The Instruction Register File

Bernard Goossens[1]

LIAFA, Université Paris 7, 175 rue du Chevaleret, 75013 Paris, France
bg@liafa.jussieu.fr

Abstract. We present the Instruction Register File (IRF) coupled with a basic block translator, aiming to deliver a high instruction fetch rate. The IRF has one write port to load instruction cache blocks into registers. It also has p read ports to fetch up to p basic blocks per cycle from up to p registers. The translator predicts up to p on-path basic blocks per cycle and translates their start address into an IRF reference. The references are used in the fetch stage to read the registers and the basic blocks limits serve to merge the accessed registers into a dynamically predicted trace line. The IRF coupled with basic block descriptor tables avoid the need to cache traces as in the trace cache micro-architecture. Moreover, the IRF places the instruction memory hierarchy out of the cycle determining path, as does the data register file with the data memory hierarchy. The IRF performance is estimated with a SimpleScalar based simulator run on the Mediabench benchmark suite and compared to the trace cache performance on the same benchmarks. We show that on this benchmark suite, an IRF-based processor fetching up to 3 basic blocks per cycle outperforms a trace-cache-based processor fetching 16 instructions long traces by 25% on the average.

1 Introduction

Fetch bandwidth is an essential issue in actual processors. Two factors tend to limit the processor fetch capacity per cycle:

1. The presence of control flow instructions breaks the sequential nature of fetching.
2. The reduction of the cycle width leads to an increased latency of the fetch path, which increases the mis-predict penalty.

The trace cache [16, 24] is the main micro-architectural technique proposed to overcome the first factor (another micro-architectural technique is multi-path fetch and execution [1]; predicated Instruction Set Architectures [14] and block-structured ISA [9] being the main contributions on the architectural side; branch alignment [4] and trace scheduling [6] are compiler optimizations to improve fetch bandwidth). Even though the trace cache is a recent proposal (1994 for the patent from Peleg and Weiser and 1996 for the main performance estimations from Rotenberg, Bennett and Smith), it has already been integrated to the

Pentium IV [19]. This shows that industry architects consider it as a major innovation in the processor front-end design.

However, the main problem with the trace cache is its redundancy [21], firstly because instructions may be duplicated in the trace cache itself. Some constraints on the trace cache content like forbidding path associativity (paths AB and AC may not reside in the cache simultaneously) and basic blocks splitting tend to diminish the redundancy. Still, the trace cache is redundant with the instruction memory hierarchy.

The block-based trace cache [2] was proposed by Black, Rychlik and Shen to prevent duplications in the trace cache. The basic blocks (in short BBs) instead of the traces are cached. A trace table is constituted, each of its entries binding a set of n BB descriptors. The fetcher predicts a next trace id (next BB start address and the predicted directions of its conditional branches) and gets its constituting BBs limits from the trace table. The referenced BBs are then fetched from the BB cache and merged to form the fetched trace line. By replacing the trace cache with a BB cache, the block-based trace cache micro-architecture decreases the instruction redundancy. However, it moves the fetched trace construction from the dispatch stage back to the fetch stage. Moreover, the BB cache is still redundant with the instruction cache and it may also waste some space due to the different lengths of the BBs.

Concerning the second limitation in the fetch bandwidth (due to the cycle width reduction), some new proposals are arising. The FTB (Fetch Target Buffer) [22] is a multi-level BTB (Branch Target Buffer [17]) decoupled from the fetch path to allow a fast next basic block prediction. However, the instruction cache remains in the fetch critical path and today yet must be pipelined to scale with the cycle (today, a fetch has a two cycles latency). Prefetching techniques have also been proposed to hide instruction cache miss latency [18, 5, 23, 26].

In this paper, we propose to remove both the trace cache and the BB cache to confine instruction redundancy to the Instruction Register File space. The fetcher dynamically constructs the fetched trace line (as in the block-based trace cache micro-architecture) from parallel register file accesses. The register file pointers are translations from BB descriptor tables such as the BBTB proposed in [27]. These translations are enqueued by a prefetcher (like the FTB in [22] which enqueues one BB descriptor per cycle in an FTQ). The IRF-based micro-architecture removes any cache from the fetch path which in the future should turn out to be our main contribution. Instead, the fetch path fetches from a register file of the same size than the data register file. Hence, the instruction path is organized on the model of the data path, with a main access point in the registers and a secondary access point in the hierarchical memory.

The paper is organized as follows: section 2 recalls the trace cache micro-architecture and explains why the IRF design should perform better. Section 3 describes the IRF fetcher and the BB descriptors prefetcher. Section 4 is devoted to the performance measures of the IRF-based micro-architecture, performed on an adapted version of the SimpleScalar [3] simulator applied to the Mediabench [12] benchmark suite. Section 5 compares these results to the performance of the trace cache micro-architecture on the same benchmark suite. Section 6 gives and discusses some hints on the IRF micro-architecture options. Then comes a summary.

