計算機アーキテクチャ 135−10
（1999. 11. 27）

投機的実行を行なうVLIWプロセッサの命令供給機構の設計

ハラダ・ウゴ・ケンジ・ペレイラ† 仲池卓也†
小林広明† 中村維男†

本論文では，投機的実行をサポートするVLIW計算機の命令フェッチ機構について述べる．ダイナミックブースティングは，コンパイラとハードウェアの支援により，プログラムの動的挙動に対応し，投機的実行を行なう手法である．ダイナミックブースティングでは，コンパイラが，連続した基本ブロック中で依存関係が無く，並列実行可能な命令を検出し，それらの命令にラベルを付ける．そのラベルが付けられた命令は，ハードウェアによって実行時に検出され，投機的実行が行なわれる．SPECint95ベンチマークを用いた実験により，ダイナミックブースティングは，最大20%の性能向上を示した．また，ダイナミックブースティングのためのハードウェアを設計した結果，ハードウェアの複雑性は，低く抑えることが可能であることが分かった．

The Design of an Instruction Fetch Unit for VLIW Processors Supporting Speculative Execution

HUGO KENJI PEREIRA HARADA,† TAKUYA NAKAIKE,†
HIROAKI KOBAYASHI† and TADAO NAKAMURA†

This paper presents an instruction fetch scheme capable of speculatively executing instructions in VLIW processors. This is achieved with the compiler and the underlining hardware working together in a scheme called *Dynamic Boosting (DB)*. In dynamic boosting, the compiler is responsible for finding instruction level parallelism (ILP) beyond the boundaries of basic blocks. It then schedules and labels the independent instructions belonging to different basic blocks in such a way that the hardware is able to detect and execute these instructions in parallel at run time. The software simulation results show that a speed-up of at most 20 % was achieved in the SPECint 95 benchmarks. In addition, the preliminary results on hardware cost and gate level speed show that the hardware complexity and cost are reasonable considering the obtained speed-ups.

1. Introduction

The efficient exploitation of the instruction level parallelism (ILP) inherent in every program can lead to remarkable performance gains. The more a processor can exploit ILP, the faster it can execute programs since it can dispatch multiple operations in a cycle. Superscalar and VLIW processors take two completely different approaches when extracting ILP from a program code.

In superscalar processors, data and control dependencies are analyzed by the hardware at runtime and instructions that are found to be independent are then executed in parallel by the multiple functional units (FU) available. Since ILP extraction occurs at run time, the hardware implementation of a superscalar processor tends to be rather complex. If the number of FUs increases, more ILP is going to be necessary to keep these units busy, meaning that data and control dependencies analyses among many more instructions will have to be performed. This inevitably results in complex fetch and decode structures.

Since software can manage complex tasks better than hardware, in a VLIW processor the compiler performs ILP extraction. Independent instructions are packed together into a single, long instruction - VLIW instruction. When a VLIW instruction is executed, all the component instructions are executed in parallel by the hardware, which tends to be simple since this is the only task left to it. In general, compilers for VLIW processors employ trace scheduling or super-block scheduling techniques to extract ILP beyond two or more basic blocks[1),2)].

† 東北大学大学院 情報科学研究科
Graduate School of Information Sciences, Tohoku University

This works well for highly predictable numerical applications, but since these scheduling techniques use profile information, their performance in branch-intensive integer applications is poor.

To take advantage of ILP beyond several basic blocks' boundaries, we propose dynamic boosting, a mechanism to dynamically schedule VLIW instructions. With the support of both the compiler and the hardware, dynamic boosting enables the schedule of instructions beyond several basic blocks. The compiler identifies the instructions that can be moved among basic blocks and the hardware dynamically schedules these instructions whenever possible to obtain more ILP. Since ILP extraction takes place at compile time, the hardware needs only to detect which instructions can be moved, by decoding the information that was attached to the instructions by the com-piler. In this paper, we evaluate the performance of the VLIW architecture using dynamic boosting and design its instruction fetch unit by using CAD tools. In addition, we examine the tradeoffs between performance and complexity of the dynamic boosting mechanism.

