Software Trace Cache *

Alex Ramírez   Josep-L. Larriba-Pey   Carlos Navarro   Josep Torrellas†
Mateo Valero

Computer Architecture Department
Universitat Politècnica de Catalunya
Jordi Girona 1-3, Module D6
08034 Barcelona (Spain)
{aramirez,larri,cnavarro,mateo}@ac.upc.es

Abstract

In this paper we address the important problem of instruction fetch for future wide issue superscalar processors. Our approach focuses on understanding the interaction between software and hardware techniques targeting an increase in the instruction fetch bandwidth. That is the objective, for instance, of the Hardware Trace Cache (HTC).

We design a profile based code reordering technique which targets a maximization of the sequentiality of instructions, while still trying to minimize instruction cache misses. We call our software approach, *Software Trace Cache (STC)*.

We evaluate our software approach, and then compare it with the HTC and the combination of both techniques. Our results show that for large codes with few loops and deterministic execution sequences like databases and some SPEC-INT codes, the STC offers similar, or better, results than a HTC. Moreover, when combining the software and hardware approaches, we obtain encouraging results: the STC and a small HTC offer similar performance to a much larger HTC alone.

1 Introduction

Instruction fetch bandwidth may become a major limiting factor for future aggressive wide-issue superscalars. Consequently, it is crucial to develop software and hardware techniques that interact to deliver multiple basic blocks to the processor every cycle.

Unfortunately, for many important codes, this is hard to do. For instance, database codes and several integer SPEC applications have frequent control flow transfers and high instruction-cache miss rates. These characteristics make supplying a high number of useful instructions a difficult task, even in the presence of aiding devices like the Hardware Trace Cache (HTC) [4, 12].

On the software side, it is possible to reorder the code in memory so that it is easier to supply useful instructions to the execution unit. Code reordering can target the elimination of cache conflicts [5, 6, 8, 7, 10, 13]. In addition, it can also map sequentially-executed basic blocks in consecutive memory positions [7, 10, 13]. Both aspects may increase the number of useful instructions fetched per access for future wide-issue superscalars.

In this paper, we focus on the interaction between hardware and software to provide a high instruction bandwidth. We start presenting a fully-automated, compile-time code reordering technique that focuses on maximizing the sequentiality of instructions, while still trying to minimize instruction cache misses. We call our technique *Software Trace Cache (STC)*.

We compare the results obtained with the STC to those obtained with a HTC alone and to the combination of both techniques. The results are obtained for the PostgreSQL database, an arcade game, and the SPEC'95 benchmark.

Our results show that for large codes with few loops and deterministic execution sequences like the Postgres95 database management system, the STC offers similar, or better, results than a HTC. Moreover, if we combine the STC and the HTC, we obtain very encouraging results. Specifically, the number of fetched instructions per cycle obtained with a combination of the STC and a small HTC is comparable to that of a HTC of double size alone. Finally, the STC can be useful even in combination with a large HTC due to the instruction cache miss rate reduction.

This paper is structured as follows: Section 2 describes the fetch mechanism and the HTC; Section 3 describes the STC technique; Section 4 characterizes the instruction reference stream for a variety of workloads; Section 5 uses simulations to evaluate various combinations of the STC and the HTC; Section 6 discusses related work; and in Section 7 we draw some final remarks.

2 The Fetch Mechanism

We simulate an aggressive sequential fetch unit similar to that described in [12]. As shown in Figure 1, our core fetch unit model is composed of an interleaved instruction cache (i-cache), a multiple branch predictor (BP), an in-

*This research has been supported by CICYT grant TIC-0511-98 (UPC authors), the Generalitat de Catalunya grants ACI 97-26 (Josep L. Larriba-Pey and Josep Torrellas) and 1998FI-00306-APTIND (Alex Ramírez), the Commission for Cultural, Educational and Scientific Exchange between the United States of America and Spain (Josep L. Larriba-Pey, Josep Torrellas and Mateo Valero), NSF grant MIP-9619351 (Josep Torrellas) and CEPBA. Alex Ramírez wants to thank all his fellow PBC's for their time and efforts. The authors want to thank Xavi Serrano for all his help setting up and analyzing PostgreSQL.

†University of Illinois at Urbana Champaign, USA. (torrella@cs.uiuc.edu)

terleaved branch target buffer (BTB) and a return address stack (RAS). This fetch unit is designed to fetch as many contiguous instructions as possible. The limits are posed by the width of the data path and the branch predictor throughput. In this work, we will assume a limit of 16 instructions and 3 branches per cycle.

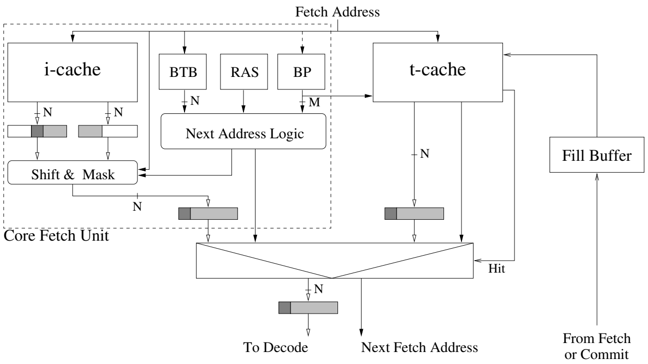

Figure 1: Fetch unit model used for simulation, complete with the Hardware Trace Cache mechanism. In our simulations N=16 and M=3.

Two consecutive i-cache lines are accessed per cycle, allowing us to fetch sequential code crossing the cache line boundary. The BTB is accessed in parallel with the i-cache, and is used to predict the address of indirect jumps and subroutine calls. The return address of subroutines can be accurately predicted using the RAS. It is assumed that instructions will be pre-decoded, allowing branches and other control transfers to be detected. The target of PC-relative branches is calculated, not obtained from the BTB. Using the outputs from the BP, the BTB and the information regarding which instructions represent control transfers, we obtain an instruction mask to select the valid instructions from the fetched i-cache lines, and generate the fetch address for the next cycle.

