Data Dependence Path Reduction with Tunneling Load Instructions

Toshinori Sato

Microelectronics Engineering Lab.,
Toshiba Corp., Kawasaki 210, Japan
toshinori.sato@toshiba.co.jp

Abstract. The technique for reducing the length of the data dependence path is presented. This technique, named *tunneling-load*, utilizes the register specifier buffer in order to hide the load latency, and thus reduces the length of the data dependence path. True data dependences can not be removed by any techniques such as register renaming, and are the unavoidable obstacle limiting the instruction level parallelism. The length of the data dependence path including the load instructions is longer than those of other instructions, because the latency of the load instruction is longer than those of other instructions. In order to reduce the dependence path length including the load instructions, we propose the tunneling-load technique. We have evaluated the effects of the tunneling-load, and found that in an in-order-issue superscalar platform the instruction level parallelism is increased by over 10%.

1 Introduction

True data dependences can not be removed by any techniques such as register renaming[12], and are the unavoidable obstacles limiting the instruction level parallelism (ILP). The latency of the load instruction is longer than those of other instructions, and hence the temporal distance of the data dependency caused by the load instruction is also longer.

Among the factors increasing the load latency, the penalty of the data cache miss has been dealt with by a variety of techniques such as non-blocking cache[9], software prefetching[14], hardware prefetching[7], and dynamic code scheduling[6]. Even if the penalty of cache miss is solved, the load latency still remains. This load latency is the load-use hazard explained as follows. Fig. 1 shows the traditional 5-stage pipeline, which consists of the instruction fetch (IF), the instruction decode and register fetch (ID), the execution and effective address generation (EX), the memory access (MEM), and the write back (WB) stages.

The load instruction is processed in the following fashion. First, the effective address is generated during the EX stage. Second, the data cache is accessed with this effective address at the beginning of the MEM stage. And finally, the result is obtained at the end of the MEM stage. Therefore, the result is unavailable to the instruction which immediately follows the load instruction. In other words, because the load instruction executes two operations including

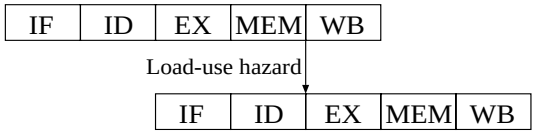

Fig. 1. Load-Use Hazard

the effective address generation and the memory access, the latency of the load instruction has to be longer than those of other instructions.

Recently, the impact of the load-use hazard has become serious. This is because the load latency becomes a major factor obstructing the processor performance, as the instruction issue rate increases. For example, increasing the load latency from 1 to 2 degrades the performance of four-issue processors by approximately 30%[5]. This means that the processor performance could be improved by 30%, if the load latency existing in the 5-stage pipeline were hidden.

In order to hide the load latency, several address prediction techniques have been proposed[1, 2, 8, 11, 15]. Using these techniques, the load address is calculated a few cycles earlier before it would normally be computed in the pipeline, and the load latency can be hidden if the prediction is correct. These techniques carry out the speculative data cache fetching, which causes the explosion of the memory traffic and the pollution of the data cache.

This paper proposes the technique solving the problems described above. The proposed technique, named *tunneling-load*, does not predict the effective address, but generate it. The correctness of the effective address generated at earlier pipeline stage is confirmed before starting the data cache access, and thus the useless speculative cache fetching is removed. Using the tunneling-load, the data dependences caused by the load instructions can be hidden, and therefore it becomes possible to reduce the length of the data dependence path including the load instructions. As the results, the more ILP can be exploited.

The organization of the rest of this paper is as follows. Section 2 surveys previously proposed related works. The tunneling-load mechanism is explained in Section 3. In Section 4, the evaluation methodology is presented and the effect of the tunneling-load is evaluated in Section 5. Section 6 discusses the future study. And finally, our conclusions are presented in Section 7.

2 Previous Work

In order to hide the data cache access latency, several load address prediction techniques have been proposed [1, 2, 8, 11].