2 The Trace Cache Micro-architecture

A trace cache (see figure 1) is composed of a fill unit, a cache of trace lines and a trace path predictor. The fill unit is placed in the dispatch stage of the pipeline (an alternative is to place it in the commit stage; in the former case, speculative traces are cached; in the latter only committed traces are; another alternative is trace preconstruction [11]). While instructions are in-order dispatched to the reservation stations (or any equivalent structure) they also fill a trace line buffer which is cached when full (we will define what $full$ means soon). Multiple BBs may take place in the buffer (we will also give a precise definition of a BB in the context of the trace cache soon). The fetch stage reads the trace line indexed by the program counter (PC) from the trace cache. It cuts the fetched line at the longest prefix according to the trace path predictor outcome and delivers this prefix to the dispatch stage. This is known as "partial matching" [24]. In case of a trace cache miss, the instruction memory hierarchy is requested (in fact, both the trace cache and the L1 cache are accessed; if the former hits, the latter access is cancelled). At the other end of the pipeline at control flow instruction retire time, the processor core may correct a mis-predicted path by sending a correcting address to PC. It also adjusts the trace path predictor according to the control flow instruction computed direction and target.

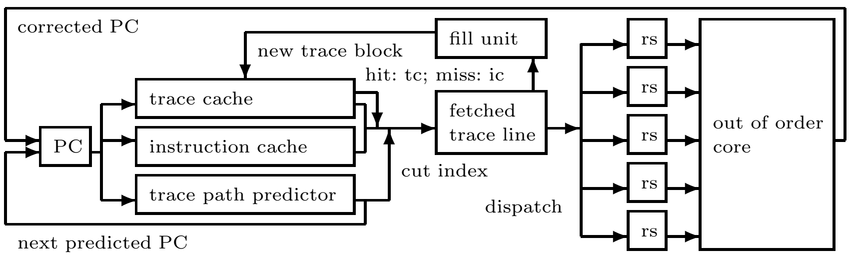

Fig. 1. The trace cache micro-architecture

In the context of the trace cache, a BB is defined by the three following rules:

1. A BB has at most one control flow instruction.
2. If a BB has a control flow instruction, it is the ending one.
3. A BB has at most n instructions (n is the trace cache block size).

The third rule ensures that a BB always fits in a trace cache line. A succession of $kn+r,\ r<n,\ k>0$ instructions with a single ending control flow instruction is splitted into $k+1$ BBs (or k if $r=0$). So, a BB is either ended by a control flow instruction or has length n. It is worth to notice that this definition does not invoke control flow instructions targets. A consequence is that two BBs may share a common suffix (e.g. the BB entering a loop and the BB starting the loop body). Two BBs may also share a common middle part (a first length n one composed of chunks A and B and a second one composed of chunks B and C; B is the starting chunk of a loop body). However, two different BBs may not start with the same prefix.

A trace line is the concatenation of full BBs. No BB may span multiple trace lines. A new BB is integrated by the fill unit to the buffer containing the trace line under construction if all the following conditions hold:

1. The BB fits in the buffer.
2. The buffer does not contain any indirect jump or trap instruction.
3. If the incoming BB is ended by a conditional branch instruction, the buffer contains at most $q-1$ other conditional branch ended BBs.

The first condition (don't cut a BB) serves to limit the redundancy in the trace cache (cutting BBs also increases their total number). The second condition (at most one indirect jump per line) comes from the fact that indirect jumps may be followed in the run trace by different BBs, which would lead to multiple trace lines with a common prefix (assuming the indirect jump ended BB does not start the trace line). To furthermore limit the redundancy in the trace cache, path associativity (two different lines starting with a common BB) is not allowed (a single trace line in the trace cache for a given trace line start address). In the third condition (at most q conditional branches per trace line) the value of q is fixed according to the average number of branches in a full trace (for traces of $n=16$ instructions, $q=3$). Notice that the last conditional branch of the line may be followed by multiple other BBs all ended by unconditional jumps (the last one may be ended by an indirect jump).

The major advantage of the trace cache over the standard instruction cache is that it contains dynamic traces rather than static code. Hence, it is able to deliver multiple BBs fastly to the dispatch stage (a multiported instruction cache can also deliver multiple BBs to the dispatch stage with a multiple-block ahead branch predictor as the one proposed in [25] but not fastly: the multiported cache is big and so has a long latency; moreover, the concatenation of the BBs is a long process because the limits of the BBs are known lately after cache read and BB inspection).

