Using Trace Cache in SMT

(CSE 241 Final Project Report)

Huaxia Xia hxia@cs.ucsd.edu

06/10/2001

Abstract

SMT (Simultaneous Multithreading) processor has the ability to issue multiple instructions from multiple independent threads in one cycle. Fetch bandwidth has been demonstrated to be one of the main performance bottlenecks for SMT processor. Trace cache is an effective technique for fetching instructions. In this paper, we implement trace cache in SMT and find its effect for SMT performance. Also, we try to discuss about several different implementations of trace cache.

1. Introduction

Simultaneous Multithreading processor issues multiple instructions from independent threads each cycle. This technique allows multiple threads dynamically sharing the function units so that SMT processor can achieve much higher throughput than either a wide superscalar or a multithreaded processor [1]. One of the main bottlenecks for SMT performance is the fetch bandwidth. Conventional instruction prefetching cannot satisfy the increasing requirement for instruction fetch bandwidth.

Trace Cache has been developed for high bandwidth instruction fetching [2] [3] [4]. It places the logically continuous instructions together so that it needn't wait for instructions from multiple cache blocks. It has been demonstrated that trace cache can improve the instruction fetching bandwidth efficiently.

We try to use trace cache in SMT to get better performance. This paper gives the experiment detail. Although the experimental results show no improvement at all, we think that it is due to the implementation limitation. We analyze the results and give some alternative implementation schemes which may have better performance.

2. Previous Work and the Organization of this Paper

This work is based on two research achievements: Tullsen, et al. gave the concept of SMT processor [1] and implemented a simulator for SMT. Our work is made over this simulator. Rotenberg, et al [2] and Patt, et al [3] developed the concept of trace cache. Our implementation of trace cache mainly refers to Patt's model.

In the following sections, we show our implementation detail in section 3. The experiment results are given and discussed in section 4. In section 5, we discussed several alternative implementation schemes. We have no time to implement these schemes but we think they may have better performance. And conclusions are given in section 6.

3. Experimental Detail

3.1 Experiment environment

The SMT simulator from Dean Tullsen is used for our implementation. It runs on Alpha machine and emulates the Alpha ISA.

The benchmarks are four applications: compress, gcc, go and li.

Most tests are based on the configuration of 2048 entries and 2 ways trace cache. We also compare the trace caches with different sizes of 2048, 1024, 512 and 256 entries.

The source codes locate in hxia/smt1. The results are in hxia/bench/out. The file hxia/Huaxia.txt describes the different versions of the binary codes. The last letter of each result file stands for its corresponding binary version (0, 1, 2, 3, 4, 5).

The datapath is as blow:

