Instruction Fetching Mechanisms for Superscalar Microprocessors

Steven Wallace and Nader Bagherzadeh

University of California, Irvine, Irvine, CA 92697, USA

Abstract. Instruction fetching is critical to the performance of a superscalar microprocessor. We develop a mathematical model for three different cache techniques and evaluate its performance both in theory and in simulation using the SPEC95 suite of benchmarks. In all the techniques, the fetching performance is dramatically lower than ideal expectations. To help remedy the situation, we also evaluate its performance using prefetching. Nevertheless, fetching performance is fundamentally limited by control transfers. To solve this problem, we introduce a new fetching mechanism called a dual branch target buffer. The dual branch target buffer enables fetching performance to leap beyond the limitation imposed by conventional methods and achieve a high instruction fetching rate.

1 Introduction

The goal of a superscalar microprocessor is to execute multiple instructions per cycle. It relies on instruction-level parallelism (ILP) to achieve this goal [5]. Unfortunately, all of this potential parallelism will never be utilized if the instructions are not delivered for decoding and execution at a sufficient rate.

The underlying problem in fetching instructions using a control flow architecture is control transfers. Even with perfect branch prediction, conditional and unconditional branches disrupt the sequential addressing of instructions. The non-sequential accessing of instructions causes difficulty with fetching instructions in hardware. As a result, the instruction fetcher restricts the amount of concurrency available to the processor [7].

Branch prediction foretells the outcome of conditional branch instructions. Instruction fetch prediction determines the next instruction to fetch from the memory subsystem [2]. Instruction fetch mechanisms involve the process of *how* instructions are fetched from memory and delivered to the decoder. This paper focuses on hardware instruction fetching mechanisms. Hence, only instruction fetching performance is evaluated and does not attempt to evaluate any other performance issues (such as branch prediction, cache, execution, etc.). Our goal is to describe, evaluate, and provide solutions to the first step in a series of hurdles for exploiting high levels of ILP.

To begin with, we describe our fetching model and the terms we use in our analysis. Then, we show why and how much performance is currently limited by control transfers. Three different cache options are then briefly described: a simple cache type, an extended cache type, and a self-aligned cache type. The way in which prefetching is applied in hardware is described. Next, the dual branch target buffer is described.

The theory behind the fetching techniques gives insight into fetching problems and can give expected performance under given conditions. Therefore, a probabilistic model based on the probability of a control transfer is presented for all combinations of the fetching techniques described. The models are evaluated under several different conditions. To verify that these models predict accurately and to show what real conditions provide, the SPEC95 suite of benchmarks are simulated using the different fetching techniques presented. Many examples and derivations of equations have been omitted in this version of the paper, but can be found in [8].

2 Fetching Model

This section describes the fetching model used in the rest of the paper. The cache *line size* is defined to be the size of a row in the instruction cache. We use the terms 'line' and 'row' interchangeably. Also, a *block* is defined to be a group of sequential instructions. A block's *width* is the maximum number of instructions allowable.

The instruction cache reads the requested *fetch block* of width q and returns it to the *instruction fetcher*. The *instruction decoder* receives a *decode block* of width n. If prefetching is applied, up to q new instructions from the instruction fetcher go into the prefetch buffer FIFO, $q > n$ and n instructions come out. Otherwise if prefetching is not used, the fetch and decode widths are equal, and the instruction fetcher delivers instructions directly to the decoder. The instruction fetcher is responsible for determining the new starting PC each cycle, sending it to the instruction cache, cooperating with branch prediction, and determining the new PC. Calder and Grunwald [1] describe different techniques for fast PC calculation.

Johnson defines an *instruction run* to be the sequentially fetched instructions between branches [5]. We further specify that a run is between instructions that transfer control. A control transfer instruction includes unconditional jumps and calls, conditional branches that are taken, and any other instruction that transfers control, such as a trap. The *run length* is the number of instructions in a run. In addition, we define a *block run* to be the instructions from the start of the block to the the end of the block or the first instruction that transfers control. The *block run length* is the number of instructions in a block run.