The IRF micro-architecture described in the next section should perform better than the trace cache for the five following reasons:

1. Partial matching does not apply in the IRF context. A BB is predicted and then fetched. In the trace cache micro-architecture, BBs are fetched and possibly predicted to be discarded. Inactive issue [7] which issues and tags the

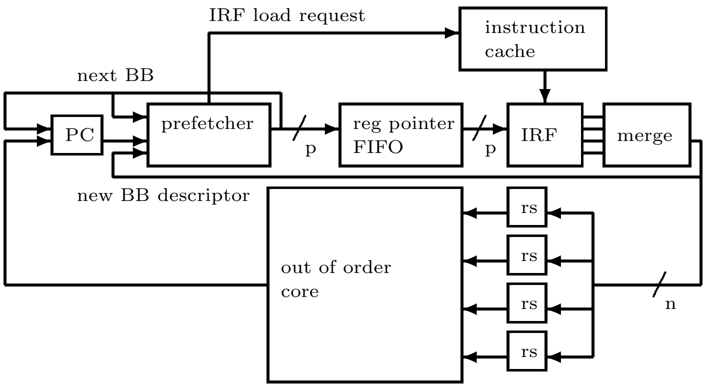

Fig. 2. The IRF micro-architecture

fetched but predicted not on-path BBs (they are committed if the prediction turns out wrong) may partially overcome this loss of fetch bandwidth, but only when the predictor is wrong, which we hope is rare.

2. The IRF is less restrictive than the trace cache concerning path associativity. For example, if path A is ended by a conditional branch and followed either by path B (taken branch path) or by path C (not taken branch path), the trace cache can keep paths AB and C or AC and B but not AB and AC. If the predictor predicts AC and the trace cache holds AB and C, it will fetch A and one cycle later C. The IRF can hold A, B and C in one, two or three registers. If AB is requested, two pointers on A and B allow their fetching in the same cycle. If later, the predicted path is A and C, two new pointers allow the fetching of A and C also in the same cycle.
3. The IRF allows the fetcher to cut an ending BB. The fetcher fetches BBs until its fetch capacity is reached. If the last fetched BB exceeds the fetch capacity, its pointer remains in the queue with an updated start offset and length. In the trace cache design, if a line fetched is not full, the trace cache fetcher has no way to fill it. Moreover, if the prediction does not match the fetched trace, the trace line is cut. Inactive issue may once more partially overcome this fetch bandwidth loss but still only when the predictor is wrong.
4. The IRF fetcher can build trace lines across indirect jump boundaries and the trace cache fetcher cannot.
5. Decoupling the BBs selection from the fetch leads to non blocking IRF load requests. Fetch is idle while the instruction register is loaded but prefetch may go on and send a new load request every cycle. Any design including a BB prefetcher as in the FTB micro-architecture offers the same advantage.

3 The Instruction Register File Machine

Figure 2 is a general view of the IRF micro-architecture. Its components are a prefetcher devoted to BBs descriptors search and translation (at most p descriptors are searched and translated into register references per cycle), a fetcher that reads (up to p) registers from the IRF and merges them, a dispatch stage which dispatches the fetched instructions to the reservation stations (at most n instructions fetched and dispatched per cycle) and an out-of-order core that executes the instructions. The dispatch stage contains a BB descriptor builder. New descriptors are added to the BB descriptor tables held by the prefetcher. In the IRF context the BB definition includes a fourth rule: a BB may not span two registers (its ending instruction is either a control flow one or is followed by a register aligned instruction).

3.1 The Fetcher and the IRF

The fetch path can be built on the model of the data path. The processor core is able to execute multiple instructions per cycle because each gets its data from a register file. Modern ISA are register oriented rather than memory oriented because a register file has the advantage over an L1 data cache that it can be reduced to a few multiported entries. By limiting the number of registers in the file, each instruction can designate at least two sources and one destination. The same structure can be applied to the instruction path to allow multiple BBs fetches by placing an instruction register file between the instruction memory hierarchy and the fetch part of the processor core. In the data register file context, the instruction controls the data register file access with its sources and destination fields. In the instruction register file context, BBs are read from the register file using register pointers prepared by a prefetcher.

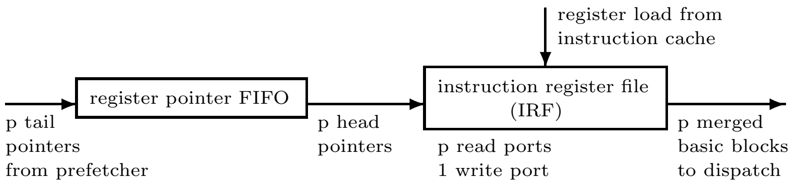

Fig. 3. The instruction register file and the register pointer FIFO

If the register file has p read ports, up to p BBs can be read and merged by the fetcher to form a dynamic trace, provided that p pointers are available (a pointer may be available but point to a loading register; in this case, the fetcher stays idle until the register is loaded). It must be noticed that the merging operation is far much simpler than what is required with a multiported cache and a multi-block fetcher. The BBs need not be inspected to find their ending instruction. Their start address and length are prepared by the prefetcher and given to the fetcher with the register reference. The merging process is just a matter of shifting and multiplexing the register file outputs. If the p BBs cumulated length overtakes the fetch capacity, the BB that has not been fully fetched remains in the pointer FIFO with an adjusted offset and length. In the reverse case, if the p fetched BBs cumulated length is far under the fetch capacity, the fixed number of read ports on the register file does not allow any further adding (in this particular case, the trace cache could perform better: a trace line may contain any number of BBs as long as the maximum of q conditional branches is not exceeded; hopefully for the IRF micro-architecture, as the simulation has shown, the case is rare). Figure 3 depicts the fetcher design.

