Number of times a trace which exist in the trace cache was rebuilt; once again this field makes much more sense when using a trace predictor.
- The time (clock ticks from the beginning of the test) it was built (saved in the trace cache).
- The last time it was used (clock ticks from the beginning of the test).
- The time it was removed from the trace cache (clock ticks from the beginning of the test).
- The trace merging (see Dynamic Filters – chapter 7) status – this field indicates whether the trace was created by merging two other traces, or died by being merged to another trace, or both. These definitions are somewhat different from the regular ones of a trace being created or being dead. That is since in both cases the trace is not removed nor created from scratch like a regular trace; however from a profiler point of view this is the best way to describe the situation.
- The trace cutting (see Dynamic Filters – chapter 7) status – this field indicates whether this trace was killed being cut or born after another trace was cut into it. Again it means being killed or created in a profiler point of view. Technically speaking the long trace no longer exists, even though the shorter one is still a part of that same trace and no eviction was made from the cache. In the same way, although no new trace was built, the short trace is still a new trace and will be considered as if it was just created.
- A pointer to the list of the trace previous instances.
6.3. Collecting the information

Every time a trace is being built it is inserted into the profile structure, if it is the first time the profiler sees this trace it will make a new trace node (trace_node_header) to be inserted first into the linked list of traces. The node will also have an instance (trace_node) which will represent that specific period of the trace in the trace cache, while the header will contain the basic information about the trace (information that in most cases won't be changed). The instance node will be updated with the information of the time the trace was formed (written into the trace cache), the same time will be used as the first and last time it was fetched; the number of times the trace was fetched will be set for 1 and times it was committed and mistaken (had one of its branches going the wrong way) to 0, since the
trace was built in the normal way it is considered not to be an outcome of a merge nor a cut.

If it is not the first time this trace is formed then a new instance will be added to the list of instances of that trace at the top of the list and will be initialized with the same values as mentioned before. The header node representing this trace will be moved to be first in the traces list for locality reasons.

Every time a trace is being fetched from the trace cache, its header node will be found and the first instance of its list, representing this lifetime of the trace, will be updated. The fetches counter as well as the last time fetched field will be updated. The same procedure will be made for every time a trace is being committed, and when one of its branches is discovered as a mistaken one, in both cases the appropriate counters will be updated. In the second case if it is not in the Oracle mode, the counter of the specific branch in the branches table (node header) will be updated too.

When a trace is being removed from the trace cache, the instance representing the lifetime of the trace exists no more and therefore the time_killed field of the instance node will get the time and the node will be considered closed, thus the next time it will be accessed it will be via a build only (in any other case it is a bug).

When a merge is being done, suppose trace1 is being merged with trace2 (trace1 is chronologically first) forming the merged trace. In such case, after the merge in the trace cache, the first trace no longer exists since the trace cache can't hold two traces with the same base address; therefore the profiler will consider trace1 as if it was removed from the trace cache and will kill it, the field merge_status will be set to either 1 or 3 indicating it was killed by a merge (3 indicates it was created by one as well). The merged trace will be treated as if it was just built and be added as a new trace or a new instance of an existing trace, it will however have its merge_status field set as 2 indicating it was born as a result of a merge.

As for the second trace it remains untouched since the trace cache does nothing with this trace as well (except copying it to the end of trace1).

When a cut is made, the trace that was cut no longer exists in the cache and therefore its instance in the profile is being considered as if it were removed from the trace cache. The
new and shorter trace is a new trace for the profiler and will be inserted to the profile as a
trace that was just built; this new trace will have its cut_status field set to 2, indicating it
was born in a cut. The longer trace which was cut will update its instance node with the
time it was killed but also that is was killed by a cut, meaning that the cut status field will
either be 1 – just killed by a cut, or 3 – born and killed by a cut.

6.4. SimpleScalar profile info

The SimpleScalar expect a parameter in the shape of "–profile_res XXX" to tell it what to
do with the profile image. The default is to do nothing and free the memory taken by the
profile structure once the program ends.

If the results are to be processed later on there are two ways to get the information: one
way is by asking the simple scalar to print the results of the profiling to the standard
output, in this case it will do a printout of all the traces being handled by the profiler sorted
by their base address and summarized into lines. Each line gives the basic info of the trace
(everything there is in the trace header) plus a sum of all the counter fields in the trace
instances, for example the number of times the trace was committed totally regardless of
the different instances. The Simple Scalar will also print the sum of the trace instances, i.e.
how many times it appeared in the trace cache.

Printing to the standard output makes it easy to split the results of a certain test into two
files, one for the profiling info and the other of the test statistics. This is thanks to the fact
that the Simple Scalar statistics are being printed to the standard error output. The regular
output of the simulation (made by the tested program) needs to be removed from the
profiling file though.

The second option is to save the entire structure into a file; as such all the information
regarding the different instances of the traces is being saved, allowing deeper analysis to
be made later on. The down side of doing so is the file size which can be quite large; it
turned up to be too large for our resources and therefore wasn't used for this work. The file
can be easily restructured to a profile structure via already made C functions and then any
static analysis can be done.
6.5. Creating build hints

Since the entire profiling structure is very large and as such was beyond our resources reach, we used the summarized printed version. This profile image could tell about each trace how many times it was used and built, as well as how many times it committed. It is also possible to get the number of times the trace had a branch miss prediction and even which branches made most of the mistakes.

However we do loose the time factor and therefore can't figure out locality of the traces, this mission was decided to be too big for our work and could be a subject of future research; it was shown though in other researches [15][6] that with good profiling analysis it is possible to determine locality of the code.

Since working with the profiler and the compiler for achieving a deep analysis of the code behavior could be an entire research for itself, and lacking neither the time nor tools to do such a job, we were satisfied with a shallow analysis in order to show potential of the method.

We used the summarized profile image of tests made on the oracle mode of the simulator and put them in an excel file; the results were sorted by address first and then number of times the trace was used. The table was programmed to check competition between traces with the same base address and low use of traces in general. In order to call a trace "competition free" we demanded it would be used 8 times more then the other traces with the same base address. Then we removed traces that although being used a lot are seldom built by putting a threshold to the number of appearances.

At last, in order to be able to mount the profit which might be gained from saving each trace in the special trace cache we made a simple price function that added the number of taken branches to each time a trace is built; every appearance of a trace in the trace cache was worth a point and each taken branch in such appearance would add another point.

The motivation for such function was the assumption that jumping to another cache line while building a trace will add penalty.

Eventually the traces with the highest price were taken and inserted to the code as build hints.
7. Dynamic Filters

7.1. Introduction

The idea that a small part of the code is executed a substantial part of the running time is long been discussed, it is also a known fact that smaller caches work faster and consume less power, therefore the idea of saving only the small part of the code which is used a lot in a small and efficient cache is the logical next step.

When using a trace cache which saves traces of instruction code, the question that needs to be dealt with is how to filter only the desirable few traces to be saved in the trace cache at the expense of the rest of the code which is seldom used.

One idea for such selection was doing it via software. In the Dynamo project [22] there was a use in a kind of simulated trace cache; while the hardware continued to use a regular instruction cache, the stream of instructions were monitored by the software. Once a trace was used frequent enough to exceed a given threshold it was inserted as a continuous trace of instructions and stored in a special part of the memory. Then, when the same stream of instructions was fetched again, the program directed the fetch address to the place in the memory it used in order to save that trace; at the end of the trace the IP address was rerouted by the program to continue fetching from the code segment. The traces saved in the special memory are optimized and the size of this memory is flexible, unlike as in a hardware trace cache.

In software trace cache [19] the same idea was promoted; although using a regular instruction cache the traces were made via software. However in this research the changes in the code were made statically prior to running the program. A profiler was used in order to determine which directions of the branch instructions are more common, after doing this analysis the code was reordered in such a way that the common choice would be not to jump. [24] was a following research that given a code made by the software pipeline technique the machine will use a trace cache but will save in it only traces with taken branches. In such way, the static work made by the software, filtered many traces for the trace cache by making most of the hot instructions streams sequential; in this research it was assumed that there is no penalty in reading instructions from the regular instruction cache rather then the trace cache.
In [23] there is a use of a physical trace cache, however it uses the benefits of the Dynamo project by having a java virtual machine which helps manage build and filter the traces; after being sorted the traces are saved in an actual trace cache (rather then in a different part of the memory as was done in Dynamo).

The other approach is to use dynamic instruments and filters in order to sort out the traces. In [4][8] there is a use in a decoded trace cache working with a CICS architecture. It is suggested in [4] to have two trace caches, an FTC and an MTC; traces are saved in the FTC and once a trace is used often enough it is copied from the FTC to the MTC thus preserving the traces that are hot for longer periods of time. In [8], as a continues work, they present the PARROT concept where the hot traces in the MTC aside for being saved longer are subject to dynamic optimizations.

In the replay framework [7] the machine saves only a few but very long and hot traces then it performs optimizations on these traces (called frames in this work) in such a way that the traces become atomic units and can not recover from a miss prediction. A trace's instructions either commit together or not at all, doing so allows the machine to perform more powerful optimizations but the traces should be carefully chosen.

In the sampling filter [5] the traces are filtered randomly when only once in a while a trace is able to be saved in the trace cache. Since the hot traces are used a lot, eventually they will be saved in the trace cache and will be able to stay there for a long time (since there is less traffic in the cache).

We will now present the filters we have tested; we assumed in our work that the code is a given and no changes are done to the code. For the filters part, the compiler has no role whatsoever.

We also did not use any dynamic optimizations technique on the code saved in the trace cache, the code stays as is for the entire program run.

### 7.2. The filters which were tested

Given the chosen machine we thought of a few techniques in order to filter unwanted traces from the trace cache.
This chapter will go over the different techniques; we will show the motivation for each one and point the advantage and disadvantage of using them. The following chapter (7.3) will discuss the results and conclusions from the tests which were made.

**Check each branch:**

Our machine fetches simultaneously from the trace cache and the regular instruction cache, the fetch is made when the machine is not reading a trace nor building one, it means that the caches are checked only when a new trace is about to start. The machine checks both caches because it uses a small trace cache and will probably need to build a lot, therefore the penalty for doing the check sequentially will be too high.