Golden et al.[11] and Eickemeyer et al.[8] proposed the load target buffer (LTB) and the load delta buffer (LDB), respectively. These mechanisms are very similar. The LTB (LDT) is indexed by the instruction address and accessed at the IF stage. The load address is predicted by the actual address previously used and the stride value, and the predicted and actual addresses must be compared when

the actual one is generated at the end of the EX stage. If these two addresses match, the load latency is hidden by 2 cycles on the traditional 5-stage pipeline. The accuracy of prediction by the LTB (LDT) is relatively low, because these techniques cannot use the value of register file after modification. Hence, there is a tradeoff between the prediction accuracy and the hardware cost. In order to achieve sufficient accuracy, a lot of entries are necessary. In [8], the reduction of the entries is examined, but the reduction degrades the prediction accuracy. Furthermore, these techniques have a serious drawback. They execute both the actual effective address generation and the comparison between the predicted and actual addresses in one cycle. This can easily become a critical path limiting the cycle time of the pipeline. Thus, in practical implementation, there might be misprediction penalties in spite of their conclusion that the performance is not degraded by any miss penalties.

Austin et al.[1] examined the data cache organization, and proposed the technique whereby the effective address was generated and the data cache was accessed during the EX stage. Because the tag field of the effective address is required after the data cache has been read by the index field, only the set index must be calculated early in the stage. By analyzing the reference type and the offset distribution of several benchmark programs, the fast prediction mechanism of the set index is explored, which is constructed with one stage of logical OR gates. However, the simplest prediction circuit has a significant impact on the cycle time of the processor, because it is on the critical path through the cache memory array. That is, it is difficult to implement the prediction mechanism in the situation that the access time of the data cache dominates the cycle time of the processor, as has recently become commonplace. Another proposal by Austin et al.[2] is the base register and index cache (BRIC), which holds the base and index register values. Since the BRIC is accessed earlier than the register fetch stage and the effective address is generated using the fast address calculation[1], the load instructions can be executed up to 2 cycles earlier than normal execution. This technique relies on the fast address calculation, and thus it is also difficult to implement the prediction mechanism due to the same reason explained above.

All of previous works are promising only if the prediction is correct. However, when the prediction fails, there are several drawbacks related to the speculative cache access. Useless cache access results in the explosion of memory traffic and the pollution of data cache causing the performance degradation. Furthermore, the exception signaling for the speculative cache access is not desirable, because the exception might not occur when the data cache is accessed with the actual address. Thus, the special exception handling mechanism for the speculative cache access must be implemented.

3 Tunneling-Load

In this section, we propose the tunneling-load technique in order to reduce the length of data dependence path. Firstly, we present the mechanism for the

tunneling-load technique. And next, we explain why speculative data cache fetching can be eliminated.

3.1 Tunneling-Load Mechanism

We have already proposed the address prediction scheme using the register specifier buffer (erstwhile address prediction buffer)[15], which has the same problems explained at previous section. In order to solve the problem, we propose the tunneling-load technique. The tunneling-load technique utilizes the register specifier buffer (RSB), which is similar to the branch target buffer (BTB), but it has three differentiate features. First, the RSB does not contain the target addresses, but the register specifiers which store the base and index addresses. Second, the effective address can be generated, even if there are not any entries where the register specifiers are contained. In such a case, the RSB supplies the stack pointer and the zero register specifiers. And finally, the RSB is not accessed by the program counter (PC), but the target program counter (TPC) which walks ahead of the PC. Fig. 2 shows an entry of the RSB. It mainly consists of three fields, namely the tag instruction address field, and the base and index register specifier fields. In addition, each entry contains the valid bit (`v`) and the information field for the least recently used replacement (`LRU`).

Fig. 3 indicates the tunneling-load mechanism. Let us see the pipeline attached with the RSB. As shown in Fig. 3, the traditional 5-stage pipeline can be reconstructed into a 6-stage pipeline including the RSB access (RSB), the instruction fetch (IF), the instruction decode and register fetch (ID), the execution and effective address generation (EX), the memory access (MEM), and the write back (WB) stages. The PC and TPC indicate the IF and RSB stages, respectively. The reconstruction of the pipeline into 6-stage does not increase the branch misprediction penalty because the PC indicates the IF stage. During the recovery process, the TPC is also restored with help of the branch prediction after the PC is corrected.