This paper is organized in five sections. Section two introduces the dynamic boosting algorithm. Section three describes the hardware configuration for dynamic boosting. Section four discusses the hardware design, and reports the simulation results and our analyses. Finally, section five gives some concluding remarks.

2. Dynamic Boosting Algorithm

Boosting is originally defined as the operation of statically moving an instruction to a preceding basic block[3]. In dynamic boosting, VLIW instructions move in a similar way, but their boost time and direction is decided at run time by the hardware.

In dynamic boosting, the compiler analyzes the dependencies that exist in the code and classifies each instruction as *normal*, *lift* or *boost*. *Normal instructions* are the regular instructions contained in the code, and do not interfere in the dynamic boosting scheme. *Boost instructions* are the instructions that can be boosted, and *lift instructions* are the ones, which are executed in parallel with the boost instructions.

Fig.1 shows how the compiler schedules instructions. I1 and I2 are lift instructions and I3, I4, I6 and I7 are boost instructions. BB0 (Basic block 0) is called a lift block because it contains lift instructions. For a similar reason, BB1 and BB2 are called boost blocks. The position of each instruction is decided at compile time. When lift instruction I1 (I2) is detected by the hardware, it is merged with boosting instruction I3 (I4) when the path BB0→BB1 is taken, or with the instruction I6 (I7) when the path BB0→BB2 is taken.

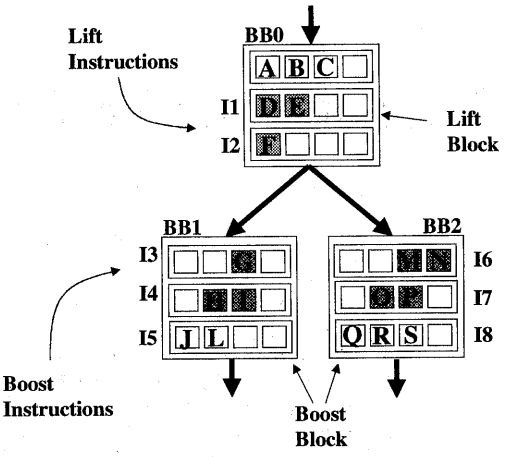


図 1 Instruction scheduling with dynamic boosting

Fig.2 shows the basic hardware structure needed to implement the dynamic boosting scheme. *I-Cache* is the instruction cache. BTB is a Branch Target Buffer. The predictor can be a single level predictor or a two-level predictor[4),5)]. The boost instructions are stored in the *Boosting Table (BT)*.

When a lift instruction (I1 in Fig. 1) is detected by the hardware, it is merged with the correspondent boost instruction (I3 or I5 in Fig. 1) according to the taken path. Lift and boost instructions are fetched from the I-cache and the BT, respectively, and merged in the *Merger*. A BT entry is indexed with a lift instruction address and a new BT entry is created whenever a lift instruction misses on it. When this happens, the correspondent lift-boost pair is executed sequentially, and all the executed boost instructions are then stored in the BT. In this way, the next time this lift instruction is fetched, boosting is performed since the necessary boost instructions are available from the BT.

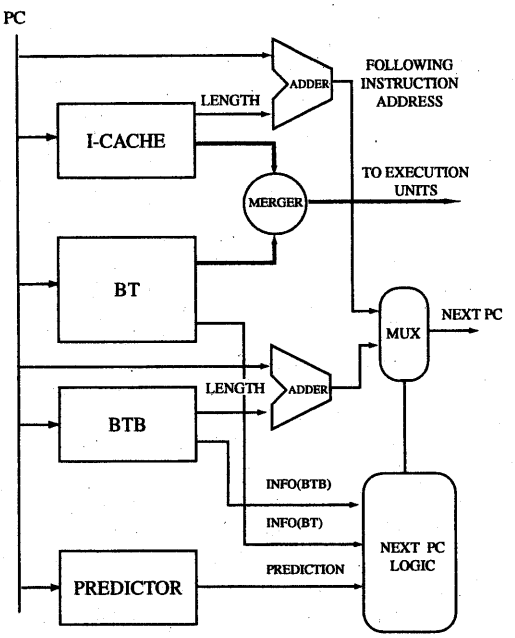
The *Next PC Logic (NPL)* decides the address of the next instruction to be fetched. Choices are between the following instruction address, the target address of a branch. The following instruction address is calculated by adding the length of the