We do not allow our fetch unit to stop at indirect jumps if those do not break the execution sequence. We add a *breaking* bit to the BTB, which informs the fetch unit when the predicted jump address will break the sequence.

Summarizing, instruction fetch stops in one of these conditions:

- 16 instructions are fetched
- 3 branches are fetched
- A branch is predicted taken
- An indirect jump is predicted to break the execution sequence
- A system call is fetched
- On any misprediction or BTB miss

For high branch prediction accuracy we use a 4KB GAg correlated branch predictor, with 14-bit history length, extended to allow multiple branches to be predicted in a single cycle, a 256-entry BTB enhanced with the *breaking* bit, and a 256-entry RAS.

We also simulated our core fetch unit in conjunction with the basic trace cache model described in [12]. The complete fetch unit, with a trace cache (t-cache), is what we call the Hardware Trace Cache (HTC). The fill buffer reads instructions from either the fetch unit (speculative trace construction) or from the commit stage (non-speculative trace construction) and stores them in a special purpose buffer. When a trace is completed, it is stored in the t-cache in conjunction with the branch outcomes that led to that instruction sequence. If the same leading instruction and the same branch outcomes are encountered in the future, the trace is fed directly from the t-cache to the decode unit. The fill buffer stops building a trace on the same conditions as the core fetch unit, except for the case of sequence breaks, as the t-cache is able to store non-contiguous instructions in contiguous memory positions.

3 The Software Trace Cache

The number of useful instructions per cycle provided to the processor is broadly determined by three factors: branch prediction accuracy, instruction cache miss rate and the execution of non-contiguous basic blocks. To deal with the last two problems, we propose a code reordering technique which uses the whole memory space as a *Software Trace Cache* to store the most popular sequences of basic blocks. In order to avoid sequence breaks we will reorder the basic blocks in a program to change taken branches to non-taken ones, moving unused basic blocks out of the execution path and inlining basic blocks from the most popular functions. To reduce the instruction cache miss rate we map the most popular traces in a reserved area of the i-cache.

Our algorithm is based on profile information. This means that the results obtained will depend on the representativity of the training inputs. The most popular execution paths for a given input set do not need to be related to the execution paths of a different input set.

Running the training set on each benchmark, we obtain a directed graph of basic blocks with weighted edges. An edge connects two basic blocks p and q, if q is executed after p. The weight of an edge $W(pq)$ is equal to the total number of times q has been executed after p. The weight of a basic block $W(p)$ can be obtained by adding the weight of all outgoing edges. The branch probability of an edge $B(pq)$ is obtained as $W(pq)/W(p)$. All unexecuted basic blocks are pruned from the graph.

Next we describe how we select the *seeds* or starting basic blocks for our code sequences, the algorithm which builds the basic block traces from the selected seeds, and the mapping algorithm used to allocate these traces minimizing instruction cache misses.

3.1 Seed selection

We obtain an ordered list of seeds by sorting the entry points of all functions in decreasing frequency of execution. This tries to expose the maximum temporal locality, as the first traces built will start on the most frequently referenced functions.

It is possible to obtain better results with a seed selection based on the internal structure of the code, as we show in [11]. However, access to the source code of applications is not always granted, and gaining such deep understanding of the code is a time consuming task, which may not offer an improvement large enough to compensate for the effort.

3.2 Trace building

Using the weighted graph obtained running the training set, and starting from the selected seeds, we implement a greedy algorithm to build our basic block traces targeting an increase in the code sequentiality. Given a basic block, the algorithm follows the most frequently executed path out of it. This implies visiting a subroutine called by the basic block, or following the control transfer with the highest probability of being used. All the other valid transitions from the basic block are noted for future examination.

For this algorithm we use two parameters called *Exec Threshold*, and *Branch Threshold*. The trace building algorithm stops when all the successor basic blocks have been visited or have a weight lower than the Exec Threshold, or all the outgoing arcs have a branch probability less than the Branch Threshold. In that case, we start again from the next acceptable transition, as we noted before, building secondary execution paths for the same seed. Once all basic blocks reachable from the given seed have been included in the main or secondary sequences, we proceed to the next seed.

Figure 2.a shows an example of the weighted graph and Figure 2.b shows the resulting sequences. We use an ExecThresh of 4 and a BranchThresh of 0.4. Starting from seed $A1$ and following the most likely outgoing edge from each basic block we build the sequence $A1 \rightarrow A8$ (Figure 2.b). The transitions to $B1$ and $C5$ are discarded due to the Branch Threshold. We noted that the transition from $A3$ to $A5$ is a valid transition, so we start a secondary trace with $A5$, but all its successors have been already visited, so the sequence ends there. We do not start a secondary trace from $A6$ because it has a weight lower than the Exec Threshold.

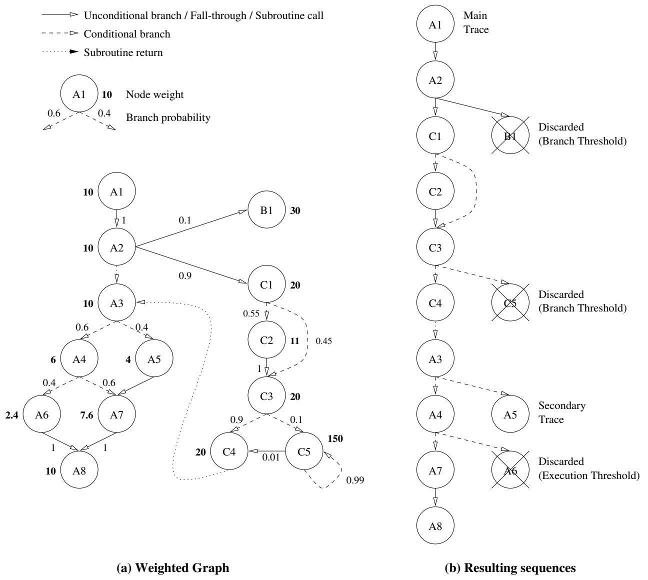

Figure 2: Trace building example.