The technique requires additional hardware support consisting of the RSB, the TPC, a scoreboard, an address calculating adder, and two 5-bit comparators. The scoreboard consists of n 1-bit entries, where n is the number of registers. If a bit is set, this indicates that the corresponding register is updated at immediately previous cycle. The comparators compare the register specifiers obtained from the RSB and those decoded from the instruction code. In addition, the register file must provide two more read ports, and the data cache has to be dual-ported.

The process of the technique is as follows. It consists of five steps. `Step 1`: The RSB is accessed by the TPC and supplies the base and index register

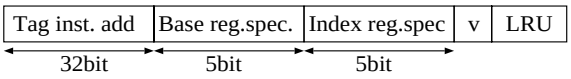

Fig. 2. RSB Entry Field

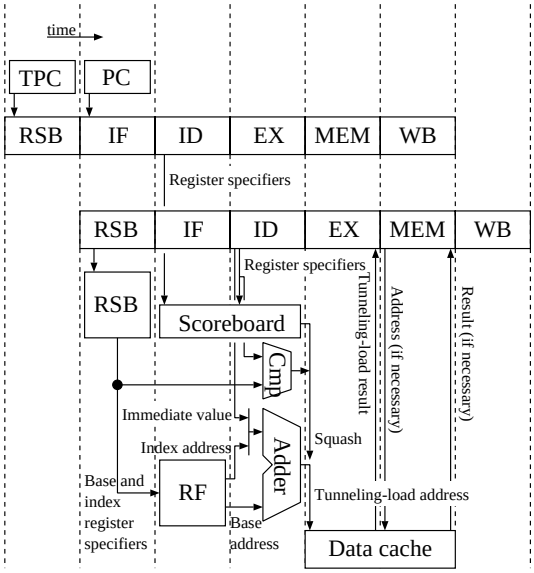

Fig. 3. Tunneling-Load Mechanism

specifiers to the pipeline.

`Step 2`: At the next cycle, the PC indicates the target instruction indicated by the TPC at the previous cycle, and the target instruction is fetched. The base and index values are obtained from the register file with the base and index register specifiers supplied by the RSB.

`Step 3`: At the next cycle, the effective address is generated. If the addressing mode is `register + register`, it is generated with the base and index values. Otherwise, it is generated with the base value and the immediate offset which is decoded at this cycle. Simultaneously, the target instruction is decoded. The scoreboard bits for the base and index registers are checked. If any scoreboard bits are set, the generated address is squashed because it causes the address-generation miss[1]. The comparisons between the register specifiers obtained from the RSB and those decoded from the instruction code are also executed. If they do not match, the generated address is squashed because it causes the primary or bad TPC misses. When the target instruction is not a load instruction, the generated address is discarded. Otherwise, if the address of target instruction is not contained in the RSB, the address and the base and index register specifiers are stored in the RSB. And lastly, the scoreboard bits for the destination registers of all instructions at this stage are set.

`Step 4`: Next, when the tunneling-load succeeds, the datum is fetched from the data cache by the generated address and the load latency is successfully hidden. Otherwise, there is not any operation, thus useless speculative cache access is eliminated.

[1] The address-generation, primary, and bad TPC misses are explained in Section 3.2.

`Step 5`: Finally, if the tunneling-load fails, the datum is fetched by the normally generated address at the MEM stage normally. Thus, there are no miss penalty cycles.

As described above, no speculative data cache fetching belongs to the tunneling-load. Hence, this technique is free from the problems of any load address prediction techniques: the memory traffic explosion, the data cache pollution, and the misprediction penalty. Note that even though the speculative cache access is not executed, the load latency is successfully hidden by the tunneling-load because the speculative cache access is useless.

3.2 Eliminating Useless Cache Fetching

In this section, we explain how the speculative data cache fetching can be eliminated. The address generation misses are classified into three categories: the address-generation, the primary, and the bad TPC misses[15].

The address-generation miss is caused when the base and index register values are modified by the instructions immediately before the target load instruction. This register modification can be found before the load address is generated. Let us see Fig. 3. The destination register specifiers of the preceding instructions which modify the base and index values for the succeeding load instruction are supplied at the ID stage. Similarly, the base and index register specifiers of the succeeding load instruction are obtained from the instruction operand at the ID stage. Therefore, it is possible to detect the register modification in question by scoreboarding between the destination registers of the preceding instructions and the source registers of the succeeding load instruction at the ID stage.

