An Improved Dispatch and Issue Mechanis with Dynamic Instruction Reuse

Yi -Ming Chen Chang-Jiu Chen

Department of Computer Science and Information Engineering
National Chiao Tung Universit

ABSTRACT

Among researches about improving performance on superscalar processor, an idea is emerged and it is named dynamic instruction reuse. This concept will help improving the performance and the utilization of the resource which can be reduced at the same time. In this paper we propose an improved dispatch and issue mechanism with dynamic reuse in superscalar. The percentage of average reduction of total number of cycles is from 12.5% to 15.2% by using dynamic instruction reuse and trivial computation

Index terms : Dynamic Instruction Reuse, Instruction Dispatch, Superscalar

1. Introduction

Empirical observations suggest that many instructions or groups of instructions be executed over and over again with the same input. Such instructions do not have to be executed repeatedly. By buffering the res ult of the instruction executed previously, future dynamic instances of the same static instruction can use the result if that the input operands in both cases are the same.

Dynamic instruction reuse checks reuse conditions at run-time. It is obvious that the number of instructions executed dynamically can be reduced if instructions that are going to produce the same value can be reused repeatedly. Since dynamic instruction reuse can benefit the performance, we would like to apply this concept to the architecture of the superscalar processor.

A proposed micro-architecture with dynamic instruction reuse in superscalar architecture is depicted in *Figure 1*. As we can see in this figure, before dispatching instructions to the instruction window, we should check whether the instructions could be reused. If so the instruction is not dispatched to the instruction window and just bypass it (Instruction Window), and proceeds directly to the Reorder Buffer (ROB). If the instruction can not be reused, it would be dispa tched to the instruction window for later execution.

This paper contains 5 sections. In section 2, we will talk about our surveys on dynamic instruction reuse. In section 3, we will propose a improved dispatch mechanism for the simplified superscalar archi tecture, which applies the concept of dynamic instruction reuse, as depicted in *Figure 1*. In section 4, we will show the simulation results, and analyze the simulation result to show that the effect of dynamic instruction reuse is worthy to develop this c oncept and the proposed architecture works. In section 5, we will make some conclusions on this topic, and talk about the future work we are going to work out.

2. Dynamic Instruction Reuse and Related Survey

In this section, we will first show the phenom ena of dynamic instruction reuse, and then introduce the schemes, which developed in [6].We will also talk about the position of dynamic instruction reuse and at last the generi microarchitecture of dynamic instruction reuse.

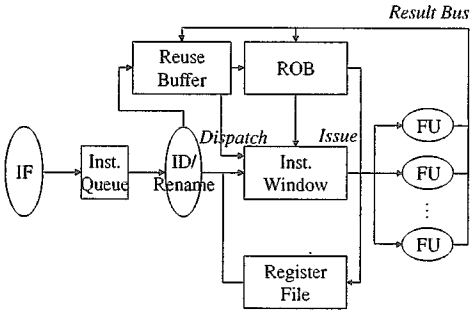

Figure 1: Superscalar Architecture with dynamic instruction reuse mechanism

2.1 Phenomena of Dynamic Instruction Reuse

Sodani mentioned two scenarios in [6]. The first scenario involves speculative execution in a dynamically scheduled processor. This scenario is termed *squash reuse.* [6] Another scenari is *general reuse.*[6]

2.2 Schemes for Dynamic Instruction Reuse

In this section, we introduce three hardware schemes to implement dynamic instruction reuse , which was firstl proposed in [6]. In each scheme we store the results of a previously executed instruction in a hardware structure called Reuse Buffer (RB), which is illustrated in *Fig 2* [6].

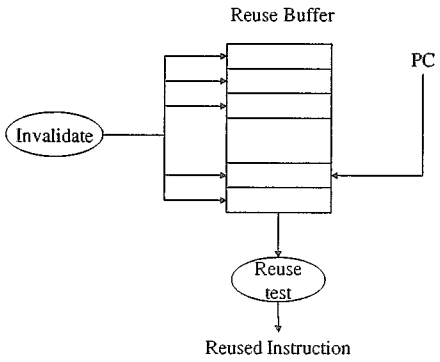

Figure 2 : Generic Reuse Buffer with invalidation mechanism and reuse test, which indexed by PC

As in [6], scheme S_v is a straightforward implementation of the reuse concept. The operand values of an instruction are stored along with its result.