3.2 The Prefetcher and the BB Descriptor Tables

The second part of the proposed micro-architecture is the prefetcher (see figure 4). It is responsible of the instruction register pointers production. For this purpose, it uses a register allocation table (TRALL). This table keeps for each register of the IRF the aligned address of the instructions it contains. Translating a BB address into a register pointer is a parallel search in this table. If the BB aligned address is found, the hit entry number gives the desired pointer. In case of a miss, a free register is allocated (a register is free if no waiting-to-be-fetched BB refers to it; table entry e contains a field pointing to the newest enqueued waiting-to-be-fetched BB refering to register e; a FIFO head entry h referencing e frees register e if when dequeued by the fetcher, the TRALL table entry e still points to h). If no register is free, prefetching stops. When a free register is allocated, a register load is requested (load from the instruction memory hierarchy). As we assume a single read port L1 instruction cache, we stop prefetching for the cycle (hence, the prefetcher can initiate at most one load per cycle).

The translations wait in a register pointer FIFO until they are consumed by the fetcher. Each FIFO entry contains a register pointer field, an offset field (offset of the BB first instruction in the register), a length field (length of the BB), an address field (base address of the BB; this is useful for the BB instructions executions) and a prediction field (used for conditional branch ended BBs; the prediction made is forwarded to the core to allow its verification).

The BB address the prefetcher translates is obtained from a table search. The search may provide the BB address itself (conditional branch and immediate jump) or a way to get it (return: the return address is given by the return address stack; indirect jump: the target is searched in a BTB). The prefetcher uses three tables of BB descriptors, each table caching descriptors according to the BB ending instruction. There is one table for the conditional branch ended BBs (TCOND), one for the indirect jump ended BBs (TIND; it includes returns) and one for the immediate jump ended BBs (TIMM). The tables are separated because the descriptors they hold are not identical. In every descriptor we find a tag field (the upper part of the BB start address, the lower part being used to address the tables; as the tables have different sizes, the tag field width varies from one table to another). There is also a length field. In the TIND and the TIMM table, we find a target field (for branches, it contains the computed target instead of the displacement; this loses space but accelerates the prefetching process). In the TIMM and TIND tables, we have a link bit (indicating whether the ending jump instruction is a jump and link or not). In the TIND table, we have a return bit (indicating whether the ending jump instruction is a return or not). Each descriptor in the TCOND and TIMM tables should be approximately 8 bytes wide. Each descriptor in the TIND table should be approximately 4 bytes wide (no target address field).

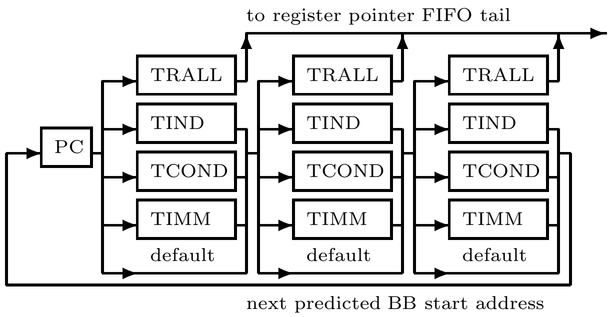

Fig. 4. The prefetcher path with 3 searches per cycle

The current BB start address is the PC. It addresses the three BBs tables and the TRALL table in parallel. It also accesses the conditional branch predictor and the indirect jumps BTB (not represented on figure 4). The current BB is translated and its register pointer translation is enqueued in the register pointer FIFO. The prefetch stops if the queue is full. In parallel with the translation, one of the BBs tables may hit and fix the next BB address. If the TCOND table hits, the branch prediction is used. If the TIND table hits, its return field indicates whether the target address should be obtained from the top of the return address stack or from the BTB. If all the tables miss, the next BB address is the next aligned address (such a miss either comes from a BB that had to be cut not to span two registers or from an unknown BB; in the latter case, the BB may contain a never yet encountered control flow instruction; it will be seen at dispatch time, the sequence of instructions will be cut into two BBs at the control flow boundary and the prefix BB will have its descriptor written in the appropriate BB descriptor table; eventually, at retire time the missed control flow instruction will restart the prefetcher).

If the IRF has p read ports, the fetcher consumes pointers at a peak rate of p per cycle. The prefetcher must produce them at that same rate which means that each table must be accessed p times in the cycle (the last access to the tables in the cycle fixes next PC). If p is high and the cycle is short, it may be a problem for the TCOND table, due to its size (in the simulation described in section 4, we assumed a 2048 entry TCOND table wich represents 16KB of storage; the TIMM and TIND tables were assumed to have 256 entries each which represents 2KB and 1KB of storage). An alternative design is to have multi-descriptors tables as in the block-based trace cache micro-architecture (a table gives multiple BB addresses in a single access; the addresses can then be translated in parallel). The table content relies on a branch prediction. Rakvic, Black and Shen [20] have proposed a way to build it at completion time. However, binding multiple BBs together has bad side effects such as wrong path prefetching and descriptor redundancy. Another option is to have a two level TCOND table as in the FTB design. We do not further investigate such design options in this paper to focus on the IRF design and performance.