The primary miss is meant that the address of the target load instruction is not cached in the RSB. The primary miss does not always cause the misprediction, because the stack pointer specifier supplied by the RSB as the default may be proper. The misprediction occurs when the stack pointer is not the base register for the load instruction in question, or when the stack pointer is the base register but its value is modified. Therefore, the misprediction in this category is verified by two mechanisms. One is the scoreboarding described above, which checks the modification of the stack pointer value. The other is a comparator which compares the stack pointer specifier and the register specifiers obtained from the instruction code.

The bad TPC miss is caused by the branch mispredictions including two cases explained as follows. One is that the instruction in question is not a load instruction. In this case, even if an incorrect address is generated, the address is discarded at `Step 3` and the misprediction never occurs. The other is the case that the instruction is a load instruction. In this case, the specifiers supplied by the RSB are not guaranteed as correct ones, so the same mechanism as to the primary miss is enough for verification: the scoreboarding and the comparing.

From these investigations above, we obtain the following. The correctness of any load address predictions can be verified at `Step 3`. Therefore, it becomes possible to eliminate useless data cache accesses.

Table 1. Baseline Processor Configuration

Fetch Width	1, 2, 4, 8 instruction(s)
Branch Predictor	1024 entry direct-mapped BTB, gshare scheme[13], 12-bit BHR, 4096 entry PHT, 2 cycle miss penalty
Issue Width	1, 2, 4, 8 instruction(s)
FU's	1, 2, 4, 8 universal functional unit(s) except Ld/St
FU Latency	iALU 1, iMul 3, iDiv 6, Ld/St 2, fAdd 2, fMul 3, fDiv/Sqrt 6
RF's	32 32-bit integer registers, 32 32-bit floating point registers
I-Cache	16K 2way set-associative, 32 byte blocks, 6 cycle miss penalty
D-Cache	16K 2way set-associative, 32 byte blocks, 2-port,write-back, non-blocking load, hit under miss, 6 cycle miss penalty
L2 Cache	ideal

4 Evaluation Methodology

4.1 Processor Model

We evaluated the effect of the proposed mechanism by using the SimpleScalar tool set (version 1.0.2)[3]. The SimpleScalar architecture is based on the MIPS architecture, and the cycle-by-cycle simulator is executed on a SPARCstation.

The baseline model is an in-order-issue superscalar processor. The superscalar degree is between 1 and 8. The pipeline is a 6-stage pipeline shown in Fig. 3. The configuration of the baseline processor is summarized on Table 1. The functional units are universal and there are no restriction on the combination of instructions issued together, except that the number of the load instructions which can be issued per cycle is one. The instruction fetch width and issue width are same. The RSB is constructed full-associatively, and the number of its entry is 64.

4.2 Workload

The SPEC92 benchmark suite is used for this study. The reference input files which are provided by SPEC are used with slight modifications. The left side of Table 2 shows the summary. The Fortran programs were converted to C programs using AT&T F2C (version 1994.11.03), and then all programs were compiled by GNU GCC (version 2.6.3) with the optimization option, `-O3`. Each program was executed to completion or for the first 1 billion instructions. The right side of Table 2 shows the dynamic instruction count executed. Note that the statistics in the table do not include the cycles executed by the operating system.

5 Evaluation

In this section, we present experimental results. Firstly, we evaluate the performance improvement when the RSB is attached to the baseline model. Secondly, we make an evaluation with an alternative memory system for the baseline