3 Fetching Limitation

Let n be the width of a block and b be the probability that an instruction transfers control. The expected block run length, $r(n, b)$, is

$$r(n, b) = n(1-b)^n + \sum_{i=1}^{n} i(1-b)^{i-1} b = \frac{1-(1-b)^n}{b}. \qquad (1)$$

Equation 1 represents the weighted sum of all events that could occur in a sequence of n instructions. The first term is the case where there is no control transfer in a block. The second term represents all possible permutations of a control transfer in a block. The limit of $r(n, b)$ as the block width increases is given by

$$\lim_{n\to\infty} r(n, b) = \frac{1}{b}. \quad (2)$$

If a control transfer *requires* another cycle to reach the target address, then only one block of instructions can be fetched in a cycle. Regardless of the type of software scheduling or hardware techniques used to improve fetching, $1/b$ is the limit for the average number of instructions fetched per cycle. Under these conditions, $1/b$ is the maximum average number of instructions per cycle that can be executed on any single-threaded control-flow architecture.

4 Hardware Techniques

4.1 Simple, Extended, and Self-Aligned Cache Options

A straightforward approach to fetch instructions from the instruction cache, the *simple* cache, is to have the line size equal the width of the fetch block. If the starting PC address is not the first position in the corresponding row of the instruction cache, then the appropriate instructions are invalidated and fewer than the fetch width are returned. As with all fetching techniques, if there is an instruction that transfers control, instructions after it are invalidated.

One way to reduce the chance that instructions will be lost from a misaligned target address of a control transfer instruction is to *extend* the instruction cache line size beyond the width of the fetch block. To avoid lost instructions on sequential reads that are not block aligned, the instruction fetcher must be able to save the last $n - 1$ instructions in a row and combine them with instructions that are read the next cycle. Only when there is a control transfer to the last $n-1$ instructions in a cache row, instructions are lost due to target misalignment.

The target alignment problem can be solved completely in hardware with a *self-aligned* instruction cache. The instruction cache reads and concatenates two consecutive rows within one cycle so as to always be able to return n instructions. The hardware cost of the self-aligned cache is to use a dual-port instruction cache, perform two separate cache access in a single cycle, or split the instruction cache into two banks.

4.2 Prefetching

All of the above techniques can be used in conjunction with prefetching. Prefetching helps improve fetching performance, but fetching is still limited because instructions after a control transfer must be invalidated.

The fetch width, q, $q \geq n$, is the number of instructions that are examined for a control transfer. Let p be the size of the prefetch buffer. After the instruction fetcher searches up to q instructions for a control transfer, valid instructions are stored into a prefetch buffer. Each cycle, the instruction decoder removes the oldest n instructions from the prefetch buffer.

4.3 Dual Branch Target Buffer

In this section we introduce the dual branch target buffer (DBTB), which is based on the original branch target buffer (BTB) design by Lee and Smith [6]. Unlike the previous techniques mentioned thus far, the DBTB can bypass the limitation imposed by a control transfer. The DBTB is similar to the Branch Address Cache introduced by Yeh, et. al [9], except the DBTB does not grow exponentially. Conte et al. introduced the collapsing buffer, which allows intra-block branches [4]. The DBTB can handle both intra-block and inter-block branches.

The purpose of a BTB is to predict the target address of the next instruction given the address of the current instruction. We take this idea one step further. Given the current PC, the DBTB predicts the starting address of the following two lines. Using the predicted addresses for the next two lines, a dual-ported instruction cache is used to simultaneously read them. Hence, the first line may have a control transfer in it without requiring another cycle to fetch the subsequent line.

The DBTB is indexed by the starting address of the last row currently being accessed in the instruction cache (i.e., the current PC). The entry read from the DBTB can be viewed as two BTB entries, BTB1 and BTB2. The DBTB entry indexed may match both in BTB1 and BTB2, in one or the other, or none at all. This allows a single DBTB entry to be shared between two different source PCs. Although physically they are one entry because they are accessed with the same index, logically they are separate.