current instruction to the current address. And the target address of a branch is the sum of the current address and the offset to the target. The offset is stored in the BTB. Note that, in the scheme of Figure 2, we can save cache space by caching offsets to target addresses instead of caching the complete target addresses themselves.

It is important to note that the branch target changes when boost happens. In the example in Fig. 1, after boosting the new target address for the branch in I2 becomes the address of instruction I5 when instructions I3 and I4 are boosted ,and the address of instruction I8 when instructions I6 and I7 are boosted. In this paper, we call these new targets *after-boost targets*, and the offsets to the after-boost targets *after-boost offsets*. Also, we call a branch contained in a lift instruction a *lift branch*. After-boost targets are calculated by adding to the target of the lift branch, the after-boost offset. When the lift branch falls through, the after-boost offset is simply the sum of the lengths of all boost instructions together. However, when it is taken, the after-boost offset is calculated by adding the original offset of the lift branch to the sum of the lengths of all boost instructions.

In dynamic boosting, since there are two correspondent boost instructions for each lift instruction, a prediction about which basic block is going to be executed next is necessary when a lift instruction is detected. We make this possible by indexing the BTB and the predictor with the address of the first lift instruction of a lift block.

Memory bandwidth restrictions limit the number of boost instructions that can be cached in the BT. That is the reason why, although there are two correspondent boost blocks to each lift block, boost instructions of only one boost block can be cached in the BT. The scheme presented so far is called *One-Path dynamic boosting* for this reason. There is a need to reconstruct an entry in the BT, whenever the predicted path does not match the path stored in the BT. To prevent the possible performance reduction due to repeated entry reconstructions, the following scheme was proposed. Boost instructions of one path are copied to empty slots in the correspondent lift instructions and the ones in the other path are stored in the BT. As this scheme prepares boost instructions on two paths, it is called *Two-Path dynamic boosting* scheme. This scheme can be implemented in the datapath that performs one-path boosting with very few modifications.



INFO(BTB) = DID BTB HIT ?

INFO(BT) = DID BT HIT ? LAST BOOST INSTRUCTION FETCHED?

図 2 Instruction fetch mechanism implementing dynamic boosting

3. Hardware Implementation

In this section, implementation of the dynamic boosting scheme is discussed.

3.1 Boosting Table (BT)

Each BT entry stores all the boost instructions of a given boost block, and is indexed by the address of the first lift instruction in the correspondent lift block. The BT used in the one-path boosting scheme has a *tag* field and a *boosting number (BN)* field in addition to the field reserved to store the boost instructions. The tag field is used to uniquely identify the mapping of the instruction to the correspondent entry. In the boosting number field is stored the number of the boost instructions stored in the entry.

The basic structure of a BT is shown in Figure 3. It is dual ported, and permits simultaneous read and write operations as long as the accessed entries are different. The *busy* signal is asserted whenever the accessing addresses are the same. *Hit* is asserted when the accessing address hits on BT, and *Last Boost* tells when the last boost instruction was read out from BT. *Index control* is a sim-