Table 2. Benchmark Programs

Benchmark	Input	Modification	# of inst(mil.)
008.espresso	tial.in		1000.0
022.li	li-input.lsp		1000.0
023.eqntott	int_pri_3.eqn		1000.0
026.compress	in		664.4
072.sc	loada2		1000.0
085.gcc	insn-recog.i		164.2
013.spice2g6	greycode.in		1000.0
015.doduc	doducin		1000.0
034.mdljdp2	mdlj2.dat	MAX_STEPS=250	1000.0
039.wave5			1000.0
047.tomcatv		N=129	480.8
048.ora	params	ITER=15200	116.8
052.alvinn		NUM_EPOCHS=50	1000.0
056.ear	args.short		420.5
077.mdljsp2	mdlj2.dat	MAX_STEPS=250	1000.0
078.swm256	swm256.in	ITMAX=120	1000.0
089.su2cor	su2cor.in		766.2
090.hydro2d	hydro2d.in		11.2
093.nasa7			1000.0
094.fpppp	natoms		1000.0

Table 3. Examined Models

Name	B1L1	B2L1	B4L1	B8L1	B1L2	B2L2	B4L2	B8L2
Issue	1	2	4	8	1	2	4	8
RSB	No							
Ld/St	1				2			
D cache	1-port				2-port			

Name	R1d	R2d	R4d	R8d	R1s	R2s	R4s	R8s
Issue	1	2	4	8	1	2	4	8
RSB	Yes							
Ld/St	2				1			
D cache	2-port				1-port			

model. The baseline model is attached with two load-store units and its data cache is modified to be dual-ported. And lastly, we investigate the performance degradation when the data cache of the evaluated model is single-ported. The designs we examine are listed in Table 3. First row shows the names for evaluated models. Second row indicates the instruction fetch width, and the next row shows if there are the RSB attached to the model. Fourth row indicates the number of load store unit. And last row shows the number of the data cache port.

We use the Instruction Per Cycle (IPC) as a performance metric.

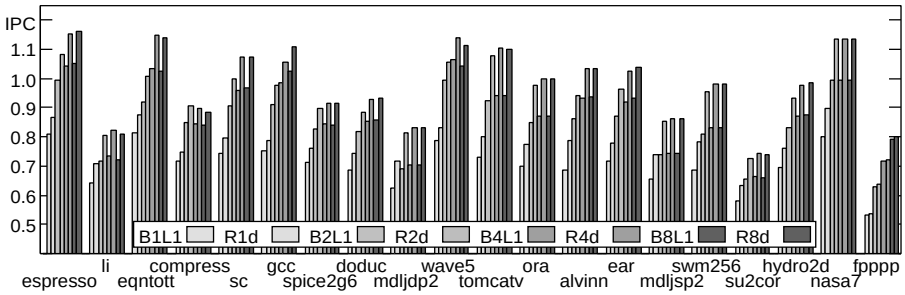

Fig. 4. Simulation Results (1)

5.1 Effect of Tunneling-Load

Fig. 4 shows the performance improvement when the tunneling-load mechanism is attached to the baseline model, which has only one load-store unit and whose data cache is single-ported. Note that the data cache of the evaluation model is dual-ported.

From Fig. 4, in the case of the single-issue model, the performance for the integer programs is increased by 6.7% on average with the maximum of 9.7%. For the floating-point programs it is increased by 9.7% on average with the maximum of 15.2%. The results are different from those explained in [15], which says the performance of the single-issue processor is improved by 1.5% on average for the floating-point programs. This is because the latencies of floating-point units are different between the evaluate model in this paper and that in [15].

In the case of the dual-issue model, the performance is increased by 9.1% on average with maximum of 11.5%, and 11.7% on average with maximum of 17.7%, for the integer and floating-point programs, respectively. As can be seen from Fig. 4, when the issue width is increased from one to two, the performance is jumped up significantly in both the baseline and evaluated models.

In the case of the four-issue model, the performance is increased by 9.9% on average with maximum of 12.2%, and 12.0% on average with maximum of 18.1%, for the integer and floating-point programs, respectively. In the case of the eight-issue model, the performance is increased by 9.8% on average with maximum of 11.9%, and 12.1% on average with maximum of 16.8%, for the integer and floating-point programs, respectively. It should be noted that in some programs the performance of the eight-issue model is lower than that of the four-issue model. One of the reasons is the miss rate of the instruction cache. In the case of wave5, for example, the instruction cache miss rate of the eight-issue model is 10.8% worse than that of the four-issue model.