The problem is that when checking the trace cache only between traces, the machine might miss opportunities to fetch already existing traces from the trace cache. Assuming that reading the same path of instructions from the trace cache rather then the instruction cache should allow the machine to fetch more instructions each cycle, it might be wise to check the trace cache even before the building process is over in order to see whether the machine is reaching a path of an already existing trace.

Since our trace builder stops building traces only at basic blocks whenever possible, checking the trace cache only after branches should do the work.

This idea is supported by [25] where it indicates that one of the trace cache problems in achieving better performance is the duplication problem. This problem occurs when the same basic blocks are being saved a few times due to traces which start elsewhere each time but reach the same area of the code.

The main drawback of checking the cache at each branch (if we do not count power issues) is that the machine might be building small traces; that is because building a trace could be interrupted by a hit in the trace cache (another trace was found) and therefore the building process is stopped.

As a result, although duplication is prevented by checking the trace cache at each branch, inserting small traces harm the trace cache utilization as well.

When using the merge technique, as will be explained next, this problem is solved and therefore these two techniques should go together. The interesting part is that checking each branch with other filters as well as with no filters at all in general didn't harm the results and sometimes even improved them.
Cut & Merge

The last technique of checking the trace cache at each branch as well as the regular process of building traces may cause for short traces to be saved in the trace cache.

Short traces in a trace cache containing relatively large cache lines are not welcome events; these traces cause for a waste of cache data area which makes its utilization lower.

The problem of having short traces could be solved by (a) filtering them (not to save them in the trace cache) (b) not building such traces (allowing duplication) (c) trying to merge sequential traces.

Not saving small traces is not always a good idea. It could be that the trace is often used and then although being small, having it in the trace cache could yet improve performance due to alignment reasons; for example, when the small trace is being positioned in between two cache lines (in the regular instruction cache), the trace cache will put it at a beginning of a new cache line. Such filter was tested also and will be discussed later.

In order not to build such traces the machine can avoid checking the trace cache at each branch as well as bring the rules for ending a trace to the minimum requirement (end just at indirect jumps). In general doing so didn't improve performance, the damage made by rebuilding already existing paths even caused sometimes for a drop in performance, especially in the smaller trace caches.

The idea of merging traces dynamically in its simplest form means that the trace cache checks any two sequential traces to see whether merging is at all possible; merging is possible if there is enough space for the instructions of both traces in one cache line and the number of branches in them both is supported by the trace cache for a single trace.

Once the trace cache discovers that merging is an option it has to decide whether it is desirable; sophisticated checks can be made here like how often the two traces come together (this will call for more memory to save history), how many other traces are followed by the second trace, and so on. In this work it was decided to keep it simple and merge whenever possible; complex merging mechanism could be investigated in future works.

Another advantage of the merging technique is its ability to transform dynamically a function call to an inline function when the function body is quite short. The machine can do that by acknowledging that there was a function call between the two traces and that there is enough room to merge those two. Such a merge can drop the call/return protocol
thus letting the trace cache enjoy the benefits of inlining without having the regular instruction cache pay for it.

Most trace caches stop building a trace while hitting a call instruction, mainly because the new called function might be used in other places as well; in such cases a new trace for the new function could make the trace available for function calls in other places too, thus improving the usage of the trace. When using a merging mechanism it is possible to know whether the function is short enough to fully enter the previous trace, a more complex technique could try to collect statistics and see from how many other places the function was called; having this information can help decide whether the merge is desirable.

While having long traces is very good for better utilizing wide machines, it may insert into the cache traces that mistakes more often (many branches equals more chance for mistakes). The machine might discover late that the merge was not a good one, especially in our machine which does not collect information before merging. It is also possible that merging only half the trace would have been better. This could also be the case when building a trace the regular way when dealing with long traces; while the builder builds as long a trace as possible, it could be that the first few branches of the trace often behave the same while the last one is unpredictable. In such traces it might be in the machine's best interest to keep the trace shorter but safer, the branch prediction mechanism might handle that problematic branch better.

When using a machine without a trace predictor it is important for the machine to be able to protect itself from bad traces. Regular LRU system that removes the least recently used cache line might help bad traces to stay in the trace cache because they were frequently fetched although no fully executed (committed). Such traces will be rewarded for merely being accessed; if the trace tends to make many mistakes it serves the opposite purpose, an address which is often fetched will constantly be using a bad path that will keep being in the trace cache for being fetched again and again.

In order to solve this problem the machine can kill bad traces (remove them from the trace cache), another idea can be to try and avoid having such traces in the trace cache by saving only traces that keep on being built, hoping that the branch predictor will not tend to build the same bad traces over and over again. Both techniques will be discussed later.

One thing that is common for those techniques is when having a trace that has a wrong predicted branch, they both will punish the entire trace for that mistake; the trace as a hole is being "judged" as a successful one or not. As mentioned before, a long trace has more
chance for mistakes, therefore the cut and merge technique "punishes" the bad branch rather than the entire trace. The trace cache keeps track of which branch causes the mistakes, when it reaches a certain threshold the trace will be cut at the bad branch, leaving the trace shorter but hopefully better.

Using the cutting technique with the merging mechanism can help the machine to dynamically form the best traces for a certain base address; section 7.3 will show the experimental results of this technique.

Kill

As described in the cut & merge part, a different way to approach the problem of bad traces in the trace cache is to remove traces which often cause mistakes. When using this filter, each trace in the trace cache uses some kind of a mechanism in order to get feedback regarding its performance, the mechanism should check whether the trace was fully committed or not. The easiest way to do so will be via a counter that goes up when the trace is committed or down when it makes a mistake, each counter is set to an initial value and has an upper bound, when the counter reaches zero the trace is removed. In this work we used a 2 bit counter which was initialized to 2.

Another option can be to combine the cutting and the killing techniques by deciding how deep we want to allow cuts to be made. In such case there is a certain point defined by number of instructions or percentage of instructions from the entire trace, where traces aren't cut anymore since they are too small, instead they are removed from the cache.

Usage filter

The third approach to deal with bad traces is not to insert traces until they are proved "worthy"; when using this filter, traces are built as usual but they are directed to the trace cache to be saved only after being built several times. This filter uses a cache-like memory which only holds the tag directory information; we will refer to it as the usage table. Each trace after being built is first checked by this usage filter, if it is a new trace its information is added to the usage table, if it already exists its counter is updated. Each entry in the table has its own counter to indicate how many times a trace was used; once the counter exceeds a certain threshold the trace is directed to the trace cache to be saved.
A similar technique was used in [4] or [7], as well as in many researches where some parts of the code were subject to dynamic optimizations, more example can be found in [8][22][23].

A few decisions must be made when using this filter:

- What is the size of the usage table?
- What would be the threshold?
- Whether or not it is allowed for the table to have few traces with the same base address. This option stands by its own regardless of the trace cache, which in our case does not allow such a condition. It could prove to be better for the usage table not to be effected by rare paths exceptions.

There were few techniques that were suggested and tested in early stages but did not show good potential and therefore were not further examined, here are two of them:

**Small traces**

It was suggested that small traces are a waste of cache space and might not be too productive when using wide machines. The idea was to filter those small traces by putting a minimum number of instructions for traces in order to be saved in the trace cache. This filter was initialed with a threshold number and any trace shorter then this number was not saved.

Small traces might be a product of a function call, an indirect jump, long basic block, finding an existing trace (when using the Check each branch mode) or a trace cut (when using that mode).

This filter was tested with several threshold numbers however results tended to be worse than without that filter and therefore the idea was neglected. We believe that there were too many small traces in between cache blocks that made it harmful not to save them at all (each build took some extra cycles).

The merging technique is trying to fix the problem of small traces by merging them to the traces that came before or after, when doing so it better uses the trace cache data area without having to fetch those small traces from the instruction cache. The merging
mechanism showed much better results and seemed like a better solution to such a problem.

**Zero branch traces**

Another filter which was tested was to filter traces that have no branches from the trace cache; fetching such traces from the regular instruction cache should have the same efficiency, as suggested by [24].

This proved to be a wrong assumption in our case; it would appear that having the trace start in the beginning of a cache line (i.e. having basic blocks aligned with the cache lines) has much impact on the performance.

Therefore we came to the conclusion that saving traces with no branches isn't futile at all, on the contrary, it is another way the trace cache can help a wide machine fetch more instructions each cycle by dynamically rearranging the code as cache friendlier.

### 7.3. Results

We performed a set of tests in order to see the influence of the different techniques as a factor of the trace cache size. The basic configuration of the machines we used is an 8 wide out of order machine which has a 32KB size of instruction cache of 3 cycles latency for the first access (second access to the same block is 1 clock cycle). The traces were all 64 instructions long and contained 6 inner branches at most, the trace caches (being smaller) were considered to have 2 cycles latency for first access (we kept this latency when using bigger trace caches as well, in order to make them all comparable). More of the machine parameters and different parts can be found in the Simulation Environment section – chapter 5.

The first thing that emerged from the collected data is that using no filters at all may prove to be pretty harmful; it gets even more so as the trace cache size increases (as can be reflected in Figure 7-1). At first sight it might appear weird that the performance decreases as the trace cache size increases; however, as mentioned in 7.2, this behavior occurs due to the fact that normal LRU considers a hit in the cache as a good event and therefore makes the cache line that was asked for stay longer in the cache. This assumption is indeed true when using a regular cache in which the cache lines are chunks of memory; when these chunks are used more often they are parts of the memory which is truly desirable in the
cache. This is not the case when referring to the model of a trace cache used in this work, the ground rules are somewhat different and finding a cache line in the cache is not enough.

A cache line in the trace cache is a trace of instructions constructed out of basic blocks connected by branches, once a hit in the trace cache occurs it means in our case that the base address of an existing trace was searched by the fetching unit and found in the trace cache, however it is uncertain yet whether this trace is the right path to choose.

In fact, a trace starting in a certain address could be quite a poor choice of branches, and might cost the machine dearly in miss prediction penalties. The problem with the LRU mechanism is that it might take a trace with a "popular" base address and reward it each time it is being accessed no matter how many miss predictions it causes. As a result, a bad trace might stay in the cache for a long time and cause penalties each time it is fetched.

**Figure 7-1** shows the CPI of the different SPEC programs when using only an LRU mechanism in the trace cache; each program was tested with different trace cache sizes.