Figure 4 depicts the prefetcher. The simulated micro-architecture assumed 3 read ports on the IRF ($p = 3$). On the figure, the different tables are assumed to have 3 identical copies each. Different implementations options are possible (among which we can mention multi-ported tables or wave-pipelined accesses [8]) but are not furthermore investigated in the paper.

4 The IRF Micro-architecture Performance

Table 1. The Mediabench suite

program	*run size*	*IPB*
rawcaudio	6,618,178	3.7357
rawdaudio	5,459,855	3.3642
epic	52,901,786	6.7474
unepic	6,988,923	4.7280
cjpeg	15,510,101	6.3729
djpeg	4,649,119	19.6027
pegwitencode	32,157,921	8.5496
pegwitdecode	18,182,963	9.0270
g721encode	275,165,274	4.3445
g721decode	268,411,763	4.3072
gs	500,000,000	4.6841
mipmap	68,415,518	6.2431
osdemo	22,712,467	5.4228
texgen	99,441,965	7.5833
toast	234,070,244	20.4992
untoast	73,621,328	5.5866
mpeg2encode	500,000,000	5.7330
mpeg2decode	171,225,367	8.4695
average		7.50

In order to measure both the fetch bandwidth and the IPC performance of an IRF micro-architecture, we have adapted the SimpleScalar simulator to an IRF-based fetcher and a BB descriptor prefetcher and translator. The simulator was run on the Mediabench benchmark suite. Table 1 gives the main characteristics of the programs in the Mediabench suite (mainly the number of instructions in a full run, limited to the 500 millions first instructions and the number of instructions per branch (IPB)).

Table 2. Performance of the IRF micro-architecture

program	*IRF PPC*	*IRF IPC*
rawcaudio	9.7320	1.7403
rawdaudio	9.5707	2.6249
epic	15.5377	6.6439
unepic	8.3403	3.6559
cjpeg	12.4742	5.0975
djpeg	15.1504	7.3157
pegwitencode	16.4982	3.5058
pegwitdecode	16.8531	3.7092
g721encode	12.1354	3.7108
g721decode	11.7895	4.1403
gs	9.6364	3.6196
mipmap	15.3304	5.8612
osdemo	15.5832	8.2981
texgen	19.4091	6.3602
toast	12.2308	7.5460
untoast	11.3206	6.7441
mpeg2encode	14.6591	2.5157
mpeg2decode	23.9409	6.1446
average	13.8996	4.9574

The IRF micro-architecture design parameters for the simulation were fixed as follows: The IRF register file has 8 registers of 32 instructions each, 3 read ports and one write port. The maximum fetch length is 16 instructions. In the prefetcher, the register pointer FIFO has 16 entries. The TIND and the TIMM tables have 256 entries each (1KB and 2KB of storage). The TCOND table has 2048 entries (16KB of storage). The return stack has 64 entries. The indirect jumps BTB has 256 entries (2KB). The branch predictor is a bimodal one with 4096*2 bits counters (1KB). As the IRF removes the instruction cache from the fetch path, we have modeled a big and long latency L1 instruction cache (as in the HP PA8500 [10]). It has a capacity of 512KB and a 3 cycles latency. Because each register holds 32 instructions (twice the fetch peak bandwidth), to allow a full load in one shot, each L1 cache line holds 128 bytes. The associativity is 4. The L1 data cache has a capacity of 64KB, with blocks of 64 bytes and an associativity degree of 4. The latency is 2 cycles. The L2 cache is unified, 1MB, 128 bytes blocks, associativity 4 and a latency of 8 cycles. The memory has a first word access time of 50 cycles and a following words access time of 2 cycles. The bus width is 16 bytes. There is a 16 entries ITLB and a 32 entries DTLB both with an 8 cycles access latency.

Table 3. Miss rates of the IRF micro-architecture

program	*IRF miss*	*b.pred. miss*
rawcaudio	0.04%	29.34%
rawdaudio	0.10%	19.53%
epic	0.87%	4.38%
unepic	0.41%	6.97%
cjpeg	5.41%	7.24%
djpeg	4.71%	8.64%
pegwitencode	7.01%	14.17%
pegwitdecode	6.74%	14.27%
g721encode	5.10%	10.27%
g721decode	4.88%	8.45%
gs	15.03%	5.09%
mipmap	26.18%	1.25%
osdemo	8.32%	1.73%
texgen	18.28%	5.51%
toast	13.66%	7.30%
untoast	1.55%	2.42%
mpeg2encode	0.48%	24.34%
mpeg2decode	2.60%	10.04%
average	6.74%	10.05%

The size of the IRF (8*32*4 bytes, 3 read ports, 1 write port) has been choosen to keep its die occupation area close to a standard 32*8 bytes data register file with 8 read ports and 4 write ports.