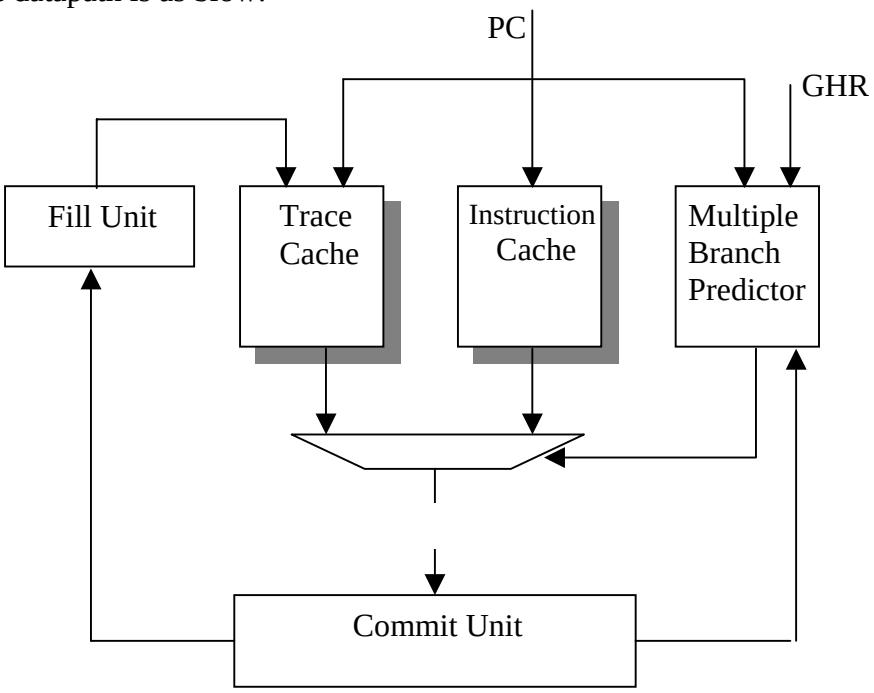

Figure 1. The trace cache datapath

3.2 Data Structure

(1). We define a new type "TraceCache_t", which is one entry of trace cache.

```
#define TRACELINE 16    //don't change this number!
typedef struct TraceCache{
    unsigned char blockcount;  //# of basic blocks in this trace line
                        // if bc=0, then it's not valid.  The branches
                        // may be condition/unconditional branches, or
                        // calles. They cannot be returns, indirect
                        // jumps or traps.
    unsigned char branchpred;     //branch prediction for the branches
    unsigned char blockindex[3];  //index of the basic blocks, bi[0]==0
    unsigned int instrcount;    //# of instructions
    address_t tag;      //the address of the first branch
    address_t addr[3];  //the starting addresses of the basic blocks
    instruction_t instr[TRACELINE];
} TraceCache_t;
```

(2). In the “context”, we add trace cache and multiple predictors as below:

```
#define MP_FIRSTSIZE 0x10000    //64K
#define MP_SECONDSIZE 0x4000    //16K
#define MP_THIRDSIZE 0x2000     //8K
    unsigned char mp1[MP_FIRSTSIZE];
    unsigned char mp2[MP_SECONDSIZE];
    unsigned char mp3[MP_THIRDSIZE];
#define TRACESIZE 2048
#define TRACEWAY  2
    TraceCache_t tc[TRACESIZE];
    TraceCache_t tcFill; // trace line for fill unit.
    TraceCache_t *ptc;//pointer for current trace line. if the trace line
                      // is larger than prefetch buffer, then we cannot put
                      // all the trace line into prefetch buffer.
    int iitl;         // instruction index for the trace line. Index of
                      //  instruction to be copied into decode buffer.
```

Here, the three prediction buffers are 2-bits bimodal prediction patterns. The integer “iitl” is used when the decode buffer cannot hold all the instructions in the trace line.

In each trace line, we need 2 bits for “blockcount”, 3 bits for “branchpred”, 4*3 bits for “blockindex”, 4 bytes for “instrcount”, 16 bytes for “tag & addr”, and 64 bytes for instructions. Thus totally 87 bytes are needed per trace line, and 2048 trace lines need 174KB.

The multiple predictors need 2 *(64K+16K+8K) bits = 22KB.

(3). Each scoreboard entry needs to save corresponding instruction word and branch prediction. The filling for trace cache is done in commit phase. We need to know the information about the instructions such as the instruction word and the branch prediction. In conventional case, we cannot know such information in commit phase, so we save them in scoreboard.

There are 512 scoreboard entries, so the extra storage costs are (33 bits * 512) = 2176 bytes.

3.3 Fill instructions into trace cache

The filling is done in commit phase. Each time an instruction is committed, it is sent to the fill unit. We need deal with the following cases:

(1). We won’t deal with “unpredictable” branches in trace cache.

An “unpredictable” branch is either an indirect jump/call or a return instruction. This kind of instructions has multiple targets so that it is not practical to predict the instruction sequence. Once we meet an unpredictable branch, we will stop filling the current trace line. As for the Alpha ISA, an unpredictable branch is one of the following instructions:

- Indirect jump: JMP
- Indirect call: JSR, JSR_COROUTINE
- RET

All the other branch instructions are “predictable”. Each of them has at most two branch targets so that it’s potentially easy to predict. These instructions include:

- Unconditional short branch: BR, BSR, CALL_PAL
- Conditional float branch: FBEQ, FBLT, FBLE, FBNE, FBGE, FBGT

•Conditional integer branch: BLBC, BEQ, BLT, BLE, BLBS, BNE, BGE, BGT

(2). A trace line won't start from a non-branch instruction.

In fact, a trace line can only start from predictable branches. In Rotenberg's paper[2], their trace line can start from any instruction, either branch or non-branch. We'll discuss this problem in section 5.5.

(3). An alternative method for filling is to do it right after the fetch phase. Fetch-and-fill has the advantage of lower cost, because it needn't save the instructions and the branch predictions in scoreboard. Nevertheless, we choose the scheme of "commit-and-fill" because intuitively it gives more accurate instruction sequence.

3.4 Update the multiple branch predictors

This is also done in commit phase. We use 2-bit GSHARE predictors. The three level prediction buffer has 64K, 16K and 8K entries respectively, just like those in Friendly's paper [3].