Although when looking at the smaller trace caches the behavior differs from program to program, when using a very big trace cache (256 sets * 4 ways) the CPI is higher (the program will take more clock cycles) in most programs, that is despite the fact that the trace cache was given more volume. There are few exceptions such as gcc and vortex which do get some improvement although not as much as could be expected.

There are two major reasons why few programs behave differently; in the gcc case, the amount of traces being used is very big (much bigger then all the rest of the programs) in such a way that the need for more cache memory overcome the damage made by the bad traces. The huge amount of traces also helps to minimize the damage made by the bad trace; traces are replaced frequently since the working space is much bigger then the trace cache size and that causes many conflicts. Frequent replacement is good because it prevents the bad traces from staying in the trace cache longer.

In the vortex case the improvement comes from a different place, the traces of this program tend to have very clear paths. It means that for most base address there is only one path (or at least one major path) to follow. This situation makes it easy for the branch prediction to warm up quickly which causes the phase of creating bad traces to be shorter.
In many cases the results get worse even when using small caches. Less than half the programs got better results when using 8 sets trace cache rather than 4 (as well as when going from 8 to 16), we will see later on that when using filters the situation changes and bigger trace caches improve performance, we will also show that even the programs that managed to get some improvement from the bigger caches can improve much more using filters.

![Figure 7-1 – CPI results of the different programs as a parameter of trace cache size](image)

There are few ways to solve the problem of the damage inflicted by the regular LRU, one is to use a trace predictor. Such mechanism changes the ground rules; a cache line is no longer referred to as its base address but a combination of a trace base address and its branches directions. When doing so, it might be wise to allow few traces which start at a certain base address to be in the trace cache at the same time. If the machine manages to ask for good traces, i.e. have a good trace predictor, then this solution makes the LRU relevant once more; now when a hit in the trace cache occurs it is indeed a welcome event and keeping this trace longer is advised.

Such a solution is sensitive to the quality of the trace predictor, the predictor will need to predict not only the outcome of the next branch but of the branches that follows, the bigger the traces are (in branches terms) the harder it is to make a good trace predictor, yet longer traces are wanted since wider machines can get better efficiency using them.
Trace predictor was not the subject of this research and in order to remove the trace predictor from the equation the described machine (chapter 5) was chosen; our machine uses no trace predictor and allows one trace per each base address.

We do not try in this thesis to compare the machines or find their advantage and disadvantage, this research simply shows how the suggested techniques can help the given type of machine to gain better performance; when trying to apply the techniques on other machines modifications might need to be made.

Nevertheless in the end of this chapter we will show results of a machines using trace predictor compared to a similar machines using only the filters suggested here. We will show that although the machine is somewhat simpler it can give good enough results to compete with a trace predictor.

Another way to approach the problem of bad traces in the trace cache is by filtering the traces getting into the cache with a usage filter (as described in 7.2), when doing so traces which didn't prove persistency (were not built over and over again) are not saved in the trace cache thus assuming that there aren't too many bad traces in the trace cache. Such a mechanism can be metaphorically referred to as a "doorman" placed outside the cache trying to prevent unproven traces from getting into the trace cache.

As a complementary mechanism after filtering traces getting into the trace cache, there is a filter scanning the traces in the trace cache; this mechanism is meant for those traces who managed to trick the first filter and get in. While the usage filter can only try to prevent bad traces from getting in, the kill or cut & merge techniques are monitoring the outcomes of the traces then deciding whether to reward them or punish.

The kill filter only removes bad traces; it can be thought of as an inside cleaner, taking out of the trace cache traces that made too many mistakes. It is designed to do the same thing as the usage filter does (prevent traces that make many mistakes from being in the trace cache) as such its affectivity decreases once the usage filter works better. It can be seen in the results that when not using the usage filter the kill filter can improve much, however when using a strong usage filter the effect is less noticeable.

The cut & merge mechanism also tries to eliminate bad traces from the trace cache, however it has another advantage, it can help form longer traces. The traces are changed
dynamically being constantly fixed, therefore it can be assumed that eventually they will become good traces and make less mistakes; as such this mechanism can still contribute even when using a strong usage filter.

Figure 7-2 shows CPI results geo-mean of all SPEC programs which were tested (13 programs, as described in the Appendix). Each group indicates a trace cache size measured in number of sets (each cache is 4 way associative), and each couple inside every group indicate a different evicting technique inside the group, from left to right: regular LRU, the kill filter, and cut & merge.

The left column in each couple is referred to the aggressive usage filter, in this case a trace is being written into the trace cache after it is built 20 times; the right column represents the soft usage filter, when a trace can enter the trace cache after only 3 repeats.

A 2 bit counter was used for both the kill mechanism and the cut mechanism, a trace or a branch is being "punished" when the counter reaches zero; the counters are set for 2. As for the usage table, we used a table of twice the size of the trace cache allowing only one trace per base address (just as the trace cache does). More about the machine's parameter is available in the Simulation Environment part – chapter 5.
In general the aggressive usage gives better results; when examining the results of tests where only a usage filter was used the aggressive usage prevails in all cache sizes.

The big gap that appears when going to the huge caches of 256 sets (256KB) emerges from the drop of performance in the soft usage filter. This drop occurs since the usage table size grows with the trace cache size; when the table gets big enough bad traces have more chances to go over the low threshold of the soft usage filter.

Figure 7-3 shows an analysis of the programs performance in the soft usage case in order to see what caused them to drop. The gcc program, having quite a big code and many different traces, continues to improve because it needs a very big trace cache. It appears that gzip and even more ammp are the main cause for the drop in performance due to the big change occurring in these programs when enlarging the trace cache from 64 sets to 256.

![Figure 7-3 - The distribution of CPI results in the soft usage case (the values are relative and can't be compared between programs)](image)

In the aggressive filter, as seen in Figure 7-4, the problem seen in gzip and ammp does not exist anymore and the results continue to improve when the cache size increases. There is however a down side for using an aggressive filter which is not inflected in the performance charts. When preventing more traces from being saved in the trace cache more traces are built, this causes for more instructions to be fetched from the instruction
cache. In many processors fetching more from the instruction cache is not wanted due to power reasons, it could also be that the decode stage is more complex then in our simulator and then the results will look different; we will discus this aspect later on.

This trend of the aggressive filter giving better results is not true when using big trace caches combined with other filters. It can be inflicted from Figure 7-2 that while the aggressive usage filter does a better job in saving only the most important traces, it gives less chance for the inner trace cache techniques to do their work. Therefore when the caches become larger and storage volume isn't the most important issue anymore the soft usage combined with the cut and merge technique or even the kill mechanism gives a bit better results then when combined with the aggressive usage filter.

The reason for the slight improvement in the soft usage filter is different when using the kill or the cut and merge filters. In the cut and merge case the improvement comes from giving the merge mechanism more traces to merge while the aggressive filter takes more time to allow traces into the trace cache. In the kill case the operation of the kill filter is complementary to the one of the usage filter; in most cases the soft usage filter is enough in preventing bad traces from being saved, in such cases waiting longer to save the traces is unnecessary and causes performance to drop. The few bad traces that need an aggressive
usage filter to filter them can be removed later by the kill filter. Therefore when the trace cache is big enough, saving these bad traces in the first place will not remove good traces and they are later removed so they cause little damage.

For the moment, in order to make the diagrams a little smaller, we will drop the soft usage filter. We are more interested in the smaller trace caches plus the drop in performance in the larger trace caches is not too high; we will discuss the soft usage filter some more later.

![Figure 7-5](image)

**Figure 7-5 – the different filters performance distributed by cache sizes (in sets)**

Figure 7-5 shows the results of all eviction types (regular LRU, cut & merge technique and kill technique) with or without a usage filter, as mentioned before regarding the usage filter, only the aggressive usage filter is represented here.

The regular LRU column (second one from the left in each group) represents results when no filter is being used. It can clearly be seen that just using an LRU eviction gives poor results, as already discussed, the performance in this case not only failed to improve when going to larger trace caches, but even tended to get worse. The aggressive usage filter alone manages to reverse this effect and make the results improve when the trace cache size increases (first column from the left).
The cut and merge technique or the kill technique can do the same without the need for the usage filter; these techniques by reordering the trace cache dynamically manage to eliminate the bad traces from the cache.

Table 7-1 shows how the different evicting techniques alone stand in relative to just using the aggressive usage filter, Figure 7-6 shows the comparison visually. While both, the usage filter and the other techniques are trying to prevent bad traces from being in the trace cache; the main difference is that the usage filter prevent them from being written to the trace cache while the inner trace cache techniques let them in, follow them and then if needed remove them from the cache.

When the trace caches are relatively small a bad trace can do damage not only by being fetched and executed but also by merely getting into the cache and causing other good traces to be removed. Same thing can also happen with traces which are not necessarily bad but seldom repeats; the usage filter will simply not let all those traces into the trace cache and by doing so will prevent eviction by conflict for some good traces.

<table>
<thead>
<tr>
<th>Relative to agg usage</th>
<th># sets in TS</th>
<th>filter</th>
</tr>
</thead>
<tbody>
<tr>
<td>102.30%</td>
<td>4</td>
<td>C &amp; m</td>
</tr>
<tr>
<td>102.52%</td>
<td>4</td>
<td>Kill</td>
</tr>
<tr>
<td>98.57%</td>
<td>8</td>
<td>C &amp; m</td>
</tr>
<tr>
<td>101.72%</td>
<td>8</td>
<td>Kill</td>
</tr>
<tr>
<td>96.53%</td>
<td>16</td>
<td>C &amp; m</td>
</tr>
<tr>
<td>100.55%</td>
<td>16</td>
<td>Kill</td>
</tr>
<tr>
<td>92.66%</td>
<td>32</td>
<td>C &amp; m</td>
</tr>
<tr>
<td>97.75%</td>
<td>32</td>
<td>Kill</td>
</tr>
<tr>
<td>90.95%</td>
<td>64</td>
<td>C &amp; m</td>
</tr>
<tr>
<td>96.48%</td>
<td>64</td>
<td>Kill</td>
</tr>
<tr>
<td>90.95%</td>
<td>256</td>
<td>C &amp; m</td>
</tr>
<tr>
<td>96.60%</td>
<td>256</td>
<td>Kill</td>
</tr>
</tbody>
</table>

Table 7-1 – the performance of the inner trace cache technique presented relatively to the aggressive usage filter results used on the same cache size

When approaching bigger trace caches the inner cache techniques beat the usage filter performance. The picture changes because as the caches become bigger the conflict cases in them happen less often, thus making the effect of good traces being evicted due to conflicts neglect. Since there is now enough memory space in the trace cache for many traces, it becomes a disadvantage being "pedant" as the aggressive usage filter does by not letting seldom used traces into the trace cache. Another negative effect that may occur
when using a usage filter in large traces is that the usage tables are getting quite big as well, that causes for bad traces to get more chances of being repeated and then infiltrate into the trace cache

Figure 7-6 – comparing the different filtering techniques as they stand by themselves, distributed by cache sizes (in sets)

Figure 7-7 shows how much can be gained by using the inner cache techniques on top of the aggressive usage filter.