Table 2 gives the measured PPC (instructions prefetched per cycle; this is the sum of all prefetched BBs lengths divided by the number of cycles) and IPC. The PPC value can be greater than the maximum fetch length. However, at most 16 instructions are fetched, the last BB being cut by the fetcher and fetched in two cycles.

Table 3 gives the IRF miss rate and the branch predictor miss rate. As the storage devoted to the IRF and to the branch predictor is limited by the cycle width, the miss rates are sometimes high.

5 Comparison with the Trace Cache Micro-architecture Performance

We have derived a second simulator from the SimpleScalar tool set implementing a baseline trace cache (i.e. not including enhancements such as trace packing and

Table 4. Performance of the trace cache

program	*tc IPC*	*improvement*
rawcaudio	1.8148	-4.11%
rawdaudio	2.5017	4.92%
epic	5.6007	18.63%
unepic	2.9247	25.00%
cjpeg	4.2303	20.50%
djpeg	6.6510	9.99%
pegwitencode	3.0784	13.88%
pegwitdecode	3.2759	13.23%
g721encode	2.7669	34.11%
g721decode	2.8850	43.51%
gs	3.1903	13.46%
mipmap	3.4519	69.80%
osdemo	2.7875	197.69%
texgen	4.0244	58.04%
toast	7.2792	3.67%
untoast	6.7687	-0.36%
mpeg2encode	2.4262	3.69%
mpeg2decode	5.3741	14.34%
average	3.9462	25.62%

branch promotion [15] nor inactive issue [7]). The simulator was run on the same benchmarks. Because the trace cache also removes the instruction cache from the fetch path (but it replaces it by the trace cache itself), we have modeled the same big and long latency instruction cache than for the IRF micro-architecture. The only difference was in the block length fixed to 64 bytes in the trace cache design. The branch predictor is a two level predictor [13] accessed at most three times per cycle (this should give an optimistic hit rate compared to the accuracy of a true trace path predictor) with 32K*2 bits counters at the second level and 32K*15 bits indexes at the first level. All other parameters are the same than the ones choosen for the IRF micro-architecture.

Table 4 shows the performance of the trace cache design and the improvement of the IRF micro-architecture over the trace cache one. The IRF design outperforms the trace cache design except in two cases ("rawcaudio" and "untoast"), even though its predictor is smaller and the fetch is limited to 3 BBs by the number of read ports on the IRF (the trace cache micro-architecture can fetch more than 3 BBs every time a fourth BB is not ended by a conditional branch and fits in the trace line). The maximum improvement (close to 200%) is obtained with "osdemo". This example is a very good illustration of what can go wrong in the trace cache: most of the time, the predictor predicts that the first conditional branch in the fetched trace line should go in a direction opposite to the one traced. Whatever the hit rate, the trace has to be cut after a short prefix. Maybe inactive issue could have some impact on "osdemo" performance. We did not measure this. A good score is also obtained with "mipmap" (70%)

Table 5. Variation of the register file cut out

program	*IPC 4*	*IPC 16*
rawcaudio	1.7397	1.7326
rawdaudio	2.6298	2.5629
epic	6.6385	6.2655
unepic	3.6310	3.5974
cjpeg	5.0869	4.9964
djpeg	7.1764	7.4095
pegwitencode	3.5207	3.5149
pegwitdecode	3.7254	3.6976
g721encode	3.7524	3.6659
g721decode	4.1764	4.0654
gs	3.6788	3.3631
mipmap	5.8857	4.7360
osdemo	8.3626	8.2349
texgen	6.5506	5.9858
toast	7.5583	7.5346
untoast	6.7125	6.7515
mpeg2encode	2.5196	2.4948
mpeg2decode	6.0914	6.0823
average	4.9687	4.8162

and "texgen" (58%). In both cases, the IRF miss rate is high (this explains that the gain is not as good as in "osdemo") but the branch predictor miss rate is low for "texgen" and very low for "mipmap". These examples show all the potential of the IRF design if coupled with an accurate branch predictor. At the other end, we find "rawcaudio" and "untoast" for which the trace cache performs a little better than the IRF (4%). In "rawcaudio", it comes from a bad IRF performance due to a very high mis-prediction rate (close to 30%). The reason is clear when we see that the IPB is only 3.74. For "rawdaudio" this time it is the IRF that performs 4% better than the trace cache. The IPB is also very low (3.36 instructions per branch). However the predictor miss rate is lower (20%) which explains why the IRF has a better score. For "untoast" even though the miss rates are very low, the trace cache has a higher IPC. It comes from a good performance of the trace cache as shows the fact that it is the only case for which the IPC is greater than the IPB (we run more than one basic block per cycle). In the case of "djpeg" and "toast", the IPB is very high (20 instructions per branch). In such a condition, fetching multiple BBs does not give any advantage. The two architectures are quite close, with a slight advantage to the IRF (10% and 4%).