A better alternative multiple-branch predictor is given in Yeh's paper [4]. They use only one global prediction table. The second and the third prediction are made using the lower bits of GHR combined with the most recent predicted branch. The following graph is from Yeh's paper.

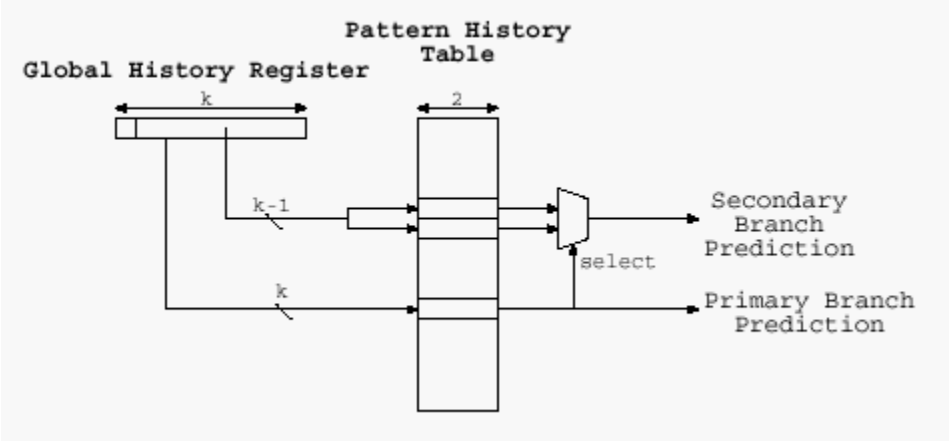

Figure 2. Multiple branch predictor

The new method uses less storage space and gives better prediction. But we have no enough time to change our old predictor.

3.5 Fetch instructions from trace cache

In the simulator, we first check the trace table, if get instructions from trace cache, then decode them one by one; otherwise, try to get instructions from instruction cache. Although in real hardware these two actions should be done in parallel, but it is not necessary for simulation.

When sending the instructions to decoder, we must check the branch information generated for the last instruction. Because the decoder will analyze the instruction, assign reservation resources to it, and generate some branch information, which may be inconsistent with the instruction sequence in trace line. For example, if the decoder predicts a branch to be not taken, but the next instruction in trace line is the taken

target, we must either modify the information for the branch or discard the next instruction. In the implementation, we take the later choice because we don't real know what the branch information contains.

4. Experiment Results

The results is listed below:

4.1 Four threads

Benchmarks: compress, gcc, go and li run together.
Parameters: 2048 entries, 2 way, gshare, branch only.

	Useful fetch	Total fetch	IPC	Trace cache hits
Without trace cache	1200000032	1830783987	4.034	--
With trace cache	1200000048	1859447487	4.072	42375948

The performance has about 1% improvement. The trace cache hits 42375948 times, the hits rate is 2.1%, which means one hit per 48 instructions fetched.

4.2 One thread

Parameters: 2048 entries, 2 way, gshare, branch only.

Program	Trace cache	Useful fetch	Total fetch	IPC	Trace cache hits	Improve
Compress	Without	300000039	808882946	1.654	--	0
	With	300000039	808882946	1.654	0	
Gcc	Without	300000070	675784346	1.701	--	0.8%
	With	300000070	684286577	1.715	24505871	
Go	Without	300000015	790511159	1.783	--	0.06%
	With	300000015	798551908	1.784	30140556	
Li	Without	300000071	472127893	2.684	--	1.5%
	With	300000073	474993641	2.724	21029956	

The performance has about 1% improvement on average. The trace cache hits rate range from 3.5% to 4.4%.

The exception is Compress, there is no trace hit for it at all. The reason might be because Compress contains almost no simple loop and almost all the branches seem to be randomly taken or not taken. In this case, GSHARE cannot work well, but GLOBAL should work very well.

4.3 Compare the effect of trace cache size

Trace Cache	Useful fetch	Total fetch	IPC	Trace cache hits	Improve
No	1200000032	1830783987	4.034	--	--
256 entries	1200000049	1856069436	4.068	34827861	0.84%
512 entries	1200000064	1856916914	4.070	37604562	0.89%
1024 entries	1200000097	1858297062	4.073	40465452	0.97%
2048 entries*	1200000048*	1859447487*	4.072*	42375948*	0.96%*

*Note: In the test for 2048 entries, we didn't change the "TOTALFETCHLIMIT" in fetch.c; while in the tests for 256, 512, 1024 entries, we ignored the "TOTALFETCHLIMIT" when fetch instructions from trace cache. The test for 2048 entries without the limit is still running when this paper is written.