As expected, the kill filter can not gain much improvement on top of the strong usage filter since it tries to remove bad traces from the trace cache and the usage filter already is doing quite a good job in not letting them in. The small improvement we do get of about 1%-2% comes from those bad traces which somehow managed to enter into the trace cache, this phenomenon tends to happen as the usage table becomes bigger (which is correlated to the trace cache size), when having the kill mechanism it is possible to correct those mistakes.

However the cut and merge technique is a different story, although trying to eliminate bad traces as well, this mechanism tries to form a better and longer alternative trace. Since the machine being used is quite wide there is much to be gain from longer traces, so even if the usage filter manages to prevent bad traces from the cache, it still doesn't provide the machine with better traces.
Figure 7-7 – comparing the inner trace cache techniques when used with a usage filter, distributed by cache sizes (in sets)

Table 7-2 shows the results of the cut & merge technique relatively to the regular LRU on the same machine with the same usage filter, aside from the small trace cache of 16 traces (4 ways * 4 sets) which also shows a 4% improvement, there is an improvement of 6%-8% on top of the already improved machine in the rest of the cases.
ones in the left to a regular LRU. Each three goes from right to left by how aggressive was the usage filter used in those tests (non at all, threshold of 3, threshold of 20). Each column is presented relatively to the test of the cut and merge technique with the aggressive usage filter and a trace cache of 16 traces (4 sets * 4 ways); the columns are a geometrical mean of all the 13 benchmarks results.

![Instruction fetched from IC](image)

**7-8 – comparing the number of instructions fetched from the Instruction cache in all filters combinations distributed by cache size (in sets)**

Figure 7-9 shows the relative CPI of the same tests, it brings all the tests together in the same order as 7-8 in order to show the tradeoffs needed to be done by performances and fetching less from the instruction cache.

Fetching less from the instruction cache was not such an issue for our machine since it is a RISC machine and also power consumption was put aside. So far performance was the only issue; however it is important to show that the aggressive usage filter, although showing good performance in the small trace caches, has its price when considering the utilization of the trace cache. If it is desirable that more instructions will be fetched out of the trace cache, then this should be taken into consideration; such cases could be when the decoding stage or the building process cost the machine extra in time or in power (for example a CICS machine, when the trace cache has the decoded instructions), or when the trace cache being used spends less power then the instruction cache.
Bad or not, it can be clearly inflicted that as for fetching from the instruction cache (7-8) the aggressive usage filter cause for about 2 times more instructions to be fetched then without a usage filter in most cases. The soft usage filter appears to be a good compromise between performances and fetching less from the instruction cache. We can see that this filter gives similar numbers (of instructions fetched from the instruction cache) to not using a usage filter and yet in most cases manages to give almost as good a performance as the aggressive filter.

Interestingly, in the small trace caches (16 traces or 32 traces) the soft usage filter gives even better utilization of the trace cache than not using a usage filter. This happens because in those trace caches the problem of blocking traces for the first few times (thus having to build them over and over) is neglect to the problem of traces conflicting (causing much replacement in the trace cache), when the caches get bigger traces conflict less and then the usage filter blocking becomes more important.

Another interesting fact is that although the CPI results of the trace cache without filters at all (regular LRU trace cache) get worse as the trace cache becomes bigger, the number of instruction being fetched from the instruction cache decreases. The reason for that is that many bad traces get into those trace caches because the lack of a usage filter; those bad
traces are not removed since there are no inner cache filters, thus causing the machine to have many branch miss-predictions. However, when the machine recovers it builds a new trace from where the bad trace had a miss-prediction and therefore the next times it will make a mistake at the same branch, the correcting trace will already be available in the trace cache and since the trace cache is big enough it might take a while before they both are removed.

Figure 7-10- actual part of the instruction code fetched out of the instruction cache, distributed by cache size (in sets)

Figure 7-10 shows the percentage of the committed instructions which came from the instruction cache. It is mainly meant to show how big a factor these instructions are when considering the entire test. It can be seen from the diagram that when the trace cache is big enough the amount of instructions fetched out of the instruction cache is neglect. However in the smaller trace caches it is a significant amount of instructions, therefore when fetching from the instruction cache is an issue for the machine it should be considered once choosing filtering techniques.

If we look for example on the 16 traces cache (4 sets), more than a quarter of the instructions came from the instruction cache in all tests using aggressive usage filter. In the same cache size when using the cut and merge technique with soft usage filter causes fetching about a third less instructions at the cost of only one percent in
performance, when the number of instructions is that high such a change could make all the difference.

Another statistic worth checking is the amount of dynamic inlines made by the cut and merge mechanism. Whenever a complete function is merged into a trace the call and return instructions become unnecessary and can be spared, such a condition could make the machine run less instructions plus save time by not doing the function call routine. We consider this situation as a dynamic inlining.

Due to the way our simulator works, this kind of process could not have been simulated and therefore the complete function was entered to the trace including the call and return instructions plus all the actions that are driven from that.

We bring Table 7-3 to show the amount of such inlines made in the different tests in order to indicate the improvement in performance using this technique could add to the results. This situation is possible only when using the merge technique, therefore the results in the table shows geometrical mean of all tests which used the cut and merge technique for each program.

<table>
<thead>
<tr>
<th>prog</th>
<th>commited</th>
<th>Fetched</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ammp</td>
<td>2593745</td>
<td>28817506</td>
</tr>
<tr>
<td>Art</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Bzip</td>
<td>2282231</td>
<td>6510114</td>
</tr>
<tr>
<td>equake</td>
<td>518842</td>
<td>1092933</td>
</tr>
<tr>
<td>Gcc</td>
<td>740977</td>
<td>2979599</td>
</tr>
<tr>
<td>Gzip</td>
<td>3144015</td>
<td>15106479</td>
</tr>
<tr>
<td>mcf</td>
<td>7827</td>
<td>147995</td>
</tr>
<tr>
<td>mesa</td>
<td>1669820</td>
<td>4228387</td>
</tr>
<tr>
<td>parser</td>
<td>1591524</td>
<td>7112855</td>
</tr>
<tr>
<td>perl</td>
<td>245957</td>
<td>751085</td>
</tr>
<tr>
<td>twolf</td>
<td>46332</td>
<td>1604406</td>
</tr>
<tr>
<td>vortex</td>
<td>1341402</td>
<td>2031236</td>
</tr>
<tr>
<td>Vpr</td>
<td>2909037</td>
<td>7526794</td>
</tr>
</tbody>
</table>

The numbers in the table represent number of times a full function was fetched from the trace cache after being merged into a trace, therefore the numbers stand for pure profit which could be gain. These function calls which could be spared are taken from tests of
500,000,000 instructions, we expect a sum of a few million function calls spared to be seen in the performance.

It must be said though that eliminating function calls will require a recover mechanism for those traces. If a branch miss-prediction occurs in such traces inside a function call correcting it might not be possible.

Such a mechanism could call for flushing the entire trace or at least the part of it referring to the function which was called, another implementation could be a mechanism that saves the function's return address attached to the trace just in case.

7.4. **Comparing to trace predictor**

The trace predictor presented in chapter 4 gave a far better performance then the trace cache used in this work; however, as already discussed earlier, when using the trace cache without any filters its results are indeed poor.

We present the same diagram presented in 4.4 but instead of using just LRU in the trace cache, we use an aggressive usage filter and the cut and merge technique. Figure 7-11 shows how the dynamic filters managed to improve the performance of the trace cache we used for this work and make it a worthy opponent.

![Figure 7-11 – Compared results of machines: with no TC, with TC and dynamic filters, with TC and Tpred. The results are distributed by trace cache size (in sets)](image)

53
The regular instruction cache in the machine without a trace cache is twice the size of the one used in the other machines (256 K); it is the same machine for all experiments (trace cache size is not an issue there).

While in all tests not using trace cache gives by far less performance, using the filters is always as good as using a trace predictor if not better. The abnormality seen in 4.4 when using a trace cache of 256K disappeared and using filters in the trace cache improves performance in more then 5% in relate to the trace predictor.
8. Hints

8.1. Introduction

The current process of translating high level language into machine code involves several different mechanisms; the code is first being translated into intermediate code which is machine independent. The intermediate code is being translated into machine language by several phases that optimize it in respect to the specific characterizations of the machine. Some compilers in addition, use runtime information of previous runs to improve the execution time of the code for future runs, this technique is called feedback directed optimizations (FDO) [15][26][27] and is discussed in the Profiling part - chapter 6.

The compiler, by analyzing the code structure and profile data, gets a lot of information about the program's behavior; however, there are very few mechanisms that allow the compiler to pass information to the hardware, therefore much of the compiler's knowledge is either lost or does not reach its full potential for making the program run faster.

A simple example for such ideas is implemented as part of IBM’s Power-PC processors, where a special bit, in the instruction decode, was added for branch instructions indicating the compiler’s/programmer’s suggestion of what should be the default value of the prediction for that instruction [28].

Another example for such work can be found in [16] where FDO information was used in order to add hinting bits for the instructions directed for the value prediction mechanism. Those bits were used to hint the mechanism whether a specific value has a good chance of being predicted correctly as well as what kind of prediction technique is advised.