6 Variations of the IRF Design

6.1 Varying the Register File Parameters

Table 5 shows the variation of the IPC with different number of registers for the IRF. In this experiment, we change the register length proportionally in order to keep a constant IRF size (4 registers of 64 instructions and 16 registers of 16 instructions). We can see that the baseline 8 registers IRF design and the 4 registers file are very close (less than 0.3% difference). The 16 registers file is not as good (3% behind) because each register contains no more instruction than the fetch maximum width. Recall that a BB may not span two registers. This experiment shows that it is better to have wide registers than many.

Table 6 shows the variation of the IPC with different number of registers in the IRF (4 registers and 16 registers) and a constant register length of 32 instructions (the size of the IRF is varied). The experiment shows that 4 registers give the same performance than 8. It also shows that a 16 registers IRF is not cost effective, giving an IPC very close to the one obtained with 4 registers.

Table 6. Variation of the number of registers

program	*IPC 4*	*IPC 16*
rawcaudio	1.7398	1.7409
rawdaudio	2.6228	2.6250
epic	6.6700	6.6753
unepic	3.6489	3.6585
cjpeg	5.0897	5.1022
djpeg	7.2645	7.3673
pegwitencode	3.5073	3.5027
pegwitdecode	3.7116	3.6998
g721encode	3.7109	3.7109
g721decode	4.1402	4.1404
gs	3.6191	3.6199
mipmap	5.8575	5.8645
osdemo	8.2796	8.3133
texgen	6.3552	6.3627
toast	7.5451	7.5465
untoast	6.7427	6.7515
mpeg2encode	2.5156	2.5158
mpeg2decode	6.1436	6.1450
average	4.9536	4.9635

Table 7 shows the IPC when we vary the number of BB descriptors prefetched (2 or 4 per cycle; the number of read ports varies accordingly). The experiment clearly shows that increasing the number of BBs is not cost effective: it should increase the cycle width and the IPC does not get better. Of course, this is mainly due to the branch predictor accuracy. If improved, the IPC could also improve. On the other side, decreasing the number of BBs from 3 to 2 reduces the performance of less than 7%). It could be an alternative if the cycle width is still a problem to perform 3 BBs translations in a cycle (the IRF design still outperforms the trace cache of 19%).

Table 7. Variation of the number of read ports

program	*IPC 2*	*IPC 4*
rawcaudio	1.6835	1.7766
rawdaudio	2.4143	2.7070
epic	5.9511	6.5090
unepic	3.4757	3.5977
cjpeg	4.6668	5.1155
djpeg	7.2522	7.2734
pegwitencode	3.4423	3.5081
pegwitdecode	3.6176	3.7209
g721encode	3.5036	3.7992
g721decode	3.8675	4.2665
gs	3.3859	3.7167
mipmap	5.3869	5.6694
osdemo	7.1942	7.4454
texgen	5.8945	6.2776
toast	7.4022	7.5668
untoast	6.6037	6.7495
mpeg2encode	2.4150	2.5348
mpeg2decode	5.8224	6.1290
average	4.6655	4.9091

6.2 Varying the L1 Instruction Cache Parameters

Table 8 shows the IPC in the case of a small low latency L1 instruction cache (8KB, direct-mapped, 128 bytes lines, latency 1 cycle) and an average size and latency one (64KB, associativity 2, 128 bytes lines, latency 2 cycles). It shows that the latency is less important than the miss rate, hence suggesting a big L1 instruction cache.

7 Summary

In this paper we have presented the IRF micro-architecture. It is composed of an instruction register file which is the first instruction storage on the fetch path. Its multi-ported organization and the basic blocks limits delivered by the prefetcher allow a fast multi-block fetching and merging which provides a higher fetch rate than in the trace cache micro-architecture. Moreover, the IRF does not require the redundant storage of the traces or even the basic blocks. The IRF also removes any cache from the fetch path, allowing fetching to scale with the cycle reduction without requiring new fetch pipeline stages. We have shown that on the Mediabench benchmark suite, the IRF micro-architecture when parameterized to fetch up to 3 basic blocks per cycle outperforms a 16 instructions per cycle trace cache based micro-architecture of an average 25%.