Code replication

In order to increase code sequentiality, we introduce a limited form of code replication in out method. We allow the main execution path of each subroutine to be replicated at all call points.

We introduce two new threshold values, the *ExecRep* and *BranchRep* Thresholds to control the amount of code we are replicating. A sequence will be replicated if the call probability passes the BranchRep threshold and the starting basic block for that sequence passes the ExecRep Threshold.

In the example from Figure 2, if we found a new call to $C1$ from, say $A7$, we would replicate the sequence $C1 \rightarrow C4$ and include it between $A7$ and $A8$ in the main execution path.

Threshold selection

We have a loop in our algorithm that repeatedly selects a set of values for the four thresholds and generates the resulting traces for each pass. The basic blocks included in previous passes are pruned from the newly formed traces, further limiting the amount of code replicated in each pass. By iteratively selecting less and less restrictive values for the thresholds, we build our traces grouped in passes of decreasing frequency of execution.

The values selected for the Exec and Branch threshold will determine the number of basic blocks included in each pass of the algorithm, generating larger or smaller groups of traces. The target is to pack in a given pass those traces with a similar popularity, while keeping the total number of instructions under control. For this paper we selected our Exec Threshold so that each pass contained approximately 4KB of not replicated code. To maximize the effect of the code replication we used the least restrictive ExecRep and BranchRep thresholds.

3.3 Trace mapping

As shown in Figure 3, we map our code sequences in decreasing order of popularity, concentrating the most likely used code in the first memory pages and mapping popular sequences close to other equally popular ones, reducing conflict misses among them. Also, the most popular sequences will map to a reserved area of the cache, leaving gaps to create a Conflict Free Area (CFA), shielding the most popular traces from interference with any other code.

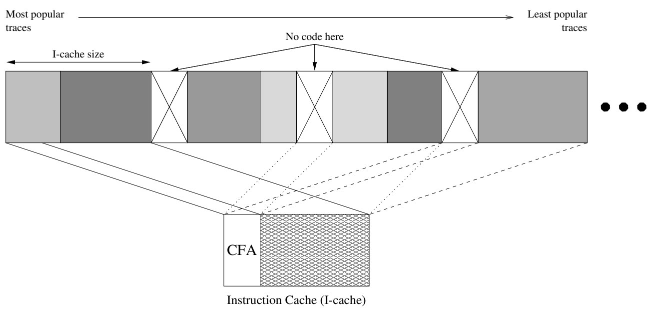

Figure 3: Trace mapping for a direct mapped instruction cache.

The same mapping algorithm can be applied to set associative cache with minor modifications. A complete study of the different factors which determine the instruction cache miss reduction offered by this mapping of code sequences, and a comparison with other code mapping algorithms for the PostgreSQL database can be found in [11].

4 Locality Study

The objective of the STC is to build at compile time the most popular traces that are built at run time by the HTC. Also, the STC targets a minimization of the i-cache miss rate at the same time. We analyze the instruction reference stream for a wide set of workloads, characterizing instruction locality and execution path determinism, which affect the performance offered by the STC. With this information we intend to predict the performance increase we can expect when using the STC for each workload. The reference locality will affect the i-cache miss rate reduction offered by our technique, and the basic block size, the number of loops and the determinism of program execution will influence the increase in code sequentiality accomplished by the basic block reordering.

4.1 Workloads

We have used four classes of workloads, trying to cover a wide range of applications: common integer and floating point codes, commercial workloads, and arcade games.

Recent studies have shown that commercial workloads do not behave like common integer codes, like the SPEC-INT set. Also, it is well known that floating point codes have a different behavior than integer codes. Our workloads include the whole SPEC 95 benchmarks. We use the PostgreSQL 6.3.2 database management system as our commercial workload and XBlast 2.2, an arcade game, as an example of a little studied workload.

All the executions and simulations needed to develop this work, have been done using the Alpha 21164 processor, DEC ATOM, and trace driven simulation. We used different input sets to obtain the profile information and to obtain the simulation results to ensure that the improvements were valid for inputs other than the profiled ones. All benchmarks were run to completion for both training and simulation.

4.2 Code Analysis

Examining the profile information obtained running the training set, we classify the workloads attending to those characteristics that affect the performance of our technique: code locality, the amount of loops, conditional branches and subroutine calls, the basic block size and the number of sequence breaks.

To examine code locality, we determine the number of static instructions needed to gather 75, 90 and 99% of the dynamic instruction references as shown in Table 1. The total code size for each benchmark and the CFA size we selected for 32 and 64KB instruction caches are also shown in Table 1. We observe that some codes have very large working sets, like *applu*, *apsi*, *fpppp*, *gcc* and *postgres* which do not fit even in 32KB caches. Furthermore, some codes exhibit very little temporal locality, like *gcc*, which can not fit 75% of the references in a 32KB cache.

We select the CFA size so that it gathers between 75 and 90% of the instruction references, while still leaving reasonable space for the rest of the code. Obviously, larger caches allow a larger CFA and more code replication. For example, *xblast* concentrates 90% of the dynamic references in 2362 instructions (9448 bytes) which almost fit in an 8KB CFA, which we will use for a 32KB instruction cache, but for a 64KB cache we will allow the CFA to grow to 16KB.