During the last few years many researcher looked at how to enhance the existing instruction set in order to use the knowledge which the compiler has [29]. The problem with the proposed extensions for “transferring information” between the compiler and the hardware is that due to compatibility issues, it has been tried to avoid changing the ISA (instruction set architecture) which the processor executes. Thus this research suggests investigating a new technique to solve the problem; we suggest using “Pragma” instructions; i.e., instructions that the machine does not need to execute, but can be used for transferring “knowledge” between the hardware and the software.
Similar ideas can be found in [30] dealing with VLIW machines; in order to save power this work suggests for the compiler to add a hint instruction telling the machine when a branch instruction will be executed thus using the branch prediction mechanism only for branch instructions.

Another work [31] uses the compiler knowledge of a program paths in order to hint real time embedded systems of how much code remains to be executed at the worst case, in such a way if the system has a deadline for executing the code (multimedia for example) it can calculate whether lowering the clock frequency in order to save power is possible.

The usage of Pragma is already spread in high-level languages as well; for example, in C language it is possible to suggest the compiler to put a certain variable in a register, or to put some data in a specific segment. We believe that the use of such instructions to allow the compiler to inform the hardware how it should behave can be of benefit for modern computer architectures and may lead to a new technology for both compilers and hardware.

Our work target is to use the compiler knowledge combined with FDO in order to create such pragma for the processor; in order to demonstrate how such pragma instructions can be used, here are three examples:

1. The compiler can indicate to the branch predictor:
   - Should the default value be Taken / Not Taken
   - Does this instruction designate a loop and if so maybe how long is the loop
   - Should a branch prediction not be used for that instruction or should a specific technique be chosen for the task

2. The compiler can help the use of data prefetching mechanisms.

3. Remarks by the compiler can help a Trace-Cache (chapter 4) build better traces:
   - How many copies of the basic block should be used in a loop unrolling
   - What part of the code can benefit from the use of a trace cache
   - Should a trace allow another entrance into the middle of the trace (another address)

In our research we used the simple scalar simulator (chapter 5) running a DLX instructions set, in order to test our speculations, the simulator's description and specification is discussed in the Simulation Environment part (chapter 5).
Using hints in general aside from not improving performance when added to the compiled code can also make it worse; adding hints means adding instructions and that can make the code take more memory space as well as make the machine work on more instructions. Such an act can cause more harm than good. In our implementation using hints; hints were used not as part of the trace itself but accessed only at the building process, doing so allowed the machine to enjoy the benefits of hints while accessing them very few times.

In this research we will focus on the trace cache aspect; most researches deal with sorting traces aimed for dynamic filters [4][5][8], filtering techniques were also examined in this work as discussed and analyzed in the Dynamic Filter part – chapter 7. We present here a different approach for sorting traces - having the compiler mark traces in advance according to a profiling image, then a special mechanism will route the traces either to a hot trace cache, a regular one or none at all.

We will also try combining the filtering techniques with the hints system and compare the different techniques. Our filters though do not move traces from one cache to another as done in [4][8], but only try to manage the traces "traffic" in and out of a trace cache better, i.e. when combining the hints with profiling, only the hints mechanism can decide which trace goes to which trace cache.

Since we are using a RISC architecture with fixed size instructions, the fetch/decode stages are not an issue when referring to machine's width. The only event that can prevent those pipe stages from reaching their full capacity (aside from structural hazards) is switching from one cache line to another in the regular instruction cache. In these situations the trace cache has a big advantage over the regular instruction cache since it merges all the basic blocks together to one cache line and therefore switches cache lines less frequently. As a result we would expect a bigger performance penalty in case of trace cache misses the wider the machine will be; we chose to use an 8 wide machine which is wide enough to make it count but still makes it reasonable.

The process of choosing hints involves using profiling information. The intention is that the compiler, after providing a machine code, will run the code and collect profiling information regarding the trace cache performance. More of that subject can be found in the Profiling part (chapter 6) and later on in this chapter under the section of Choosing hints (8.4).
We do not, however, use in this research the compiler's "knowledge" of the code, as used in [15][17][18]. The compiler builds inner structure to support all of its optimizations; this information could help to create hints. Such information could be: how many times a loop is being made (when using "for" loops), from how many places is a function being called, how long are the functions, and so on.

Getting into the compiler (gcc) inner structures and procedures is not an easy task and was beyond this research scope; however it could be a very interesting issue for further research.

### 8.2. Implementation of hints

The simple scalar instruction set is an improved MIPS/DLX instruction set [11][12][13]. The instructions are 64 bits long and a general instruction looks as followed:

```
<table>
<thead>
<tr>
<th>16-annotate</th>
<th>16-opcode</th>
<th>8-ru</th>
<th>8-rt</th>
<th>8-rs</th>
<th>8-rd</th>
</tr>
</thead>
</table>
```

Since we are using the compress mode we refer to the instructions as 32 bits long, in this case the instructions should look as followed:

```
<table>
<thead>
<tr>
<th>6-op</th>
<th>5-rt</th>
<th>5-rs</th>
<th>5-rd</th>
<th>5-shift</th>
<th>6-func</th>
</tr>
</thead>
</table>
```

In order to minimize the use of op-codes we decided to use for our hints one op-code, the op-code should state that this is a hint and the appropriate mechanism will deal with it.

For our hints and as a general structure we suggest the following fields:

```
<table>
<thead>
<tr>
<th>hit-op</th>
<th>hint-type</th>
<th>offset</th>
<th>br-dir</th>
<th># of br</th>
<th># of inst</th>
</tr>
</thead>
</table>
```

When:

- **hint-op** is the op-code for a hint instruction (6 bits)
- **hint-type** refers to the kind of hint it is, in this work only one kind of hint was tested but in general 5 bits field should be more than enough (32 types).
- **offset** indicates which instruction does the hint refer to; the default should be the next instruction (PC + 4) and then this field is 0 and for this work it was enough for all tests. However this option should be allowed for hints that need longer processing time or if there is a need to accumulate hints (for example – primer and secondary trace paths). We suggest 5 bits for this field.
While the previous fields were for all hints type the next are directed to a build hint:

- **br-dir** is a vector of the trace directions, this field refers to a hint that offers a specific path of branches to follow, it intends to bypass the branch predictor during the building process. Each bit refers to a branch instruction (in order of appearance); since all tested traces had at most 6 branches this field is 6 bits long.
- **# of br** is the number of branches in the suggested trace, to go with the 6 branches limit this field needs 3 bits.
- **# of ints** is the number of instructions in the suggested trace; all used traces had up to 64 instructions therefore 6 bits were used.

We expect the hints to be added to the code by the compiler or even a profiler tool using a profile image of the program after being run with training inputs.

Figure 8-1 describes the process of creating the hints; when only a binary code is available the process can still start at phase 2 and use only the profiling image; however doing so may harm the hints quality.

The quality of the hints will depend on the profiling image detail level and the compiler / profiler ability to analyze the given information.

---

**Figure 8-1 - scheme of the hints making process**
8.3. Type of Hints

As we approach the idea of making hints to support a "hot trace cache" which will contain good and hot traces, a set of hints should first be created. In this work we chose to use hints that give the trace cache an opinion on upcoming traces. These hints are supposed to advise the machine, based on previous profiling, of whether a trace has a good potential or not, the machine should have its own mechanism to overcome bad advices.

We propose two kinds of hints: "good trace" and "bad trace", the "good trace" advises the machine to save this trace in the hot trace cache while the "bad trace" advises it not to save the trace at all.

As for the good trace hint, we suggest two kinds of such hints. The first kind is dominant enough in such a way that when the machine gets to the point in the code the good trace starts (its base address) there is no doubt this trace is the right way to go. This is not always the case, many times there is an often used base address that has two major paths which follow it, sometimes even more; we will discuss this kind shortly.

In the dominant kind, when the profiling results shows that a trace is dominant enough for its base address and is used enough to be considered important, it might be a good idea for the hint to guide the machine of how to build this trace as well as advise it to put the trace in the hot trace cache.

When doing so, the hint mechanism bypasses the branch predictor and causes this trace to be the only trace being built or used from that specific base address. Such a statement is quite strong and should be used only when the trace is by far better then the other options, it could be wise to allow the trace cache to disengage from such a hint when it misleads the machine too often (in the rare cases where the profiling fails).

This hint was marked by us as GOOD_TRC_B and was the primer hint in our tests.

When a trace is from the second kind, hence its path is not a dominant one; it still might be a good enough trace to be kept in the hot trace cache. The question remains how it can be prevented from blocking other traces with the same base address. Such cases happen due to not using a trace predictor; applying the hints systems to a machine with a trace predictor changes the ground rules.
One solution is just to point this trace as a good trace but let the branch predictor decide whether to build it or not. In this case if the trace was built then it will be saved in the hot trace cache, if not the machine will work as usual and save it in the regular trace cache. A trace is allowed to be only in one cache at a certain time.

This hint was marked GOOD_TRC and should work good when the traces are used in phases. In such cases there are certain times during a program in which the good trace is dominant, during that time the trace will be available in the hot trace cache. When the program reaches its other phases where the trace is not dominant anymore then the branch predictor will have its chance to build a more suitable trace.

When using this hint the hot trace cache must have the option to remove traces when they tend to make mistakes, otherwise the trace might stay in the hot trace cache and block any attempt to build another trace from that same base address (when using no trace predictor if a trace's base address is available the trace will be fetched).

If the trace does not behave in phases then the last solution will not work, the machine will need some kind of trace predictor to help it choose when to use which trace.

Since the trace caches allow only one trace for a base address, the original machine (not using hints) should have the same problem with those traces.

Using a strong trace predictor is not an option because in such a case it would be best to change the trace cache type as well, however for this research we only wish to improve the given machine.

The use of some predictor mechanism only for a few traces is a better option. The profiler can find which traces might be tricky, and by allowing traces with the same base address to be in both trace caches (each cache will hold one trace); a poor prediction mechanism will be able to choose between them. The idea is to make some kind of trace predictor that will only specialize in a few base addresses. In these cases if the profiler shows two dominant traces for one base address it might be best to hint the machine of both of them, let each go to a different trace cache, and then let the predictor choose between them.