5.2 Hardware Cost Consideration

In comparison with the baseline model, the evaluated model described above has an additional load-store unit and its data cache is dual-ported. In order to

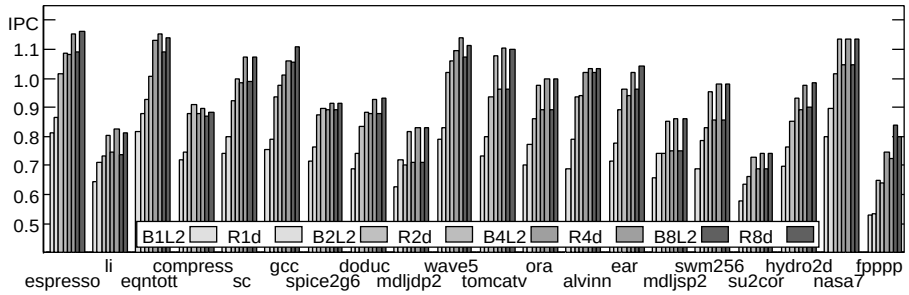

Fig. 5. Simulation Results (2)

make a fair comparison in terms of the hardware cost, we take up an alternative memory system for the baseline model, including two load-store units and the dual-ported data cache. The results are shown in Fig. 5.

In the case of the single-issue model the performance for the integer programs increased by 6.7% on average with the maximum of 9.7%. For the floating-point programs it is increased by 9.7% on average with the maximum of 15.2%. In the case of the dual-issue model, the performance is increased by 6.9% on average with maximum of 9.8%, and 9.0% on average with maximum of 16.6%, for the integer and floating-point programs, respectively. In the case of the four-issue model, the performance is increased by 5.7% on average with maximum of 10.2%, and 8.4% on average with maximum of 16.8%, for the integer and floating-point programs, respectively. In the case of the eight-issue model, the performance is increased by 6.2% on average with maximum of 10.1%, and 8.3% on average with maximum of 16.8%, for the integer and floating-point programs, respectively.

As can be seen, the performance improvement is slightly reduced in general, if the memory system of the baseline model is extended.

5.3 Single-Ported Cache Case

This section presents the results for the evaluated model whose data cache is single-ported. Because the die area of the dual-port cache is twice larger than that of the single-ported cache, the single-port cache is desirable. The results are shown in Fig. 6. The performance is comparable to the model with dual-port cache. Its degradation is less than 1%, except spice2g6. Thus, it is found that the tunneling-load is effective without additional hardware cost into the memory system.

6 Current Status and Future Study

In this section we explain the current status and future direction of this study.

As mentioned in [15], it is possible to hide the load latency that is grater than two if the distance between the TPC and the PC is enlarged. Recent study shows

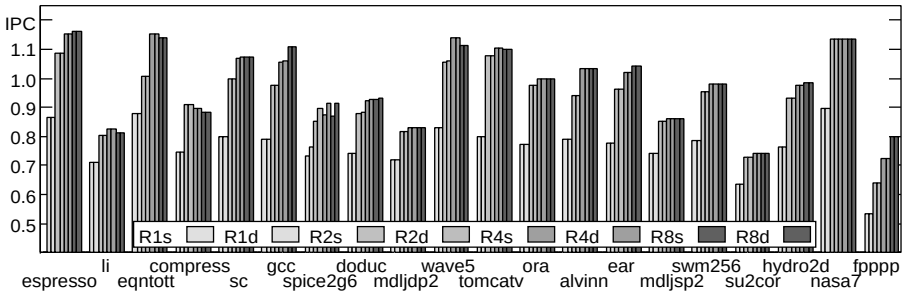

Fig. 6. Simulation Results (3)

that the tunneling-load is effective when the TPC points ahead even farther. Due to lack of space, detailed and extensive studies will be presented in future publications.

Future study dealing with the tunneling-load will investigate the register port consideration and the attachment to the out-of-order execution engine.

The drawback of the tunneling-load is the impact on the access speed of the register file when the multiple loads per cycle are supported. The number of ports on the register file would increase by four rather than two per load pipe. The additional ports may slow the register file, and this has a serious impact. Since the register port requirement is a common problem among the high-issue-rate processors, several techniques which attack to the register port requirement[4, 10, 17] are proposed. We are now investigating techniques which relieve the demand on the register ports.