Next, we examine the code sequentiality for the original layout. We observe that all floating point benchmarks have very large basic blocks (35 instructions average), leading to large code sequences (57 consecutive instructions average). Meanwhile, the average sequence length for the integer benchmarks is usually under 12 instructions, as less than 2 consecutive basic blocks are executed, and the typical basic block size is around 5–7 instructions.

Benchmark	Dynamic references			Code	CFA size	
	75%	90%	99%	size	32KB	64KB
101.tomcatv	223	308	1328	108237	8	8
102.swim	148	232	763	110350	4	4
103.su2cor	979	1839	4197	129741	8	16
104.hydro2d	1223	1977	5371	125946	8	16
107.mgrid	147	218	1029	112421	4	4
110.applu	2407	5060	10509	132803	16	24
125.turb3d	1065	1771	2828	121181	8	8
141.apsi	3099	5694	9883	156479	16	24
145.fpppp	8985	8985	9879	124970	8	32
146.wave5	1116	1919	5506	154987	8	24
124.m88ksim	458	1006	2863	51341	8	16
126.gcc	9595	22098	57878	349382	8	16
129.compress	243	338	525	21991	4	8
130.li	325	563	1365	38126	8	8
132.ijpeg	862	1489	3271	67646	8	16
134.perl	987	1582	3006	108227	8	16
147.vortex	751	1486	5128	172690	8	24
postgres	2716	5221	11748	374399	16	16
xblast	1100	2362	6326	430664	8	16

Table 1: Number of static instructions needed to accumulate 75, 90 and 99% of the dynamic references, and the total code size, including unreferenced instructions. Selected CFA size for 32 and 64KB instruction caches.

Finally, we examined a classification of the dynamic basic blocks executed by each benchmark. The different types of basic block considered are shown in Table 2. The percentage of basic blocks of each type executed is shown in Table 3. The last two columns show the percentage of branch and loop basic blocks which behave in a fixed way (FB,FL), that is, they are always taken or always not taken. A low proportion of fixed loop branches means that each loop executes few iterations (less than 20).

BB Type	Description	Target
F	Fall-through	Next instruction
J	Unconditional branches	PC relative
j	Unconditional branches	Indirect
B	Conditional branches	PC relative
L	Loop branches	PC relative
S	Subroutine call	PC relative
s	Subroutine call	Indirect
R	Subroutine returns	Indirect

Table 2: Basic block types considered.

By changing the order of the basic blocks in a program we can reduce the number of unconditional branches, and change taken conditional branches for not taken ones. Also, by inlining the most popular functions we can eliminate subroutine calls and returns, and increment the number of sequentially executed instructions. Note that the number of sequence breaks due to loop branches and unpredictable conditional branches does not depend on the organization of the code.

We consider the indirect jumps in a separate way because they can not be eliminated, as the target address is unknown, and may jump to an unexpected address. However, we can reorder the code so that the most frequent target address does not break the execution sequence.

To reduce the number of loops, compiler optimizations like loop unrolling can be used, but it is not yet included in our work. Consequently, the STC as it is now will offer little advantage to codes with lots of loops and few fixed conditional branches.

Also, codes with few subroutine calls will not benefit from the fact that the STC builds its execution sequences crossing procedure calls.

The number of predictable basic block transitions is determined by fall-through basic blocks, PC-relative unconditional branches, conditional branches with a fixed behavior and subroutine calls. The percentage of fall-through basic blocks is around 10–20% for most codes, so STC performance will be determined by the rest of the basic block classes, mainly by the percentage of loop basic blocks.

With this criteria, we expect *postgres*, with only a 3.4% of loop basic blocks, 12.8% of subroutine calls and 43% of conditional branches (76.2% of which behave in a fixed way) to be the one which will benefit the most from the STC. On the other hand, 32% of the basic blocks executed by *ijpeg* end with a loop branch, and barely 30% of its conditional branches behave in a fixed way, which makes it difficult to enlarge the execution sequences. Among the FP codes, *apsi* looks as the best candidate, with a large proportion of fixed conditional branches and few loops but few subroutine calls.

5 Simulation results

After selecting an appropriate CFA size for each benchmark, we measured the increase in the number of instructions executed between two sequence breaks obtained with the STC. Table 4 shows the percentage of basic block transitions which break the sequence, and the average number of consecutive instructions executed for each benchmark for both the original code and our proposed layout, measured running the Test set.

The number of consecutive instructions executed represents the performance limit of a sequential fetch unit. Even if we were not limited by the bus width, the branch predictor throughput, and branch mispredictions, we would still be limited by taken branches. Table 4 shows how the STC improves this performance limit.

Benchmark	Average BB size	Original		Reordered	
		%breaks	Seq len.	%breaks	Seq len.
tomcatv	44.0	80	55.2	72	61.4
swim	48.7	99	49.2	99	49.1
su2cor	19.8	52	37.7	49	40.1
hydro2d	14.9	69	21.6	53	28.3
mgrid	62.0	89	70.0	90	68.8
applu	23.4	50	46.5	58	40.7
turb3d	21.9	47	46.7	36	60.6
apsi	26.3	55	48.0	44	59.4
fpppp	69.1	43	162.5	40	171.6
wave5	24.6	62	39.5	61	40.4
Average	35.5	65	57.7	60	62.0
m88ksim	4.82	61	7.9	27	17.8
gcc	5.33	55	9.8	40	13.4
compress	6.77	58	11.7	62	10.9
li	4.20	49	8.5	37	11.2
ijpeg	16.7	68	24.4	65	25.9
perl	5.36	54	10.0	38	13.9
vortex	4.76	55	8.7	27	17.6
Average	6.85	57	11.6	42	15.8
postgres	4.58	51	9.0	25	18.3
xblast	5.25	62	8.4	27	19.5

Table 4: Percentage of basic block transitions and average number of consecutive instructions executed for the original and the reordered code. The average BB size is the same for both code layouts.