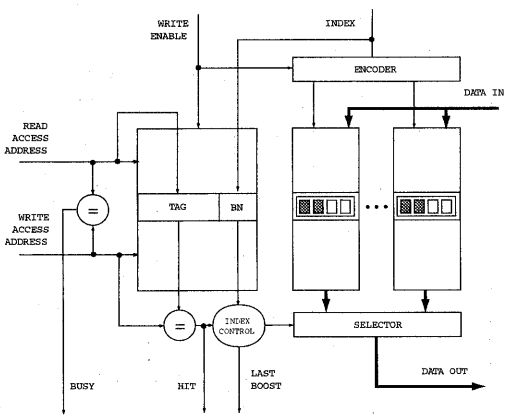


図 3 Boosting table

ple state machine that controls the output order of the boosting instructions. As inputs, the BT has *Index*, *Write Enable* and *Data In*. *Index* works as an index to the boost instruction that is going to be written into the BT. *Write Enable* enables writing to the memory block that contains the tag and BN, and also works as an enable signal to the encoder that controls the writing to the memory blocks that store boost instructions. Lastly, *Data In* contains the boost instructions to be written.

There is a small difference between the BT structures for the one-path boosting and the two-path boosting. Since the two-path scheme prepares boost instructions for both paths, it has to prepare the after-boost offsets for both paths. For this reason, we implement an extra field to store this extra after-boost offset in BT.

3.2 Branch Predictors

As was mentioned before, single level and two-level predictors can be used when implementing dynamic boosting. Nonetheless, there are considerations about which predictor should be chosen when different boosting schemes are used. The one-path scheme has a possibility of suffering performance degradations due to frequent reconstructions of BT entries when the predicted direction changes. Such changes are very common when correlation-exploiting two-level predictors are used. Single-level two-bit predictors, on the other hand, only change their prediction after two mispredictions. This supresses frequent prediction direction changes, making single-level predictors suitable to implement the one-path scheme. Since the two-path scheme prepares boost instructions on both paths, it supports frequent prediction changes.

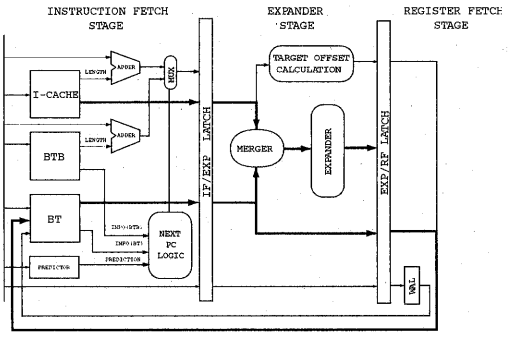


図 4 Datapath for dynamic boosting

Therefore, there are no restrictions on the type of predictors to use when this scheme is implemented.

3.3 Pipelining the Boosting

Figure 4 shows the pipeline structure used in the implementation of dynamic boosting. It is composed of three stages, named; *instruction fetch (IF)*, *expander (EXP)* and *register fetch (RF)*. In the IF stage, instruction fetch, branch prediction and next PC calculation take place. When a lift instruction hits on the BT, the boosting process starts, and boost instructions are fetched and sent to the IF/EXP latch. The merger unit concatenates these instructions in the EXP stage before sending them to the expander.

The *expander* unit rearranges instructions, and issues them to their correspondent functional units. In the RF stage, the address of the first lift instruction is stored in the *write address latch (WAL)*. This address is used to update the predictor and the BTB when the lift branch gets resolved. Operations executed in the IF and EXP stages are repeated a number of times equal to the number of boost instructions stored in the BT.

When a lift instruction misses on the BT, boosting does not happen and execution proceeds normally. The construction of a BT entry effectively starts when the lift instruction that did not hit on the BT is detected by the hardware in the EXP stage. In the RF stage, the address of the detected lift instruction is stored in the WAL. This address is used to index BT when boost instructions are written to it. This writing starts when the first boost instruction reaches the RF stage and continues until all the boost instructions pass through it.

4. Performance Evaluation with Hardware Cost Considerations

In this section, we first evaluate the extra amount of hardware for dynamic boosting, which needs to be added to the datapath of a traditional VLIW processor. Second, we discuss the best combination of size and associativity for the BT.