Table 8. Variations of the L1 instruction cache parameters

program	*IPC 8KB*	*IPC 64KB*
rawcaudio	1.7753	1.7766
rawdaudio	2.7061	2.7070
epic	6.4975	6.5072
unepic	3.5815	3.5958
cjpeg	4.8818	5.1123
djpeg	7.0541	7.2448
pegwitencode	3.4520	3.5118
pegwitdecode	3.6703	3.7164
g721encode	3.6540	3.7992
g721decode	3.7768	4.2665
gs	3.0900	3.6983
mipmap	4.3079	5.6608
osdemo	7.0019	7.3827
texgen	5.3739	6.1909
toast	7.4583	7.5615
untoast	6.6648	6.7477
mpeg2encode	2.5251	2.5345
mpeg2decode	6.0109	6.1281
average	4.6379	4.8968

In this design, the prefetch path should be the most critical one because it requires multiple sequential accesses to basic block descriptor tables and control flow predictors. We have suggested various design options to allow up to 3 descriptor translations within a cycle, among which we find a hierarchical organization of the tables as in the FTB micro-architecture, a multiple descriptors binding as in the block-based trace cache or limiting the prefetch to 2 BBs per cycle which gives an IPC only 7% less than the IPC obtained when fetching up to 3 BBs. This is still 19% better than the trace cache performance.

A future work is to investigate the prefetcher different design options mentioned in the paper (e.g. hierarchical descriptor tables and predictors or completion time binding of BBs descriptors) in relation with the cycle width. In particular for the experiments reported in this paper, we have placed the IRF and the trace cache on the same level concerning pipeline latency. In fact, because with the IRF no cache is anymore accessed in the critical path, the pipeline should be shorter than in a trace cache design in which accessing the cache should require at least 2 cycles today and probably more tomorrow.

References

1. P.S. Ahuja, K. Skadron, M. Martonosi, D.W. Clark: Multipath execution: opportunities and limits. ICS12, 1998.
2. B. Black, B. Rychlik, J.P. Shen: The block-based trace cache. ISCA26, 1999.
3. D. Burger, T.M. Austin: The SimpleScalar tool set, version 2.0. Technical report 1342, University of Wisconsin-Madison, june 1997.
4. B. Calder and D. Grunwald: Reducing branch costs via branch alignment. ASPLOS6, 1994.
5. I.K. Chen, C.C. Lee and T.N. Mudge: Instruction prefetching using branch prediction information. ICCD'97, 1997.
6. J.A. Fisher: Trace scheduling: a technique for global microcode compaction. IEEE Trans. on Computers, C30(7):478-490, 1981.
7. D.H. Friendly, S.J. Patel, Y.N. Patt: Alternative fetch and issue policies for the trace cache fetch mechanism. Micro30, 1997.
8. C.T. Gray, W. Liu and R.K. Cain III: Wave pipelining: theory and CMOS implementation. Kluwer academic publishers, Norwell, 1993.
9. E. Hao, P. Chang, M. Evers and Y. Patt: Increasing the prediction fetch rate via block-structured instruction set architectures. Micro 29, 1996.
10. ftp://www.hotchips.org/pub/hot7to11cd/hc98/ pdf_1up/hc98_1a_johnson_1up.pdf
11. Q. Jacobson and J.E. Smith: Trace preconstruction. ISCA27, 2000.
12. C. Lee, M. Potkonjak, W.H. Mangione-Smith: Mediabench: a tool for evaluating and synthetizing multimedia and communications systems. Micro30, 1997.
13. S. Mc Farling: Combining branch predictors. Technical report TN-36, DEC-WRL june 1993.
14. S.A Mahlke et al.: Characterizing the impact of predicated execution on branch prediction. Micro27, 1994.
15. S.J Patel, M. Evers, Y.N. Patt: Improving trace cache effectiveness with branch promotion and trace packing. ISCA25, 1998.
16. A. Peleg, U. Weiser: Dynamic flow instruction cache memory organized around trace segments independent of virtual address line. U.S. patent 5-381-533, 1994.
17. C.H. Perleberg and A.J. Smith: Branch target buffer design and optimization. IEEE Trans. on Computers, 42(4):396-412, 1993.
18. J. Pierce and T. Mudge: Wrong-path instruction prefetching. Micro29, 1996.
19. ftp://download.intel.com/pentium4/ download/netburstdetail.pdf
20. R. Rakvic, B. Black and J.P. Shen: Completion time multiple branch prediction for enhancing trace cache performance. ISCA27, 2000.
21. A. Ramirez, J. Larriba-Rey, M. Valero: Trace cache redundancy: red and blue traces. HPCA6, 2000.
22. G. Reinmann, T. Austin, B. Calder: A scalable front-end architecture for fast instruction delivery. ISCA26, 1999.
23. G. Reinmann, B. Calder, T. Austin: Fetch directed instruction prefetching. Micro32, 1999.

24. E. Rotenberg, S. Bennett, J.E. Smith: Trace cache: a low latency approach to high bandwidth instruction fetching. Micro29, 1996.
25. A. Seznec, S. Jourdan, P. Sainrat, P. Michaud: Multiple-block ahead branch predictors. ASPLOS7, 1996.
26. A. Veidenbaum, Q. Zhao, A. Shameer: Non sequential instruction cache prefetching for multiple-issue processors. Int. Journal of Highspeed Computing, 10(1):115-140, 1999.
27. T. Yeh and Y. Patt: A comprehensive instruction fetch mechanism for a processor supporting speculative execution. Micro25, 1992.