As expected, the FP benchmarks barely reduce the percentage of sequence breaks. The best results are obtained by *hydro2d* and *apsi*, with reductions between 20–25%, which translate to sequence length increases of 24–31%. This was to be expected due to the reduced proportion of loops executed. On the other hand, *mgrid* actually increased the percentage of sequence breaking BB transitions from 89% to 90%. It is the FP benchmark with a higher proportion of loop basic blocks executed.

For the integer codes, we obtain sequence length increases above 100% for *m88ksim*, *vortex*, *postgres* and *xblast*. Meanwhile, *ijpeg* did not experience any noticeable improvement. This roughly corresponds to what we expected from Section 4.

In general terms, most integer codes experience significant reductions in the percentage of sequence breaking BB transitions. After reordering, most codes execute 2–3 consecutive basic blocks, raising the average performance limit to 15.8 instructions.

5.1 Fetch unit simulation

Table 5 shows simulation results for the fetch unit described in Section 2, using a 32KB instruction cache (i-cache). We simulated both code layouts on the core fetch unit, and in combination with trace caches (t-cache) of 16 and 32KB. The code layout is either the original code (Base), or the optimized layout corresponding to a CFA of xKB (CFA$_x$).

We present the number of Fetched Instructions per Access (FIPA) as three separate results, the average number of instructions the core fetch unit (i-cache) provides, the average number of instructions the t-cache provides, and the average global performance. If no t-cache is present, the core fetch unit is the same as the global performance.

Also, separate i-cache and t-cache miss rates are presented in terms of misses per line access. There are two i-cache line accesses and one t-cache line access for each fetch unit access.

We also present the branch misprediction rate.

The final performance metric is the number of Fetched Instructions per Cycle (FIPC). The FIPC was obtained dividing the FIPA for an estimated number of cycles per access (CPA). Instruction cache misses cause the fetch engine to stall, increasing the CPA, and branch mispredictions cause the fetch unit to fetch instructions from the wrong execution path, effectively wasting fetch cycles.

We used a fixed number of cycles for each i-cache miss, and assumed that if both i-cache lines missed, they could be served simultaneously. We also assumed an average number of penalty cycles for each branch misprediction. As i-cache miss penalties we used 3 and 6 cycles, and branch misprediction penalties of 4, 8 and 12 cycles, as it will depend on the execution core of the processor.

Software Trace Cache

The main effects of the STC are an increase in the FIPA provided by the core fetch unit and a reduction of the i-cache miss rate. Some codes show large improvements in one or both numbers, while others seem unaffected. We found some unexpected side effects on the branch prediction accuracy.

For example, reordering the code for *postgres* increases the FIPA for the core fetch unit from 7.5 to 10.3 instructions. These 10.3 instructions per access are still far away from the 18.3 shown in Table 4, but that is a performance limit. Here we are limited by the bus width and the branch predictor throughput and accuracy, not only by taken branches.

	Basic Block Type								Fixed branches	
Benchmark	F	J	j	B	L	S	s	R	FB	FL
101.tomcatv	3.3	4.1	3.1	28.2	59.0	0.2	1.0	1.2	92.0	49.3
102.swim	7.1	5.9	2.0	19.4	57.5	0.0	3.9	3.9	54.7	100.0
103.su2cor	11.8	5.9	2.0	44.4	25.3	0.1	5.2	5.3	65.9	92.4
104.hydro2d	9.8	1.6	0.6	46.3	39.1	0.1	1.3	1.3	66.5	88.1
107.mgrid	5.2	0.1	0.1	13.2	81.1	0.0	0.1	0.1	81.1	0.0
110.applu	11.1	0.0	0.0	43.3	45.4	0.0	0.0	0.0	58.8	13.0
125.turb3d	11.8	6.4	0.4	46.9	29.3	2.2	0.4	2.6	81.5	14.8
141.apsi	17.8	2.9	0.0	44.7	23.7	3.2	2.2	5.4	77.9	18.8
145.fpppp	19.8	2.7	2.7	56.3	9.6	0.0	4.5	4.5	46.2	16.9
146.wave5	15.2	1.7	0.8	35.2	31.9	2.5	5.1	7.6	90.5	88.3
124.m88ksim	11.7	6.9	0.4	47.3	9.4	2.0	10.2	12.2	62.7	53.6
126.gcc	9.5	4.2	2.2	58.8	11.3	4.0	3.0	7.0	39.9	21.7
129.compress	12.3	5.9	0.0	37.0	16.5	14.1	0.0	14.1	48.2	44.7
130.li	21.1	3.2	1.9	39.8	11.6	7.3	3.9	11.2	44.4	50.8
132.ijpeg	13.8	5.0	0.0	43.7	32.1	1.6	1.1	2.7	29.8	35.1
134.perl	21.1	3.6	2.6	45.9	9.2	2.1	6.7	8.8	69.0	53.4
147.vortex	16.0	6.2	0.5	44.4	17.8	0.1	7.5	7.6	63.7	32.6
postgres	22.1	2.5	1.2	43.4	3.4	12.8	0.9	13.6	76.2	26.3
xblast	20.8	1.0	0.1	50.1	11.2	2.2	6.2	8.4	65.1	13.5

Table 3: Percent of the dynamic basic blocks of each type for each workload. Percent of conditional and loop branches with a fixed behavior.

This increase causes the global FIPA performance of the STC alone to come quite close to the HTC for some codes. For example, the core fetch unit provides 10.1 instruction per access with the reordered *xblast*, while using a 16KB t-cache with the original code obtains 10.8 instructions per access.

We also observe that the STC drastically reduces the i-cache miss rate for both the FP and the integer codes. Large codes like *apsi*, *xblast* and *postgres* obtain miss rate reductions around 90%, and a final i-cache miss rate around 1% on a 32KB cache.