Figure 1 is a block diagram of a DBTB entry and shows how it is used in determining the following two rows' PC starting address, PC1 and PC2. The tag of the current PC is checked against the PC tag found in BTB1. If it matches, then the predicted PC1 found in BTB1 is used. Otherwise, the prediction is to follow through to the next row of the instruction cache. If the value predicted for PC1 matches the value in BTB2, then the prediction for PC2 in BTB2 is used; else, PC2 is predicted to be the next row after PC1. The exit position in DBTB entry indicates where the control transfer (or follow through) is predicted to occur. The DBTB entry also contains branch prediction information about all the potential branches in the referenced line. It may contain no information at all, a one bit prediction, a two-bit saturating prediction, or advanced prediction methods such as two-level branch history [10].

5 Expected Instruction Fetch

The total expected instructions fetched per cycle for simple, extended, and self-aligned fetching are

$$F^{simple}(n,b) == \frac{n}{1+b(n-1)} \tag{3}$$

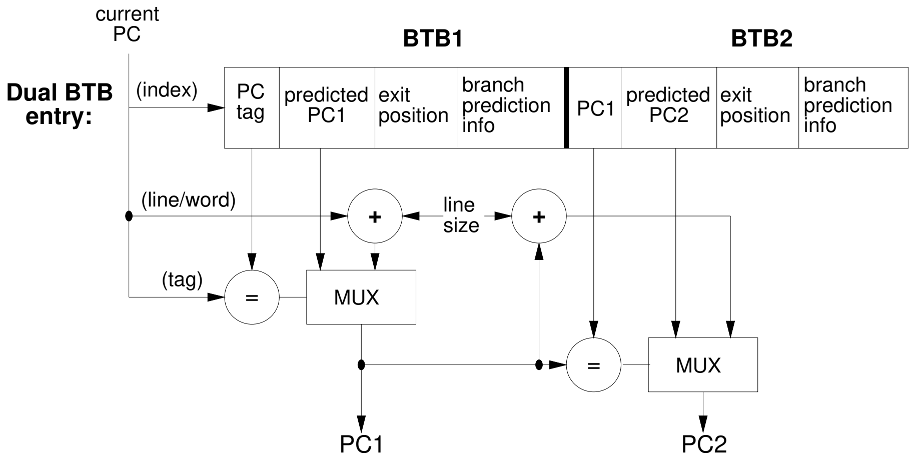

Fig. 1. Block Diagram of Dual Branch Target Buffer Entry

$$F^{extend}(n,b,m) = \frac{(m-n)}{m}\frac{(1-(1-b)^n)}{b} + \frac{n}{m}\frac{n}{(1+b(n-1))} \tag{4}$$

$$F^{align}(n,b) = r(n,b) = \frac{1-(1-b)^n}{b}. \tag{5}$$

5.1 Prefetching

All three cache techniques can be used in combination with prefetching. With prefetching, the fetch and decode widths are not equal. As a result, q may now be substituted for n as a parameter to some of the equations previously defined that did not use prefetching, as will be indicated.

Let E_i^{type} be the probability the starting address in the block is at position i, I_i^{type} be the probability exactly i instructions are available up to and including a control transfer instruction or the end of the block, and P_i be the probability the prefetch buffer contains i instructions, where *type* is one of the three different cache types: *simple*, *extend*, *align* (please refer to [8]).