4.4 Discussion about the results

From (1) and (2), we can hardly compare the impacts of trace cache on single thread and multiple threads.

In general, the improvement is quite small. The possible reasons might be:

- The SMT simulator has got almost enough fetch bandwidth without trace cache.
- Since there is a "TOTALFETCHLIMIT" in SMT simulator. If the previous threads has fetched more than TOTALFETCHLIMIT=8 instructions, then the other threads cannot fetch any more instructions in the current clock cycle. This may decrease the parallelism of multiple threads.
- The trace line can only start from predictable branches rather than common instructions. Thus the trace cache hit rate cannot be high. We cannot expect the low hit rate (2%-4.5%) gives significant performance improvement.

From (3), we find that the performance improvement increases from 0.84% for 256 entries to about 1% for 2048 entries, it is only a small increase. Therefore, a trace cache with 256 entries seems large enough. The same conclusion was also demonstrated in Rotenberg's paper [2].

5. Future Work

Because of the time limit, I haven't done the comparisons of different trace cache implementations. I think that they should also be very interesting.

5.1 Associativity of trace cache

Currently the trace cache is organized in 2-way. If we change the associativity to 4-way or direct map, how will the trace cache confliction change?

5.2 Multiple branch predictor

Currently we use 2-level 2-bit GSHARE predictor with three separate pattern buffers. The alternative implementation schemes may change the buffer size, or change GSHARE to GLOBAL.

Moreover, as we mentioned in section 3.4, the multiple branch predictor can be implemented using a single pattern buffer and the accuracy is expected to be somehow better.

5.3 Partial matches of trace line

The current implementation is based on a full match strategy, i.e., use a trace line only when its starting address and branch predictions all match. But this is not necessary. If the fetch address matches the starting address of a trace and the first few branch predictions, the trace line can be used partly.

5.4 Replacement of trace cache

When the fill buffer is full, the fill unit will copy the trace line from fill buffer to trace cache. If all the trace lines in the corresponding set are used, then we need to decide whether or not to replace and which one to replace. Current implementation will always replace a randomly chosen line.

An alternative method is to do replacement only when branches of the new trace line match (or partial match) the multiple branch predictor. When replacement, the trace line not matching the predictor will be replaced.

5.5 Starting point of trace line

As described in section 3.3, the trace lines start from predictable branches.

Another option is to start from any common instructions. The advantage is that it can remember more history information and can decrease the cache miss rate. For example, if a code segment contains no branch, but its size might be many cache blocks, then the second scheme can hide many cache block misses.

The disadvantage is that it needs much larger trace cache. The first scheme needs at most three times trace cache size compared to thread size, while the second scheme needs at most sixteen times trace cache size compared to thread size. For example, consider about a loop with 16001 (note that $16001 \equiv 1 \bmod 16$) instructions, then a useful trace cache should contain at least 16001 trace lines.

That's why we use the first scheme in our implementation.

On the other hand, the results in section 4.3 show that the trace cache with 2048 trace lines is much larger than necessary. So the second scheme might also be reasonable.

Or we can try the third scheme: the valid trace line must contain at least one branch or cross a cache block boundary. Thus we can use relative small trace cache and eliminate most of the cache misses.

By the way, the multiple branch predictor should use GLOBAL in the second and the third schemes, rather than GSHARE.

6. Conclusions

We have demonstrated that the trace cache can improve the performance of SMT processor. Our implementation uses a trace cache with 2048 entries, 2-way associativity, GSHARE multiple branch predictor, full matches and only starting from predictable branches. The performance improvement is about 1%. We expect to get better performance from the schemes discussed in section 5.

References:

[1] Dean M. Tullsen, S. J. E., Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm (1996). Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. The 23rd Annual International Symposium on Computer Architecture, Philadelphia, PA.

[2] Eric Rotenberg, S. B., James E. Smith (1996). Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching. The 29th Annual International Symposium on Microarchitecture, Paris, France.

[3] Daniel Holmes Friendly, S. J. P., Yale N. Patt "Putting the Fill Unit to Work: Dynamic Optimizations for Trace Cache Microprocessors.".

[4] Tse-Yu Yeh, D. T. M., Yale N. Patt (1993). Increasing the Instruction Fetch Rate Via Multiple Branch Prediction and a Branch Address Cache.

[5] Thomas M. Conte, K. N. M., Patrick M. Mills, Burzin A. Patel (1995). Optimization of Instruction Fetch Mechanisms for High Issue Rates. The 22nd Annual International Symposium on Computer Architecture, Santa Margherita, Italy.