The branch misprediction rate increases slightly with the reordered code. It is so because the reordered code reduces the number of taken branches, introducing more zeroes in the history register of the GAg predictor, leading to a worse utilization of the history table. For example, the branch misprediction rate for *m88ksim* increases from 4.9% to 7.9% when the code is reordered, mainly due to the reduction of sequence breaks from 61% to 27% (see Table 4).

Hardware Trace Cache

The fact that the traces provided by the t-cache are built in the fill buffer makes the FIPA provided from the t-cache independent of the t-cache geometry. Also, reading the instructions form the dynamic stream, makes the traces independent of the code layout.

The t-cache miss rate does not seem to depend on the code layout, but on the t-cache size itself. In order to reduce the t-cache miss rate, techniques like *Partial matching* [4] have been proposed, which also reduce the number of instruction provided. The HTC mechanism assumes that the t-cache will always be able to provide more instructions than the core fetch unit. This statement may not be true if we consider the increased FIPA performance of the STC and the reduced trace length caused by such techniques. It may not be worth adding such functionality to the HTC, as a compile-time optimization can obtain similar results.

The HTC does not have a visible impact on the branch prediction accuracy.

STC and HTC interaction

The best results are obtained when combining both STC and HTC, as the core fetch unit will be able to provide more instructions on a t-cache miss and a lower i-cache miss rate. For example, a combination of STC and 16KB t-cache for *vortex* provides 12.2 instructions per access, while a HTC of 32KB provides only 11.3.

For the larger codes, the t-cache can not remember all the executed code sequences, and the core fetch unit is used extensively. It is in these cases, like *gcc*, where the STC proves more useful, combining with a small t-cache to provide better results than a t-cache of double size alone. Combining a STC with a large t-cache still improves the results for the larger codes, but the FIPA increase is minimum for the small ones.

Both the STC and the HTC improve the FIPA, but the STC also targets a reduction of the CPA by minimizing i-cache misses. For small codes, like *hydro2d* which already have a very low i-cache miss rate, the STC is not too useful, with the HTC providing much better performance. In some cases, like *li*, the increased branch misprediction rate can actually hinder the performance of the HTC.

On the other hand, for the largest codes, like *postgres* and *xblast*, the STC alone can offer similar, or better results than a HTC alone. In these cases, combining the compile-time and the run-time techniques offers the best results, raising the FIPC from 5.9 with the STC, or 4.6 with the 16KB HTC, to 6.5 with a combination of both. In most cases, using a combination of STC and a small HTC offers similar or better results than a HTC of double size.

The benefits of the STC are more obvious when the i-cache miss penalty increases and the branch misprediction penalty is small, while the HTC proves most useful when the i-cache miss penalty is low.

We conclude that for large codes with few loops, the STC can provide better results than the HTC alone, and that a combination of the STC with a small HTC provides similar or better results than a much larger HTC alone. When combined with a large t-cache, the STC is still able to provide performance improvements, due to the reduced i-cache miss rate.