Unfortunately using our profiling tools combined with a simple 2 bit counter as a chooser between the two trace caches did not provide good enough results and the idea was put aside for future work. We will show in this work some good results for the GOOD_TRC_B hint as a teaser for this concept potential and leave the big profiling work to be investigated more.
The other type of hints was bad traces; as opposed to the "good trace" hint, those hints do not have the advantage of being dropped while building the trace. While the hint disappears in the building process and then is not used when the trace is fetched from the trace cache, these hints intend to prevent traces from being saved in the trace cache. Two kinds of traces can go into this category: traces that have very poor usage and traces that tend to make many mistakes.

The first kind refers to traces that are not used much after being build or even at most times not used at all. Such traces use up valuable space of the trace cache data area and prevent it from better traces. This kind of hint simulates the work of the dynamic usage filter and according to our tests it appears to have no advantage of this filter. We believe that it should only be used when the dynamic filter is not an option.

The second kind refers to traces that tend to behave unexpectedly; when traces manage to mistake very often they probably are too hard for the regular prediction to handle. In those cases we propose to put hints that will prevent those traces from being saved, thus saving storage in the trace cache and minimizing the penalties caused by the mistakes. This kind of hint could be used by the branch prediction mechanism as well, however this it not part of our research.

8.4. Choosing hints

The hint which we chose to examine is the GOOD_TRC_B hint; we will now discuss how to choose which traces to hint of.

When trying to choose good traces, a good way to do so is to analyze the profiling image of the machine when it simulates a perfect branch prediction mode. For this mission an oracle mode of the Simplescalar simulator was used and the trace cache information was collected into our profiling tool.

Doing so does not necessarily provide optimal traces; the reason for that is while using the perfect branch prediction the machine still uses its regular trace builder mechanism. This mechanism tries to fill each trace up to its capacity (maximum number of instructions or branches), stopping only on indirect jumps or at call/return/trap instructions.

Sometimes it may prove to be better building part of a trace or inlining a function call, a good example can be loop unrolling. When a loop is being rolled into a trace 3 times and it has another iteration of the loop at the end, it will build a new trace just for running that single iteration replacing the old trace. When running the same loop again and again (for
example a short loop of 4 iterations inside a long loop) it will replace the last trace that was built since it expects a trace of 3 times the loop body.

Another example could be a base address that starts always with the same branch directions for the first few branches, however in the last branches behaves differently each time; it might be wise to keep a somewhat shorter trace which will serve all cases rather then a longer one that could cause many branch miss-predictions.

Nevertheless the perfect branch predictor is still a good indicator, and the profile image made from the Oracle simulator was used for our analysis. In order to find those special cases we tried to cross information with regular machines using branch predictor and the different filters proposed in chapter 7. All the profiling images were analyzed alone and then checked whether there are different hot traces in each of them.

A good trace would appear to be a trace that is used a lot, has low usage, has a significant number of branches and instructions, has at least a few taken branches and has low competition.

- **Used a lot** – It is of course desired to have in the hot trace cache traces that will be used a lot, otherwise they are just taking up valuable space and prevent other traces from using it without a significant benefit.
- **Low usage** – Low usage refers to the trace being saved and removed a lot from the trace cache and while being saved in the trace cache it is not reused much.
  If a trace is used a lot but has a high usage then it probably means that the trace is used in bursts and each time is often used. In these cases the machine can handle it pretty good even with a small trace cache therefore putting it in a special trace cache will not improve performance all that much.
- **Branches and instructions** – Since the chosen trace would occupy space in the special trace cache; longer traces are more welcome in order to use most of the available cache size. A longer trace can also use better parallelism in a wide machine.
  Aside from having many instructions in the trace it may prove to be better if it has many branch instructions which are taken.
  When using a regular instruction cache, taken branches usually stop the fetching sequence causing the machine to fetch less than the maximal number of
instructions per cycle. Further more, if the branch jumps to a new cache line there may be some extra penalty for switching blocks.

In such cases having the trace in the trace cache can be more beneficial then building it from the instruction cache even if all the instructions are in the instruction cache to begin with; therefore there is more potential for improving performance when the traces contain taken branches.

- **Low competition** – Competition here stands for two traces that start at the same base address but use different paths. When using a trace cache that can hold only one trace that starts at a certain base address, traces that have unexpected behavior and tend to use different branch resolutions each time cause many miss-predictions in the trace cache.

Most probably there will be more miss-predictions fetching such traces from the trace cache than a regular branch prediction would have made, that is due to the fact that the branch prediction can learn more about each branch and predict it separately according to the machine status, while in the trace cache on the other hand once a trace is being fetched it is taken as a whole with all its branches directions and it is not very flexible.

Competition is hard for the trace cache with or without hints, and in some cases it might be better if the trace wouldn't be in the cache at all and will be built from scratch (such might be an indication for a BAD_TRC hint).

Therefore having such a trace in the special trace cache could be futile and a waist of cache space at best. It could even be worse especially when using it as a GOOD_TRC_B hint, when doing so the trace will block other traces from being built and if that trace is not much better than the other options performance might drop.

Following the guidelines presented above we used an excel file format to create the hints (see 6.5). Only the hint of building traces (GOOD_TRC_B) was tested since our profiling analysis couldn't give an accurate enough data to form the other hints; other kind of hints were tested but we couldn't reach conclusive results. Since dealing with profiling issues was beyond this thesis scope, it was decided to leave it as future work.

The building hint is a strait forward hint, it tells the machine how to build the trace and block other traces from being formed, if the trace is built a lot and the chances for another trace with the same base address to be needed are slim then this hint could prove to be very useful.
After the hints were inserted to the code the simulator operates as followed: the code is being fetched from either one of the trace caches or the instruction cache, if it comes from the instruction cache then a trace is built.

While building a trace if the machine comes across a hint instruction which requests to build a trace the builder will finish the trace it was building (if it exists). Then it will start building the new trace while bypassing the branch prediction mechanism with the hinted branches vector. When the trace is finished the builder will send it to the special trace cache rather then the regular one, the hint instruction, off course, will not be a part of the trace.

When a fetch is being attempted from the trace caches, a mechanism that deals with a hit in both caches was implemented; each trace has a counter indicating its successes. The trace with the better counter will be chosen, in case of a tie the special cache wins. There were a few more ideas to manage a hit in both caches, such as letting the regular trace cache win only when the hinted trace counter reaches zero, or even using some kind of a poor prediction mechanism between the two. The subject may be interesting for further research; however since we only used build hints in this work the situation of having traces with the same base address in both caches isn't possible.

8.5. Results

Since the process of choosing hints and testing them is long and complicated, we have decided to focus our efforts on only part of the SPEC programs. The programs that were chosen are such which received a high percentage of improvement when using larger trace caches.

The logic behind such a choice was that the process of putting good traces in a special cache and not saving traces with low usage imitate in some way a bigger trace cache.

Figure 8-2 shows performance results of early tests which have been made, all results refer to a system which uses the kill filtering technique without a usage filter (chapter 7). Each program is shown here with the different cache sizes, all of them are 4 ways set associative.
The programs that were selected for being hinted are equake, vortex, gcc, mesa and perl. The selected programs were observed in a perfect branch prediction mode and different heuristics for choosing hints were tried based on the profile images of the tests. After composing few sets of hints they were all tested again on a machine with a 2 levels branch predictor; the trace cache profile image received after the new tests was examined once more as well as the performance results. Techniques which brought poor improvement for the performance were dropped and the better ones were tuned and corrected according to the new profile images.

In such experimental way came the decision to use only the building hint (GOOD_TRC_B) and drop the other types for now; a price function was composed as well in order to have an automatic tool for choosing hints.

We will now present and discuss the results of these 5 SPEC programs; the tests which used hints had machines with two trace caches, one for hinted traces in size of 16 traces (4 sets * 4 ways) and the other for the rest of the traces in the same size. Any time a filter is being used, it is used only for the regular trace cache. All diagrams which compare tests of hints to ones without hints compare the mentioned machine to a machine with one trace cache of twice the size (8 sets * 4 ways).
Bigger trace caches were not tested since the working set of most programs didn't justify doing so, when working with real programs that use a bigger working set increasing the caches should be examined.

Figure 8-3 – improvement achieved by adding hints to the code as it appears in the different tests, distributed by programs

Figure 8-3 shows the improvement of each test (machine combination) in comparison to the same test made without using hints. As said before, the performance of the two trace caches with 16 traces each are compared to a one trace cache of 32 traces. This diagram shows percentage of improvement of the hinted tests' CPI in comparison to the regular ones, this means that lower numbers are better (the improvement was greater) and the number between different columns inflict nothing about the difference in performance between the two tests. The only thing which can be concluded from comparing two bars is that in one test the hints managed to help more than in the other.

It shouldn't be too surprising that presented chart shows somewhat of an opposite picture to the one received in the filters diagrams. It is driven from the simple notion that the better the results with the filter where the harder it was to improve them more.

Nevertheless in most cases the hints did manage if not to improve much then at least not to ruin the performance of the machine. It can be seen that in the Equake, Vortex, and Perl
tests the hints managed to get a significant improvement, while in Gcc and Mesa the improvement was far less impressive, if one existed at all.

The reason for that is the programs behavior; while Vortex, Equake and Perl have very clear good traces which do not tend to change their branch instructions directions during the program. Gcc on the other hand has many traces and they tend to behave unexpectedly; in the Mesa case, the problem is a bit different, the program does not have too many dominant traces and even then most of them already have quite a good usage in a trace cache of 32 traces.

It would appear that the kind of hint we were testing (GOOD_TRC_B) was not the best one to use in both the Gcc and the Mesa programs; perhaps using different types of hints there (like hinting of few paths for one base address in the gcc example) could help improve their performance better.

Despite all that, even with our regular building hints, performance in general managed to level with the machines using no hints in those problematic programs. These results strengthen the hints notion since adding hints to the programs while being very helpful in some cases was not influencing badly in others.

Figure 8-4, Figure 8-5 and Figure 8-6 show relative performance of the tests using or not using usage tables and with or without hints. Each chart presents a different eviction technique and compares all tests made with that technique; the data between charts can not be compared.

These charts complete Figure 8-3 in order to give a better point of view for the tests results as they are, the charts give better perspective of how the performances really look and do some correction to the optic illusion made when only looking at the improvement diagram.
Figure 8-4 – performance of the cut & merge tests with and without hints, distributed by programs (the results are relative and can't be compared between programs)

Figure 8-5 – performance of the kill tests with and without hints, distributed by programs (the results are relative and can't be compared between programs)
It was interesting to see how the hints stand as a technique for itself, in comparison to the other filtering techniques which were suggested in chapter 7.