The total expected instruction fetch for each of the different cache types with prefetching is

$$F_{prefetch}^{type}(p,q,n,b) = n - \sum_{i=1}^{n-1}(n-i)\sum_{\substack{j+k=i\\0\le j\le p\\1\le k\le q}} P_j^{type} I_k^{type}. \tag{6}$$

5.2 Dual BTB

Using the DBTB with the simple, extended, or self-aligned cache without prefetching is simply twice the expected value for half the block size,

$$F^{dbtb,type}(n, b) = 2F^{type}(\frac{n}{2}, b). \tag{7}$$

If prefetching is used with a DBTB, the equations for I_k^{type} in Equation 6 is replaced where q is the total fetch width from both lines and

$$I_k^{dbtb,type}(q, b) = \sum_{j=0}^{k} I_j^{type}(\frac{q}{2}, b) I_{k-j}^{type}(\frac{q}{2}, b). \tag{8}$$

5.3 Evaluation

Figure 2 shows the expected instruction fetch for the simple, extended, and self-aligned cases without prefetching for $b = 1/8$ (common for RISC architectures). Although ideally, for a block size of n a fetching rate of n instructions per cycle is desired, the difference between this ideal and the actual rate increases as n increases. Instead, it approaches $1/b$ (8 in this instance) for each case. The disadvantage for the simple and extended cache techniques is the lower rate at which they reach the limit. It takes a significantly larger value of n to reach the same expected fetch performance. With this extended case of $m = 2n$, its value is the average of the values for the align and simple cases for each n.

Figure 3 shows the expected instruction fetch for the simple cache, extended cache, and self-aligned cache with prefetching for $b = 1/8$, $n = 8$, $q = p + n$, and $m = 2q$ (extended only) verses p. Fetching with DBTB is also shown in the figure for $n = 16$, $q = p + n$, and $m = 2q$ (upper three curves). Similar to the cases without prefetching, the extended case's fetching performance is between the simple and self-aligned cache techniques. We also observe that a simple cache performs significantly less than the self-aligned and extended cache.

The plots presented show that prefetching can significantly increase expected fetching. As the fetch width, q, increases, the expected fetch rate reaches a higher plateau. Unfortunately, with $b = 1/8$ and a decode width of eight, an extensive amount of hardware – a fetch width of sixteen, a prefetch buffer size of thirty-two, and a self-aligned cache – is required to reach almost 7 instructions fetched per cycle, still noticeably below the goal of 8 instructions fetched per cycle. It is difficult to achieve a high fetching rate under those conditions because the decode width is the same size as the $1/b$ limit. On the other hand, if the DBTB is used with prefetching, a high rate close to 14 instructions fetched per cycle can be achieved.

6 Results and Discussion

To verify that our fetching model is reasonable, we simulated the SPEC95 benchmark suite on the SPARC architecture. The suite was compiled using the SunPro