Clearly, out-of-order-issue processors will not suffer as much from the pipeline interlocks caused by the load-use hazard. However, there are still existed the possibilities to remove true data dependences between an unresolved load instruction and succeeding instructions which are dependent upon the load instruction. In such a case the speculatively resolving the load instruction is effective[16]. The tunneling-load might be applicable to this purpose, and an algorithm is being examined. Note that the speculative load resolution is different in purpose from the speculative data cache fetching.

7 Concluding Remarks

We have proposed the new technique for reducing the length of the data dependence path and exploiting the ILP, and evaluated its effect by the cycle-by-cycle simulation. This technique, named tunneling-load, utilizes the RSB. Using the tunneling-load, useless speculative cache access is eliminated and the problems included in any load address prediction techniques are solved.

We started with the investigation of the load address prediction techniques, and explained the problems belonging to them. We studied the address prediction miss and presented that the speculative cache access is unnecessary for reducing

the load latency if the new verification mechanism is introduced. And then we proposed the tunneling-load technique, which does not predict the load address but generate it. By eliminating the speculative data cache access, the problems such like the memory traffic explosion and the data cache pollution are solved.

The experimental results show that the tunneling-load exploits the ILP, especially for floating point workloads. We have found that on the four-issue machine the tunneling-load with the RSB which has 64 entries improves the performance by approximately 10% on average.

We are encouraged that the tunneling-load is very effective for improving the pipeline performance.

References

1. T.M.Austin et al., "Streamlining data cache access with fast address calculation", Proc. of ISCA22, pp.369-380, 1995.
2. T.M.Austin et al., "Zero-cycle loads: microarchitecture support for reducing load latency", Proc. of MICRO28, pp.82-92, 1995.
3. D.Burger et al., "Evaluating future microprocessors: the SimpleScalar tool set", Technical Report CS-TR-96-1308, University of Wisconsin Madison, July 1996.
4. A.Capitanio et al., "Partitioned register files for VLIWs: a preliminary analysis of tradeoffs", Proc. of MICRO25, pp.292-300, 1992.
5. P.P.Chang et al., "IMPACT: an architectural framework for multiple-instruction-issue processors", Proc. of ISCA18, pp.266-275, 1991.
6. P.P.Chang et al., "Comparing static and dynamic code scheduling for multiple-instruction-issue processors", Proc. of MICRO24, pp.25-33, 1991.
7. T-F.Chen et al., "Effective hardware-based data prefetching for high-performance processors", IEEE Trans. Computers, vol.44, no.5, pp.609-623, May 1995.
8. R.J.Eickemeyer et al., "A load-instruction unit for pipelined processors", IBM J. Res. Develop., Vol.37, No.4, pp.547-564, July 1993.
9. K.I.Farkas et al., "Complexity/performance tradeoffs with non-blocking loads", Proc. of ISCA21, pp.211-222, 1994.
10. M.Franklin et al., "Register traffic analysis for streamlining inter-operation communication in fine-grain parallel processors", Proc. of MICRO25, pp.236-245, 1992.
11. M.Golden et al., "Hardware support for hiding cache latency", Technical Report CSE-TR-152-93, University of Michigan, Feb. 1993.
12. R.M.Keller, "Look-ahead processors", ACM Computing Surveys, vol.7, No.4, pp.177-195, Dec. 1975.
13. S.McFarling, "Combining branch predictors", WRL Technical Note TN-36, Digital Western Research Laboratory, 1993.
14. T.C.Mowry et al., "Design and evaluation of a compiler algorithm for prefetching", Proc. of ASPLOS V, pp.62-73, 1992.
15. T.Sato et al., "Hiding data cache latency with load address prediction", IEICE Trans. Inf. & Syst., vol.E79-D, no.11, pp.1523-1532, Nov. 1996.
16. T.Sato, "Data dependence speculation combining memory disambiguation with address prediction", Proc. of SWoPP'97 (IPSJ SIG Notes), Aug. 1997.
17. S.Wallace et al., "A scalable register file architecture for dynamically scheduled processors", Proc. of PACT'96, pp.179-184, 1996.

This article was processed using the LaTeX macro package with LLNCS style