Figure 8-7 shows the relative CPI of the five programs when using each technique alone, once more the comparison is between caches of 32 traces to two caches of 16 traces.

Although the regular trace cache in the tests using hints was a trace cache with no filters just a plain LRU eviction policy, and as such is considered to give bad performances (see the Dynamic Filters part – chapter 7), interestingly enough the hints tests managed to give better results than all the other techniques in the vortex, perl and equake examples, while not being too far behind in the mesa and gcc tests.

Apparently, having the good traces in the special trace cache was powerful enough to overcome the bad performance of the regular trace cache in those machines.
After examining the techniques alone, Figure 8-8 shows how the techniques work combined, we show the results of using hints with any other technique in comparison to paring the other filters.

As can be seen in most cases the hints can give better results; the main competitor is the cut and merge technique combined with the aggressive usage filter, the same combination which gave the best results in the Dynamic Filters chapter.

Another interesting combination is using hints with the kill technique; such combination gives pretty good results while using no memory and a very simple eviction mechanism.

Quite surprisingly, the kill technique gave better results in most cases combining with the hints then the cut and merge although when compared in chapter 7 the cut and merge filter produced superior performance. This outcome might have occurred since good traces were held in the special trace causing the merging mechanism to have a harder time at finding good combinations of traces to merge, as well as the difficulty in keeping them in such a small trace cache (the regular one).

When we added a usage filter (all techniques together), the cut and merge technique came first once more, the combination of soft usage filter with cut and merge technique and hints proved to be the best combination. It could be that since the really good traces went...
to the special trace cache, only a softer usage filter was needed in order to work with the semi good traces. The usage filter, although being soft, probably helped the regular trace cache with the problem of keeping the merged traces for longer periods of time as well.

Figure 8-8 – comparing the combinations of all suggested techniques, distributed by programs

So far the hints managed to give very good results; the down side is that the programs ran on the same input when both collecting the hints as well as testing them.

In order to test the concept of learning hints from different outputs, the hints made by the given training inputs were tested with different inputs.

It would appear that when the tested program has a repeated routine it keeps running no matter what the input is, such as the vortex example, the hints work perfectly.

Figure 8-9 shows the vortex results with and without the hints; the marked input represent the training input, the improvement remains the same regardless what the input is; the vortex program managed to benefit in each test when using the hints.

However this is not always the case; when testing the gcc program or perl results were not as good as could be hoped.
After examining the profiles of those tests it was revealed that these programs behave differently with the new inputs. The new inputs made the program run in different phases or spend much more time in other places of the code rather than the training input did. Traces which were considered good and hot in the training input became in some cases colder and in other cases even not quite good anymore (the program used different traces for the same base address).

Since Mesa and Gcc programs did not gain much improvement in performance from the hints to begin with, it was more interesting to experiment with Perl, Vortex and Equake. Unfortunately different inputs for Equake were not available; therefore we can only show the results for Vortex and Perl. The perl program achieved better performance when using the hints in the training input, however Figure 8-10 shows how the improvement seen in the marked test (training input) disappear when changing inputs.
To try and understand what caused this drop in performance, the profile image of all tests was examined; the results can best be seen in Table 8-1 and Table 8-2. The tables show the best traces used in each test as the automatic hints choosing mechanism would have selected them if it were the training input.

The first columns in blue represent the result of the training input using the perfect branch prediction mode. The other three represent the profile results of the inputs stated above each column.

"Commit" represents the number of times the entire trace committed, "appear" stands for the number of times the trace was written into the trace cache and "price" is the value issued to that trace by our price function.

The most interesting part is the line written in the bottom of each column (marked in red); this line shows the most valuable trace in each test. Surprisingly enough, all three traces were not included among the chosen hints, below each trace states the reason why the trace was not selected.

The first trace was seldom used in the original input; the second one had a very good usage which means it was used locally in the training test while behaving differently in the other test. The last trace presented is the most radical one, this trace wasn't even accessed in the original input; the program simply didn't reach that part of the code.
<table>
<thead>
<tr>
<th>TC</th>
<th>Ins #</th>
<th>File</th>
<th>Proc</th>
<th>Artar</th>
<th>Commit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6-1: Profiling results of perl, distributed by inputs (part A)
Table 8.2 - profiling results of perl, distributed by inputs (part B)
9. Conclusions and Future Research

This paper approached the issue of small trace caches and their difficulties in preserving good traces for long periods of time.

We offered to use dynamic filters in order to sort the traces inside the trace cache and to use hint instructions in order to help the machine sort traces and save them in a hot trace cache.

In the filtering part we showed that when not using a trace predictor it can be pretty harmful not to use filters at all. Few filtering solutions were attempted: usage filter, kill filter, cut and merge filter.

The usage filter managed to fix the problem by saving in the trace cache only traces which kept being built, however special memory was needed for such task. The kill filter managed to give reasonable results without the need for extra memory by removing bad traces from the trace cache assisted by feedbacks from other pipe stages. The cut and merge technique achieved even better results without the need for extra memory as well but with a more complex logic of cutting traces at their bad branches and merging sequential traces. The cut and merge technique had an extra attribute, besides removing bad traces from the trace cache it dynamically formed better and longer traces. Combining the usage filter with the cut and merge technique proved to be the best solution in achieving performance improvement.

This work also showed that if the usage filter and the cut and merge mechanism are used by a machine without a trace predictor, it can outperform similar machine which uses a trace prediction.

For future research in filtering we suggest testing ways to adjust the usage table size in relate to the trace cache size and/or the application being run. We also suggest testing ways to dynamically tune the counters which are used in the usage filter as well as the kill and cut mechanisms.

In the hints part we showed that hinting the machine of good traces in order to save them in a special trace cache can improve the results on top of using filters. We also showed that although the hints couldn't improve all tested programs they were seldom a drawback.

For future work we suggest to improve the learning process by using more powerful profilers and make use of the compiler resources. Checking the impact of hints on a
machine which uses a trace predictor and adjusting the machine to work with them could be an interesting topic as well. We also suggest expanding the hints concept and check additional hints not necessarily trace cache oriented.
Appendix A: Benchmarks

We used benchmarks from spec2000 compiled for SimpleScalar. We ran programs from both the SpecINT as well as from SpecFP [32].

The Integer benchmark we used includes:

- **GZIP** - is a popular data compression program written by Jean-Loup Gailly <gzip@gnu.org> for the GNU project. `gzip` uses Lempel-Ziv coding (LZ77) as its compression algorithm.

- **BZIP2** - Is based on Julian Seward's bzip2 version 0.1. The only difference between bzip2 0.1 and 256.bzip2 is that SPEC's version of bzip2 performs no file I/O other than reading the input. All compression and decompression happens entirely in memory. This is to help isolate the work done to only the CPU and memory subsystem.

- **VORTEX** – VORTeX is a single-user object-oriented database transaction benchmark which which exercises a system kernel coded in integer C. The VORTeX benchmark is a derivative of a full OODBMS that has been customized to conform to SPEC CINT2000 (component measurement) guidelines.

  The benchmark 255.vortex is a subset of a full object oriented database program called VORTeX. (VORTeX stands for "Virtual Object Runtime EXpository.") Transactions to and from the database are translated through a schema. (A schema provides the necessary information to generate the mapping of the internally stored data block to a model viewable in the context of the application.)

- **VPR** - is a placement and routing program; it automatically implements a technology-mapped circuit (i.e. a netlist, or hypergraph, composed of FPGA logic blocks and I/O pads and their required connections) in a Field-Programmable Gate Array (FPGA) chip. VPR is an example of an integrated circuit computer-aided design program, and algorithmically it belongs to the combinatorial optimization class of programs.

  Placement consists of determining which logic block and which I/O pad within the FPGA should implement each of the functions required by the circuit. The goal is to place pieces of logic which are connected (i.e. must communicate) close together in order to minimize the amount of wiring required and to maximize the circuit speed. This is basically a slot assignment problem -- assign every logic block function required by the circuit and every I/O function required by the circuit to a logic block or I/O pad in the FPGA, such that speed and wire-minimization goals are met. VPR uses simulated annealing to place the circuit. An initial random placement is repeatedly
modified through local perturbations in order to increase the quality of the placement, in a method similar to the way metals are slowly cooled to produce strong objects.

Routing (in an FPGA) consists of determining which programmable switches should be turned on in order to connect the pre-fabricated wires in the FPGA to the logic block inputs and outputs, and to other wires, such that all the connections required by the circuit are completed and such that the circuit speed is maximized. The connections required by the circuit are represented as a hypergraph, and the possible connections of wire segments to other wires and to logic block inputs and outputs are represented by (a different) directed graph, which is often called a "routing-resource" graph.

VPR uses a variation of Dijkstra's algorithm in its innermost routing loop in order to connect the terminals of a net (signal) together. Congestion detection and avoidance features run "on top" of this innermost algorithm to resolve contention between different circuit signals over the limited interconnect resources in the FPGA.

- **GCC** - is based on gcc Version 2.7.2.2. It generates code for a Motorola 88100 processor. The benchmark runs as a compiler with many of its optimization flags enabled. 176.gcc has had its inlining heuristics altered slightly, so as to inline more code than would be typical on a Unix system in 1997. It is expected that this effect will be more typical of compiler usage in 2002. This was done so that 176.gcc would spend more time analyzing it's source code inputs, and use more memory. Without this effect, 176.gcc would have done less analysis, and needed more input workloads to achieve the run times required for SPECint2000.

- **MCF** – is a benchmark derived from a program used for single-depot vehicle scheduling in public mass transportation. The program is written in C, the benchmark version uses almost exclusively integer arithmetic.

  The program is designed for the solution of single-depot vehicle scheduling (sub-problems) occurring in the planning process of public transportation companies. It considers one single depot and a homogeneous vehicle fleet. Based on a line plan and service frequencies, so-called timetabled trips with fixed departure/arrival locations and times are derived. Each of this timetabled trip has to be serviced by exactly one vehicle. The links between these trips are so-called dead-head trips. In addition, there are pull-out and pull-in trips for leaving and entering the depot.
Cost coefficients are given for all dead-head, pull-out, and pull-in trips. It is the task to schedule all timetabled trips to so-called blocks such that the number of necessary vehicles is as small as possible and, subordinate, the operational costs among all minimal fleet solutions are minimized.