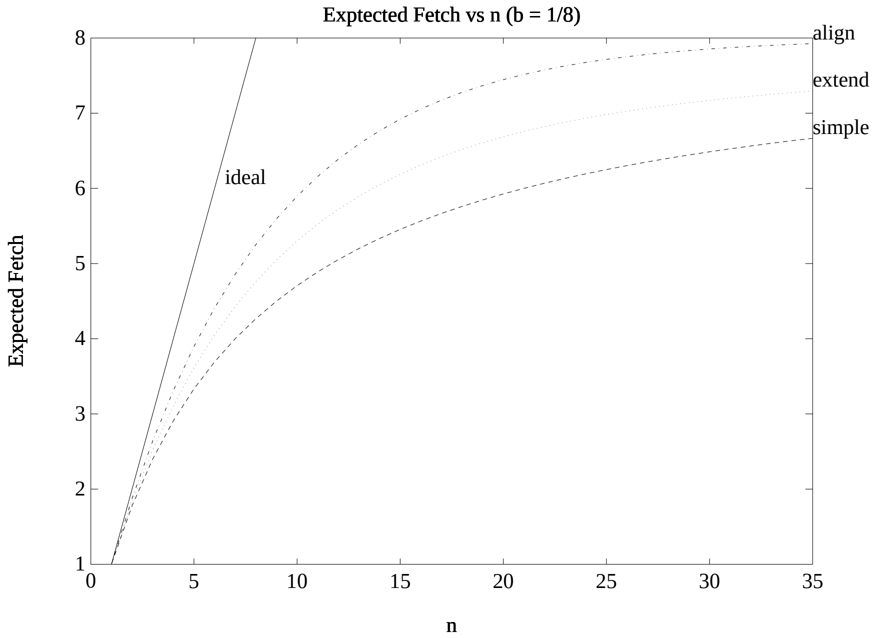

Fig. 2. Expected Instruction Fetch without Prefetching

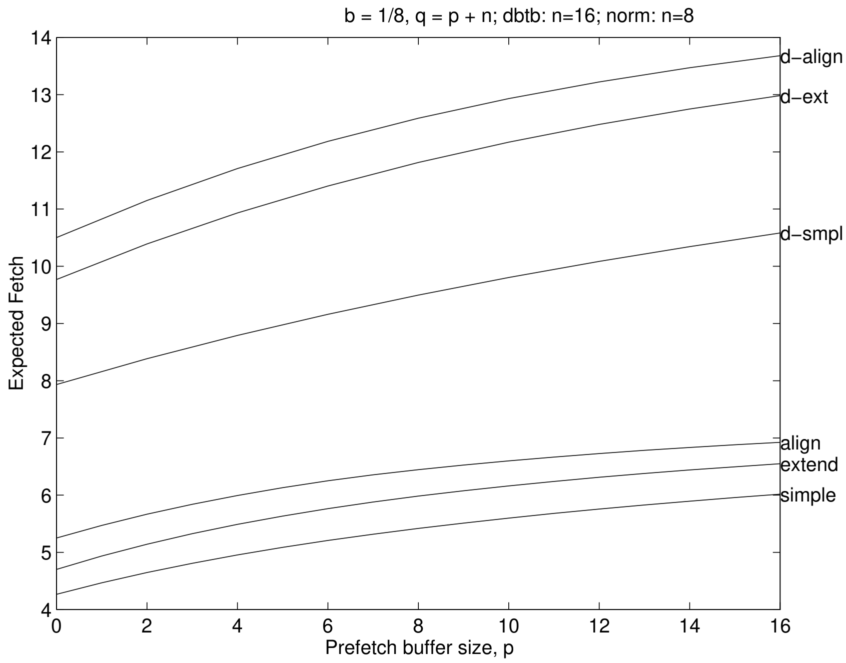

Fig. 3. Different Cache Techniques with Prefetching and DBTB

compiler with standard optimizations (-O). Programs were simulated using the Shade instruction-set simulator [3] and ran until completion or the first four billion instructions.

Table 1 shows the predicted and observed instruction fetch count results of these programs using the three cache techniques without prefetching ($n = 4$) and two with prefetching ($n = 8$, $q = 16$, $p = 16$). The first column in both tables show the value observed for $1/b$, the average run length. The average dynamic run length of a program is the total number of instructions executed divided by the number of instructions that transferred control. The observed value of b for each program was used in its calculation of the expected fetch.

A concern with the fetching model presented is that the distribution of run lengths is expected to be uniform, but in observing actual program behavior the distribution is not uniform. It does, however, generally follow the expected distribution. When the expected fetch is calculated via a weighted sum, the outcome is reasonably accurate. As can be seen in the tables, the difference between the predicted and observed fetch count is usually within a few percent.

The expected and observed performance without prefetching using DBTB for $n = 8$ is exactly twice the values listed in Table 1. Table 2 lists the performance of SPEC95 using DBTB and prefetching ($n = 8, q = 16, p = 8$). The instructions per fetch cycle (IPFC) is listed as well as the instructions per fetch block (IPFB). The results show that close to ideal ($n = 8$) fetching rate is possible if the dual branch target buffer is used with extended or self-aligned cache and prefetching. The fetching hardware mechanism no longer restricts instruction fetching, and therefore, the *possibility* of exploiting instruction-level parallelism and a high instructions per cycle execution rate.