Finally, we evaluate how the types of predictors affect the total performance of dynamic boosting. Misprediction rates for several branch prediction schemes are measured and the performance of dynamic boosting is examined when the best predictors are used.

4.1 Methodology

To estimate the combinational logic cost necessary for implementing the dynamic boosting mechanism, a complete model of the proposed datapath was described in VHDL. After verifying the correctness of the proposed implementation through gate level simulations, the design was synthesized to schematics and then mapped to the CMOS 0.5 μ m technology with the P2Lib library[6]. A VLIW processor model discussed here has an address length of 32 bits and its instruction has total length of 256 bits. Being each component instruction 64 bits long, and composed of a 32 bit MIPS R3000 instruction in addition to fields where information like instruction type (normal, lift and boost), instruction length and functional unit type are stored.

In software simulations, the optimum BT size and associativity are examined. Here, the one path scheme using a single-level 2-bit predictor was simulated. This predictor was implemented by attaching a counter to each of the BTB entries. In simulation results, this scheme is identified by a 'BTB' [7] label. In figure 5, a 2KB-BT with an associativity of 4 is represented as 2K,4W. Cache size was set to 1MB and its associativity to 8; both numbers are large enough to avoid entry conflicts on the table. Although only the results for the one path scheme are presented here, experiments considering the two path scheme were also executed and generated similar results.

In the experiments where branch predictors were compared to each other, the cache size was set as above. As the purpose of this experiment is to identify the predictor that fits best the characteristics of a traditional VLIW processor, the dynamic boosting algorithm was not implemented during simulations. We use the notation (gshare,X) to represent a two-level gshare predictor[8] that has a branch history register of length X.

表 1 Hardware costs and critical paths for IF and EXP stages

		IF	EXP
One Path	Area (cells)	1972	7622
	Critical path (cells)	N/A	11
One Path	Area (cells)	2001	7577
	Critical path (cells)	N/A	11

When the performance of dynamic boosting was measured as a function of the predictor type, the BT size was set to 128KB and its associativity to 8 in order to keep the conflicts for BT entries at a minimum. Results are shown for the BTB and for the two-level gshare predictor. The implementations for two-level predictors presented in 5) were also tested but they performed poorly when compared to gshare.

4.2 Hardware Cost and Complexity

Table 1 shows the extra amount of combinational logic needed per pipeline stage to implement a datapath capable of dynamic boosting, and the critical path in the expander stage expressed in number of gates. The large number of gates showed in the EXP stage's column is due mainly to the high bandwidth of the expander (256 bits). Nevertheless, a critical path with a small number of gates was obtained since the task performed by it is simple. The critical path for the IF stage is not available because of the lack of data about the timing response of the memory blocks it contains.

4.3 Simulation Results

Figure 5 shows speed-ups as a function of BT size and BT associativity. As it can be seen, improvements in performance due to a smaller BT miss rate were not significant when BT sizes are bigger than 1KB. Bigger associativity can reduce BT miss rates but does not improve performance. Figure 6 shows the misprediction rates for the BTB predictor and for gshare with various branch history register lengths. For history lengths bigger than 12, gshare outperforms the BTB scheme in all tested benchmarks.

Figure 7 shows the effect of dynamic boosting in the number of executed operations per cycle. Regarding predictors, BTB was used for the one path scheme, and gshare was used for the two path scheme. One path scheme performs relatively well despite the higher misprediction rates obtained for the BTB predictor in the preceding experiment.