For simplification in the benchmark test, we assume that each pull-out and pull-in trip is defined implicitly with a duration of 15 minutes and a cost coefficient of 15.

For the considered single-depot case, the problem can be formulated as a large-scale minimum-cost flow problem that we solve with a network simplex algorithm accelerated with a column generation. The core of the benchmark 181.mcf is the network simplex code "MCF Version 1.2 -- A network simplex implementation", For this benchmark, MCF is embedded in the column generation process.

The network simplex algorithm is a specialized version of the well known simplex algorithm for network flow problems. The linear algebra of the general algorithm is replaced by simple network operations such as finding cycles or modifying spanning trees that can be performed very quickly. The main work of our network simplex implementation is pointer and integer arithmetic.

- **PARSER** – The Link Grammar Parser is a syntactic parser of English, based on link grammar, an original theory of English syntax. Given a sentence, the system assigns to it a syntactic structure, which consists of set of labeled links connecting pairs of words. The parser has a dictionary of about 60000 word forms. It has coverage of a wide variety of syntactic constructions, including many rare and idiomatic ones. The parser is robust; it is able to skip over portions of the sentence that it cannot understand, and assign some structure to the rest of the sentence. It is able to handle unknown vocabulary, and make intelligent guesses from context about the syntactic categories of unknown words.

- **PERLMK** - is a cut-down version of Perl v5.005_03, the popular scripting language. SPEC's version of Perl has had most of OS-specific features removed. In addition to the core Perl interpreter, several third-party modules are used: MD5 v1.7, MHonArc v2.3.3, IO-stringy v1.205, MailTools v1.11, TimeDate v1.08.

- **TWOLF** – is the TimberWolfSC placement and global routing package is used in the process of creating the lithography artwork needed for the production of microchips. Specifically, it determines the placement and global connections for groups of transistors (known as standard cells) which constitute the microchip. The placement
problem is a permutation. Therefore, a simple or brute force exploration of the state space would take an execution time proportional to the factorial of the input size. For problems as small as 70 cells, a brute force algorithm would take longer than the age of the universe on the world's fastest computer. Instead, the TimberWolfSC program uses simulated annealing as a heuristic to find very good solutions for the row-based standard cell design style. In this design style, transistors are grouped together to form standard cells. These standard cells are placed in rows so that all cells of a row may share power and ground connections by abutment. The simulated annealing algorithm has found the best known solutions to a large group of placement problems. The global router which follows the placement step interconnects the microchip design. It utilizes a constructive algorithm followed by iterative improvement.

The basic simulated annealing algorithm has not changed since its inception in 1983. The version in the SPEC suite is the most numerically intensive version. Recent versions have reduced runtimes by clever reductions in the search space. However, the move strategy and cost functions are identical to later versions.

The version of TimberWolfSC that has been submitted to SPEC has been customized for SPEC. It has been modified specifically for the benchmark suite so that it would have a behavior that captures the flavor of many implementations of simulated annealing. The submitted program spends most of its time in the inner loop calculations. By doing this, this version traverses memory often creating cache misses. In fact, this version running small jobs looks like later simulated annealing versions running on large jobs. This was to insure that the benchmark would be applicable to future versions of the program running large instances. The submitted version should be a computers worst nightmare, yet realistic for future problems.

**Floating Point benchmark that we use:**

- **ART** - The Adaptive Resonance Theory 2 (ART 2) neural network is used to recognize objects in a thermal image. The objects are a helicopter and an airplane. The neural network is first trained on the objects. After training is complete, the learned images are found in the scanfield image. A window corresponding to the size of the learned objects is scanned across the scanfield image and serves as input for the neural network. The neural network attempts to match the windowed image with one of the images it has learned.

Description of ART 2
The ART 2 neural network models several characteristics of organic neural processing that is not modeled in more traditional Feed Forward Neural Networks (FFNN). In brief, ART 2 neural networks offer the following advantages over traditional FFNN:

- **Expectation influences inputs** - The past learnings of an ART 2 neural network influence the matching process.
- **Creates own classifications** - During training, the ART 2 neural network does not need explicit output information; it creates its own classification groups.
- **Learns on-the-fly** - ART 2 neural networks are capable of learning and classifying at the same time. The benchmark does not use this feature of ART 2 neural networks.
- **Contrast enhancement** - ART 2 neural networks perform contrast enhancement through a series of normalizations in the dynamical system.

- **MESA** - is a free OpenGL work-alike library. Since it supports a generic frame buffer it can be configured to have no OS or window system dependencies. Any number of client programs can be written to stress FP, scalar or memory performance (or a mix). Output can be written to image files for verification.

- **AMMP** – The benchmark runs molecular dynamics (i.e. solves the ODE defined by Newton's equations for the motions of the atoms in the system) on a protein-inhibitor complex which is embedded in water (see [Harrison 1993](#) for descriptions of the algorithm and stability analysis on it). The energy is approximated by a classical potential or "force field". The protein is HIV protease complexed with the inhibitor indinavir. There are 9582 atoms in the water and protein making this representative of a typical large simulation. This benchmark is derived from published work on understanding drug resistance in HIV ([Weber and Harrison 1999](#)).

- **EQUAKE** - The program simulates the propagation of elastic waves in large, highly heterogeneous valleys, such as California's San Fernando Valley, or the Greater Los Angeles Basin. The goal is to recover the time history of the ground motion everywhere within the valley due to a specific seismic event. Computations are performed on an unstructured mesh that locally resolves wavelengths, using a finite element method.
References


www-3.ibm.com/chips/techlib/techlib.nsf/techdocs


[33] www.intel.com
שימוח אפקטיבי בمتמטטיים שומר נתייב

אורן קצנגרלד
希브ו על מחקר

לשם מייליגי חלקי של הדירישות לجابלה החוזה ממיסטר
למדיעים במודיעי המחשב

اورן כצנלסון

הוגה לסטט הטכניון - מכון טכנולוגי לישראל
חיפה
מרץ 2006
המחק הרשום מהנדסית ד"ר. אבי מנדלבך ופורפ.ذهب הא.CompareTo בפתקולנוע למעדי
המחשב

אני מוזה לצביני על התמקדות הסיפת הנדסת בתחום

אני מוזה לקורב לפי לשבני המאתים של"ד אבי מנדלשטי ופורפ' בהר אסף שופט על
כל הגוזה וההימיכת החקפה המחקרញورة
אני מוזה לקורב לפי מונולוג על הצלאת שונים לנוסך מחקר, על המקור
הגרירתימ על הביצועים שלבי ירך על כ שמתים היה זז מקצוד
אני מוזה לקורב. חזר אסף שופט על כל הגוזה ה螋נית, הגרירתימ המגרנוגופים מחוק
ה},${

אני מוזה לקורב מ_allocator DSP ב-ולגרר איפסל בפרט על הגוזה ניתן גח דירש
למחקור

אני מוזה לקורב מנוכני, למיכריםeda שמי, ד"ורית אסא, מצחי שפידי וידדה קולות
על כל הגוזה והסכלות שהספינו לקודר:
TC

Trace Cache

SimpleScalar

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC

TC
זוייר מספר 6 - תוצאת הנסיכים של מנהיגות עם ולעפים זרעים שרים דינמיים, עם ובל רימוי, החלוקה
70
71
72
73
74
רשימת טבלאות

15 \textit{SimpleScalar} המпомнים של התחsemblies

46 \textit{usage} ארגוב בני תשתית TC גבולות נוספים

48 \textit{usage} \textit{cut & merge} ב.ItemStack פילר \textit{TC}

52 \textit{usage}\textit{TC} \textit{inline functions} מתן מטרות \textit{TC} (ממצאים)

75 \textit{usage} \textit{perl} "חלוקה זו" \textit{perl} \textit{perl} \textit{perl} "חלוקה זו" \textit{perl} \textit{perl} "חלוקה זו" \textit{perl} \textit{perl} "חלוקה זו"
trace cache
trace /trace
LRU

commit

# EOF

# EOF
ארבע קומבינטוריות פרמטריות מתנהלות שצפונה: \text{TC} \leftarrow \text{trace} \leftarrow \text{trace} \leftarrow \text{TC} \leftarrow \text{trace} \leftarrow \text{trace} \leftarrow \text{TC}

\text{TC} \leftarrow \text{TC} \leftarrow \text{TC} \leftarrow \text{TC} \leftarrow \text{TC} \leftarrow \text{TC} \leftarrow \text{TC}

ולשורי

לשם הרשת\text{TC} \leftarrow \text{TC} \leftarrow \text{TC} \leftarrow \text{TC} \leftarrow \text{TC} \leftarrow \text{TC} \leftarrow \text{TC}

המקודד \text{TC} \leftarrow \text{TC} \leftarrow \text{TC} \leftarrow \text{TC} \leftarrow \text{TC} \leftarrow \text{TC} \leftarrow \text{TC}

בफטריות והמר

לעופר פתרון בטיע, מתמר במציע של שיא פ톨א מילה ציון. שליט ארבעה מציגה שפע

בנוגא Ents עם נוספים של חברת \text{TC} \leftarrow \text{TC} \leftarrow \text{TC} \leftarrow \text{TC} \leftarrow \text{TC} \leftarrow \text{TC} \leftarrow \text{TC}

ולשורי registrado החברה עם פורום רומא. בступил ונכון הקופרטיור מבכירו בהרמוניה לא כל \text{TC} \leftarrow \text{TC} \leftarrow \text{TC} \leftarrow \text{TC} \leftarrow \text{TC} \leftarrow \text{TC} \leftarrow \text{TC}

רומיו אל מוריסים בולא במוריס ילעגר וללא אחר \text{TC} \leftarrow \text{TC} \leftarrow \text{TC} \leftarrow \text{TC} \leftarrow \text{TC} \leftarrow \text{TC} \leftarrow \text{TC}
The document contains Hebrew text only. It appears to be a thesis or research paper from the Technion - Computer Science Department - M.Sc. Thesis MSC-2006-26 - 2006. The content is not legible due to the quality of the image.