Using a 256-entry, direct-mapped, two-tagged DBTB, we observed that the miss rate was between 10% and 20% for most of the SPEC95 benchmarks. Also, the miss rates for BTB2 was usually slightly higher than BTB1. BTB1 and BTB2 each behaved similarly to a standard BTB. Although perfect branch accuracy was assumed in Table 2 (to make a fair comparison to the other data), it is important to realize that accurate branch prediction becomes critical since more branches need to be predicted accurately per fetch block.

The overall performance will be much lower than the *fetching* rates shown when branch prediction, cache misses, execution, etc., of a real microprocessor are simulated. In addition, the difference between the values will be much smaller. These facts do not devalue the results we have presented. These results show the upper limit achievable using different fetching mechanisms presented, both in theory and in simulation. No doubt, as branch prediction accuracy, cache performance, and execution performance continue to improve in the future, the demand for adequate fetching mechanisms will increase.

7 Conclusion

Many programs have sufficient ILP to execute eight instructions per cycle, and a superscalar microprocessor can be designed to decode and execute eight in-

Table 1. Instructions Fetched per Cycle

		$n = 4$, no prefetch						$n = 8$, prefetch			
Program	$1/b$	simple		extend		align		simple		align	
		pred	obs	pred	obs	pred	obs	pred	obs	pred	obs
go	11.6	3.18	3.20	3.35	3.34	3.51	3.56	6.75	6.75	7.65	7.75
gcc	7.5	2.86	2.86	3.07	3.06	3.27	3.41	5.32	5.10	6.55	6.52
m88ksim	10.2	3.09	3.01	3.27	3.12	3.45	3.48	6.36	6.22	7.44	7.44
compress	9.93	3.07	3.31	3.25	3.43	3.44	3.59	6.27	7.19	7.38	7.64
li	6.9	2.78	2.75	2.99	3.10	3.21	3.31	5.03	4.89	6.22	6.49
ijpeg	21.5	3.51	3.51	3.62	3.59	3.73	3.73	7.79	7.14	7.97	7.89
perl	7.8	2.89	2.88	3.09	3.14	3.29	3.36	5.45	5.16	6.69	6.64
vortex	9.2	3.01	2.90	3.20	3.02	3.39	3.57	6.02	5.38	7.20	7.05
tomcatv	22.0	3.52	3.40	3.63	3.47	3.74	3.69	7.56	7.03	7.92	7.82
swim	114	3.90	3.86	3.92	3.93	3.95	3.96	8.00	7.95	8.00	7.99
su2cor	11.7	3.19	3.25	3.35	3.40	3.52	3.62	6.77	6.10	7.66	7.53
hydro2d	12.9	3.24	3.24	3.40	3.34	3.56	3.63	7.03	6.39	7.77	7.25
mgrid	79.6	3.85	3.68	3.89	3.81	3.93	3.85	8.00	7.97	8.00	8.00
applu	25.3	3.58	3.45	3.67	3.61	3.77	3.73	7.88	7.56	7.98	7.96
turb3d	14.6	3.32	3.37	3.46	3.46	3.61	3.69	7.30	5.87	7.85	6.92
apsi	54.9	3.79	3.76	3.84	3.81	3.89	3.87	7.99	7.93	8.00	8.00
fpppp	13.6	3.28	3.13	3.43	3.30	3.58	3.60	7.15	6.09	7.80	6.96
wave5	23.6	3.55	3.54	3.65	3.62	3.75	3.74	7.85	7.42	7.98	7.92

Table 2. IPFB and IPFC using DBTB w/Prefetching ($n = 8$)