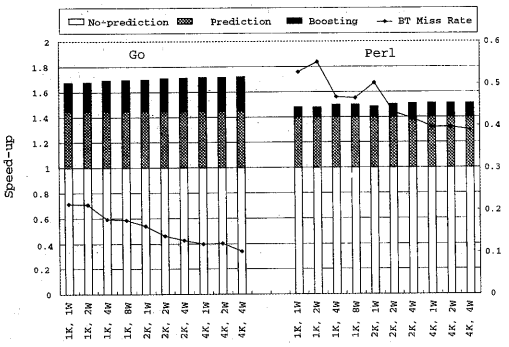


図 5 Speed-up and BT miss rates for various combination of BT size and associativity

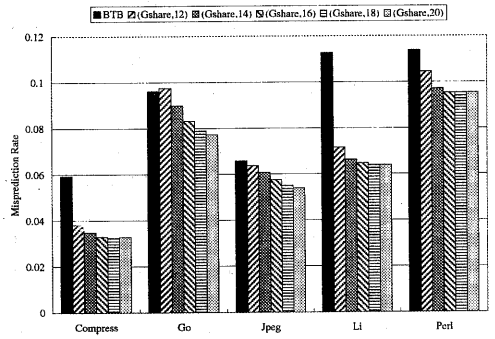


図 6 Misprediction rates

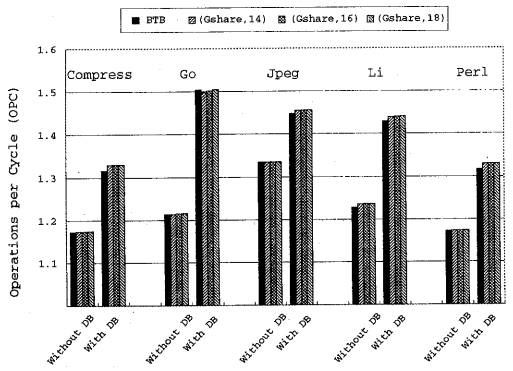


図 7 Operations per cycle with and without dynamic boosting

The results clearly show that VLIW processors that implement dynamic boosting are in average 20% more efficient than the ones which don't.

5. Conclusions

Traditional VLIW processors have trusted the burden of ILP extraction solely to the compiler. Static scheduling of instruction has proved to be efficient on loop-intensive and highly predictable numerical applications. However, their performance in branch-intensive integer applications is somehow degraded. The main reason for this is the frequent behavior changes observed in this programs at run time. To solve this problem this paper presented the dynamic boosting mechanism. In dynamic boosting, instructions can be moved beyond basic blocks boundaries according to highly accurate dynamic branch predictors. This paper has also presented a hardware design and its performance with cost evaluation. The results show that at a very low hardware cost, performances up to 20% were achieved.

参 考 文 献

1) J. A. Fisher. Trace schedulig : A technique for globa microcode compaction. *IEEE Trans. Computer*, Vol. C-30, pp. 478–490, July 1981.
2) S. A. Mahlke, W. Y. Chen, R. A. Bringmann, R. E. Hank, W. W. Hwu, B. R. Rau, and M. S. Schlansker. Sentinel scheduling : A model for compiler - controlled speculative execution. *ACM Trans. Computer Systems*, Vol. 11, No. 4, pp. 376–408, November 1993.
3) M. D. Smith, M. Horowitz, and M. S. Lam. Efficient superscalar performance through boosting. In *Fifth International Conference Architectural Support for Programming Languages and Operating Systems*, pp. 248–259, 1992.
4) T.-Y Yeh and Y. Patt. Two-level adaptive training branch prediction. In *Proc. 24th Annual Symp. and Workshop Microarchitecture*, pp. 51–61, 1991.
5) T.-Y Yeh and Y. Patt. Alternative implementations of two-level adaptive branch prediction. In *Proc. 19th Symposium on Computer Architecture*, pp. 124–134. ACM, May 1992.
6) H. Onodera, A. Hirata, T. Kitamura, K. Kobayashi, and K. Tamaru. P2lib: Process portable library and its generation system. *Journal of Information Processing Society of Japan*, Vol.40, No.4, pp. 1660–1669, April 1999.
7) J. K. F. Lee and A. J. Smith. Branch prediction strategies and branch target buffer design. *IEEE Computer*, pp. 6–22, January 1984.
8) S. MacFarling. Combining branch predictors. Wrl technical note tn-36, Digital Western Research Laboratory, 1993.