Bench.	Setup		FIPA			Miss rate (%)		Branch	FIPC [miss pen,br pen]			
	t-cache	Layout	i-cache	t-cache	global	i-cache	t-cache	mispr. (%)	6/8	6/12	3/8	6/4
hydro2d	0	Base	11.7	—	11.7	0.08	—	0.3	11.5	11.4	11.5	11.6
	0KB	CFA_8	12.2	—	12.2	0.01	—	0.4	11.9	11.8	11.9	12.0
	16KB	Base	11.4	15.5	15.4	0.09	3.3	0.3	15.0	14.8	15.0	15.2
	16KB	CFA_8	11.1	15.5	15.5	0.01	3.9	0.4	15.0	14.8	15.0	15.2
	32KB	Base	9.1	15.5	15.5	0.08	0.5	0.3	15.0	14.8	15.1	15.2
	32KB	CFA_8	9.9	15.5	15.5	0.01	0.7	0.4	15.0	14.8	15.0	15.2
apsi	0KB	Base	13.8	—	13.8	1.3	—	2.8	11.8	11.4	12.4	12.3
	0KB	CFA_{16}	14.2	—	14.2	0.2	—	3.3	12.8	12.2	12.8	13.4
	16KB	Base	13.5	15.9	15.3	1.5	23.1	2.8	12.9	12.4	13.5	13.4
	16KB	CFA_{16}	13.7	15.9	15.4	0.2	21.0	3.3	13.7	13.1	13.8	14.4
	32KB	Base	13.3	15.9	15.5	1.4	16.5	2.8	13.0	13.5	13.7	13.6
	32KB	CFA_{16}	13.7	15.9	15.5	0.2	16.5	3.3	13.8	13.2	13.9	14.5
m88ksim	0KB	Base	6.9	—	6.9	11.6	—	4.9	3.2	3.0	4.1	3.5
	0KB	CFA_8	10.0	—	10.0	0.06	—	7.9	5.5	4.5	5.5	7.1
	16KB	Base	6.5	13.8	11.1	14.5	36.9	4.9	4.4	4.0	5.7	5.0
	16KB	CFA_8	8.8	13.7	11.6	0.06	42.7	7.9	6.0	4.8	6.0	7.9
	32KB	Base	6.4	13.7	11.9	13.5	25.4	4.9	4.7	4.2	6.1	5.4
	32KB	CFA_8	8.1	13.8	12.0	0.06	31.5	7.9	6.1	4.9	6.1	8.1
gcc	0KB	Base	7.7	—	7.7	10.2	—	10.0	3.2	2.7	3.7	3.8
	0KB	CFA_8	8.8	—	8.8	7.3	—	11.8	3.4	2.8	3.9	4.3
	16KB	Base	7.1	13.6	9.8	12.4	59.4	10.0	3.5	2.9	4.2	4.2
	16KB	CFA_8	8.2	13.3	10.5	8.3	56.0	11.8	3.7	3.0	4.2	4.7
	32KB	Base	6.9	13.6	10.1	12.4	51.6	10.0	3.6	3.0	4.2	4.4
	32KB	CFA_8	8.0	13.3	10.7	8.3	48.6	11.8	3.7	3.0	4.2	4.8
li	0KB	Base	7.4	—	7.4	0.1	—	5.0	5.3	4.6	5.3	6.1
	0KB	CFA_4	8.4	—	8.4	0.1	—	5.8	5.5	4.7	5.6	6.6
	16KB	Base	6.4	13.5	10.6	0.2	40.4	5.0	6.8	5.8	6.8	8.2
	16KB	CFA_4	7.3	13.2	11.1	0.2	36.4	5.8	6.6	5.5	6.6	8.2
	32KB	Base	6.1	13.4	11.1	0.2	32.0	5.0	7.0	5.9	7.0	8.5
	32KB	CFA_4	6.8	13.4	11.5	0.2	29.0	5.8	6.8	5.6	6.8	8.5
ijpeg	0KB	Base	11.7	—	11.7	0.07	—	9.7	8.3	7.2	8.3	9.7
	0KB	CFA_8	11.6	—	11.6	0.01	—	10.1	8.2	7.1	8.2	9.6
	16KB	Base	9.7	15.5	14.5	0.1	17.4	9.7	9.6	8.2	9.6	11.4
	16KB	CFA_8	10.3	15.4	14.5	0.01	18.4	10.2	9.5	8.1	9.5	11.5
	32KB	Base	8.8	15.5	14.5	0.1	14.4	9.7	9.6	8.2	9.6	11.5
	32KB	CFA_8	9.0	15.4	14.6	0.01	14.6	10.1	9.5	8.1	9.6	11.5
vortex	0KB	Base	7.6	—	7.6	7.8	—	7.4	3.5	3.1	4.2	4.1
	0KB	CFA_8	10.2	—	10.2	1.1	—	6.2	5.9	5.0	6.1	7.2
	16KB	Base	6.7	13.8	10.8	8.1	42.1	7.4	4.6	3.9	5.3	5.5
	16KB	CFA_8	9.2	13.8	12.2	1.3	34.0	6.2	6.5	5.4	6.7	8.2
	32KB	Base	6.7	13.7	11.3	8.2	34.7	7.4	4.7	4.0	5.4	5.7
	32KB	CFA_8	8.6	13.8	12.3	1.3	27.7	6.2	6.5	5.4	6.8	8.2
postgres	0KB	Base	7.5	—	7.5	9.3	—	3.0	4.1	3.9	5.2	4.3
	0KB	CFA_{16}	10.3	—	10.3	1.0	—	8.4	5.7	4.8	5.9	7.2
	16KB	Base	7.2	14.2	10.6	10.0	51.8	3.0	5.3	5.0	6.8	5.7
	16KB	CFA_{16}	9.7	14.0	11.9	1.1	49.6	8.4	6.2	5.1	6.4	7.9
	32KB	Base	7.0	14.2	11.6	9.9	35.7	3.0	5.8	5.4	7.4	6.2
	32KB	CFA_{16}	8.9	14.1	12.3	1.2	33.7	8.4	6.3	5.2	6.5	8.1
xblast	0KB	Base	7.1	—	7.1	4.8	—	7.7	3.9	3.4	4.4	4.5
	0KB	CFA_8	10.1	—	10.1	0.6	—	7.9	5.7	4.8	5.8	7.2
	16KB	Base	7.0	13.7	10.8	6.8	43.4	7.7	4.9	4.2	5.6	5.9
	16KB	CFA_8	9.2	13.7	11.5	1.5	48.7	8.8	5.8	4.8	6.0	7.4
	32KB	Base	6.8	13.7	11.1	7.2	37.9	7.9	4.9	4.2	5.6	5.9
	32KB	CFA_8	8.6	13.8	12.1	0.9	33.1	7.8	6.4	5.2	6.5	8.2

Table 5: Simulation results for a 32KB instruction cache.

6 Related Work

There has been much work on code mapping algorithms to optimize the instruction cache miss rate. These works were targeted at less aggressive processors, which do not need to fetch instructions from multiple basic blocks per cycle.

Hwu and Chang [7] use function inline expansion, and group into traces those basic blocks which tend to execute in sequence as observed on a profile of the code. Then, they map these traces in the cache so that the functions which are executed close to each other are placed in the same page.

Pettis & Hansen [10] propose a profile based technique to reorder the procedures in a program, and the basic blocks within each procedure. Their aim is to minimize the conflicts between the most frequently used functions, placing functions which reference each other close in memory. They also reorder the basic blocks in a procedure, moving unused basic blocks to the bottom of the function code, even splitting the procedures in two, and moving away the unused basic blocks.

Torrellas *et al* [13] designed a basic block reordering algorithm for Operating System code, running on a very conservative vector processor. They map the code in the form of sequences of basic blocks spanning several functions, and keep a section of the cache address space reserved for the most frequently referenced basic blocks. A comparison between the STC, the Pettis & Hansen method and the Torrellas *et al* method can be found in [11].

Gloy *et al.* [5] extend the Pettis & Hansen placement algorithm at the procedure level to consider the temporal relationship between procedures in addition to the target cache information and the size of each procedure. Hashemi *et al* [6] and Kalamaitianos *et al* [8] use a cache line coloring algorithm inspired in the register coloring technique to map procedures so that the resulting number of conflicts is minimized.