Program	$1/b$	simple		extend		align	
		IPFB	IPFC	IPFB	IPFC	IPFB	IPFC
go	11.6	9.90	7.79	11.2	7.90	12.3	7.98
gcc	7.5	8.01	7.18	9.3	7.49	10.5	7.93
m88ksim	10.2	9.24	7.68	11.0	7.87	11.7	7.98
compress	9.93	9.98	7.78	10.5	7.86	11.8	7.95
li	6.9	7.64	7.37	9.8	7.74	10.4	7.91
ijpeg	21.5	12.0	7.89	12.8	7.91	13.6	8.00
perl	7.8	8.37	7.36	9.9	7.67	10.7	7.93
vortex	9.2	8.07	7.14	10.6	7.52	11.8	7.99
tomcatv	22.0	11.7	7.88	12.4	7.97	13.9	8.00
swim	114	15.0	7.99	15.3	8.00	15.8	8.00
su2cor	11.7	9.5	7.53	11.2	7.78	12.4	7.99
hydro2d	12.9	9.8	7.36	11.5	7.90	12.7	8.00
mgrid	79.6	15.5	8.00	15.6	8.00	15.7	8.00
applu	25.3	12.4	7.97	13.2	8.00	13.9	8.00
turb3d	14.6	10.6	7.38	11.9	7.53	12.5	7.94
apsi	54.9	14.0	7.98	14.7	8.00	15.1	8.00
fpppp	13.6	13.0	7.79	13.5	7.92	14.9	7.99
wave5	23.6	12.4	7.89	13.2	7.96	14.0	7.99

structions per cycle. Unfortunately, because of control transfers in programs, a simple fetching mechanism can not reach this high demand. In fact, it falls far short.

The extended and self-aligned cache techniques showed extremely poor instruction fetching performance, although the self-aligned cache always performed better than the other two. Prefetching helped the situation, and made it possible to approach the upper limit imposed by the probability of a control transfer in a particular program. Using the dual branch target buffer, simulations showed that it is possible to achieve performance beyond the $1/b$ limit. Nevertheless, the fetching performance of a dual branch target buffer is limited to $2/b$ instructions per cycle.

The models that predict the behavior of fetching performance have proven invaluable in the study of instruction fetching. It enables the production of graphs that clearly show the relationship between different fetching options without running hundreds of simulations. They can be helpful in the design of a new superscalar microprocessors to determine which technique will meet the performance objective.

References

1. Brad Calder and Dirk Grunwald.: Fast & accurate instruction fetch and branch prediction. 21^{st} Annual International Symposium on Computer Architecture (1994) 2–11.
2. Bradley Gene Calder.: Hardware and software mechanisms for instruction fetch prediction. PhD thesis, University of Colorado (1995).
3. Bob Cmelik and David Keppel.: Shade: A fast instruction-set simulator for execution profiling. ACM SIGMETRICS (1994).
4. Thomas M. Conte, Kishore N. Menezes, Patrick M. Mills, and Burzin A. Patel.: Optimization of instruction fetch mechanisms for high issue rates. 22^{nd} Annual International Symposium on Computer Architecture (1995) 333–344.
5. Mike Johnson.: Superscalar Microprocessor Design. Prentice Hall, Englewood Cliffs (1991).
6. Johnny K. F. Lee and Alan J. Smith.: Branch prediction strategies and branch target buffer design. IEEE Computer (1984) 6–22.
7. M. D. Smith, M. Johnson, and M. A. Horowitz.: Limits on multiple instruction issue. Proceedings of the Third International Conference on Architectual Support for Programming Languages and Operating Systems (1989) 290–302.
8. Steven Wallace and Nader Bagherzadeh.: Instruction fetching mechanisms for superscalar microprocessors. Elec. & Comp. Engr. U. Cal., Irvine. Tech. Rep. ECE 96-05-02 (1996)
9. Tse-Yu Yeh, Deborah T. Marr, and Yale N. Patt.: Increasing the instruction fetch rate via multiple branch prediction and a branch address cache. 7^th ACM International Conference on Supercomputing (1993) 67–76.
10. Tse-Yu Yeh and Yale N. Patt.: A comparison of dynamic branch predictors that use two levels of branch history. Proceedings of the 20th International Symposium on Computer Architecture (1993) 257–266.