Techniques developed for VLIW processors, like Trace Scheduling [3] also identify the most frequent execution paths in a program. But these techniques are trying to optimize the scheduling of instructions in the execution core of the processor, not the performance of the instruction fetch engine. Individual instructions are moved up and down, crossing the basic block boundary, to optimize ILP in the execution core of the processor, inserting *compensation code* to undo what wrongly placed instructions did when the wrong path is taken. The traces they define are logical, the basic blocks need not be actually moved in order to obtain the desired effect. In that sense, these techniques and the STC may be complementary, one optimizes instruction fetch while the other optimizes instruction scheduling, both using the same profile information.

On the hardware side, techniques like the Branch Address Cache [14], the Collapsing Buffer [2] and the Trace Cache [4, 12] approach the problem of fetching multiple, non-contiguous basic blocks each cycle. The Branch Address Cache and the Collapsing Buffer access non-consecutive cache lines from an interleaved i-cache each cycle and then merge the required instructions from each accessed line. The Trace Cache does not require fetching of non-consecutive basic blocks from the i-cache as it stores dynamically constructed sequences of basic blocks in a special purpose cache. These techniques require hardware extensions of the fetch unit, and do not target an i-cache miss rate reduction, relying on other techniques for it.

Some other works have examined the interaction of run-time and compile-time techniques regarding the instruction fetch mechanism. Chen *et al.* [1] examined the effect of the code expanding optimizations (loop unrolling and function inlining) on the instruction cache miss rate. Also, as an example of software and hardware cooperation, Patel *et al* [9] identify branches with a fixed behavior and avoid making prediction on them, increasing the potential of the Trace Cache.

7 Conclusions

In this paper we present a profile based code reordering technique which targets an optimization of the instruction fetch performance in the more aggressive wide superscalar processors.

By carefully mapping the basic blocks in a program we can store the more frequently executed traces in memory, using the instruction cache as a *Software Trace Cache* (STC), obtaining better performance of a sequential fetch unit, and complementing the Hardware Trace Cache (HTC) mechanism with a better failsafe mechanism.

Our results show that for large codes with few loops and deterministic execution sequences, like database applications, the STC can offer similar, or better, results than the HTC alone. However, optimum results come from the combination of both the software and the hardware approaches. The number of fetched instructions per cycle obtained with a combination of the STC and a small trace cache is comparable to that of a HTC of double size alone.

The storage of the most popular traces in the instruction cache leads to a new view of the fetch unit, where the trace cache is more tightly coupled with the contents of the instruction cache. Some traces are being redundantly stored in both caches, effectively wasting space, and displacing potentially useful traces from the trace cache. It is yet another example of the need for the software and the hardware to work together in order to obtain optimum performance with the minimum cost.

References

[1] W. Y. Chen, P. P. Chung, T. M. Conte, and W.-M. Hwu. The effect of code expanding optimizations on instruction cache design. *IEEE Transactions on Computers*, 42(9):1045–1057, Sept. 1993.

[2] T. Conte, K. Menezes, P. Mills, and B. Patell. Optimization of instruction fetch mechanism for high issue rates. *Proceedings of the 22th Annual Intl. Symposium on Computer Architecture*, pages 333–344, June 1995.

[3] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, 30(7):478–490, July 1981.

[4] D. H. Friendly, S. J. Patel, and Y. N. Patt. Alternative fetch and issue techniques from the trace cache mechanism. *Proceedings of the 30th Anual ACM/IEEE Intl. Symposium on Microarchitecture*, Dec. 1997.

[5] N. Gloy, T. Blackwell, M. D. Smith, and B. Calder. Procedure placement using temporal ordering information. *Proceedings of the 30th Anual ACM/IEEE Intl. Symposium on Microarchitecture*, pages 303–313, Dec. 1997.

[6] A. H. Hashemi, D. R. Kaeli, and B. Calder. Efficient procedure mapping using cache line coloring. *Proc. ACM SIGPLAN'97 Conf. on Programming Languaje Design and Implementation*, pages 171–182, June 1997.

[7] W.-M. Hwu and P. P. Chang. Achieving high instruction cache performance with an optimizing compiler. *Proceedings of the 16th Annual Intl. Symposium on Computer Architecture*, pages 242–251, June 1989.

[8] J. Kalamaitianos and D. R. Kaeli. Temporal-based procedure reordering for improved instruction cache performance. *Proceedings of the 4th Intl. Conference on High Performance Computer Architecture*, Feb. 1998.

[9] S. J. Patel, M. Evers, and Y. N. Patt. Improving trace cache effectiveness with branch promotion and trace packing. *Proceedings of the 25th Annual Intl. Symposium on Computer Architecture*, pages 262–271, June 1998.

[10] K. Pettis and R. C. Hansen. Profile guided code positioning. *Proc. ACM SIGPLAN'99 Conf. on Programming Languaje Design and Implementation*, pages 16–27, June 1990.

[11] A. Ramírez, J. L. Larriba-Pey, C. Navarro, X. Serrano, J. Torrellas, and M. Valero. Code reordering of decision support systems for optimized instruction fetch. Technical Report UPC-DAC-1998-56, Universitat Politecnica de Catalunya, Dec. 1998.

[12] E. Rottenberg, S. Benett, and J. E. Smith. Trace cache: a low latency aprroach to high bandwith instruction fetching. *Proceedings of the 29th Anual ACM/IEEE Intl. Symposium on Microarchitecture*, pages 24–34, Dec. 1996.

[13] J. Torrellas, C. Xia, and R. Daigle. Optimizing instruction cache performance for operating system intensive workloads. *Proceedings of the 1st Intl. Conference on High Performance Computer Architecture*, pages 360–369, Jan. 1995.

[14] T. Y. Yeh, D. T. Marr, and Y. N. Patt. Increasing the instruction fetch rate via multiple branch prediction and a branch address cache. *7th Intl. Conference on Supercomputing*, pages 67–76, July 1993.