Rather than store operand values, we store operand (architectural) re gister identifiers in the RB. When an instruction writes into a register, all instructions with a matching (source) register identifier in the RB are invalidated. This scheme is called S_n.

Scheme S_{n+d} extends scheme S_n by attempting t establish chains of dependent instructions, and to track the reuse status of such instruction chains.

2.3 A simplified Superscalar Architecture with Dynamic Instruction Reuse

Figure 1 shows a generic microarchitecture with a RB.At this point, the RB is accessed to see if a reusable result for the instruction can be found. Loads bypass the IW only if both micro-operations, address calculation and the actual memory operation, can be reused.

Since the RB contains state that will determine the outcome of future instructions, it needs to be maintained precisely (just like a register file). For scheme S_v, insertin instructions into the RB speculatively requires no special actions – the reuse test ensures that the correct result is obtained. For scheme S_{n+d}, the RST controls the reusabilit of instructions. Just like the rename map in a superscalar processor, checkpoints of the RST have to be taken when a speculation decision is made, and it has to be repaired in the case of an incorrect speculation.

3.An Improved Dispatch and Iss e Mechanism for Superscalar Architecture with Dynamic Instruction Reuse

According to the experimental evaluation in [6], we can easily see that among all 3 reuse schemes, the S_v performs best in 1024 -entry RB and has almost the same performance with fewer entries of RB while compare to the other two schemes. In this section we will put our focus on the dispatch and issue problems in superscalar architecture

with scheme S_v, and propose a mechanism to solve the problem which incorporate trivial computation with the current architecture. We will also talk about the implementation of the mechanism and evaluate the hardware cost.

3.1 Determine Reusability

In the previous section, we knew that the most important part of dynamic instruction reuse is in the instruction decoding stage. During the instruction decode stage, the fetched instructions are decoded at the same cycle and check if the instruction can be reused, as depicted in *Figure 3* .

First, let's see how to determine the reusability of these 4 instructions. If the instruction can not be reused, it would be sent to the instruction window for later issuing to the function unit, else the reuse buffer would send the result to the reorder buffer for later update.

If the instruction can not be reused, it would be sent to the instruction window for later issuing to the function unit, else the reuse buffer would send the result to the reorder buffer for later update, but there are some problem remaining unsolved. We will see it in the next section.

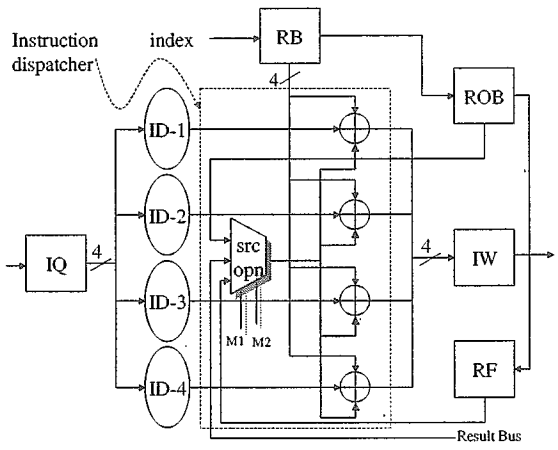

Figure 3: Four instruction decoding with comparison with reuse information

3.2 Rules for Dispatch Mechanism Inside Scheduler

We can find that there is only one problem that will affect the reusability of the dynamic instruction reuse – RAW data dependency. We assume that there are four decoders that could decode four instructions at the same cycle.

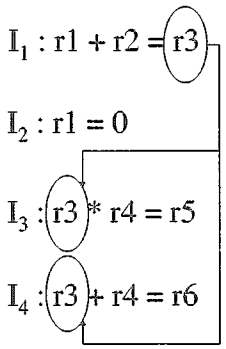

Figure 4: Example of denpendent-instruction-pair

For ensuring the discussions we define the term *'dependent-instruction pair'*. A dependent-instruction-pair is a pair of instructions that has RAW data dependency relationship. In *Figure 4* there are 2 dependent-instruction-pairs, they are I_1 and I_3, I_1 and I_4.

According to the dependence, we can conclude that we need a scheduler to handle these cases of RAW data dependency problems as shown in *Figure 5*.

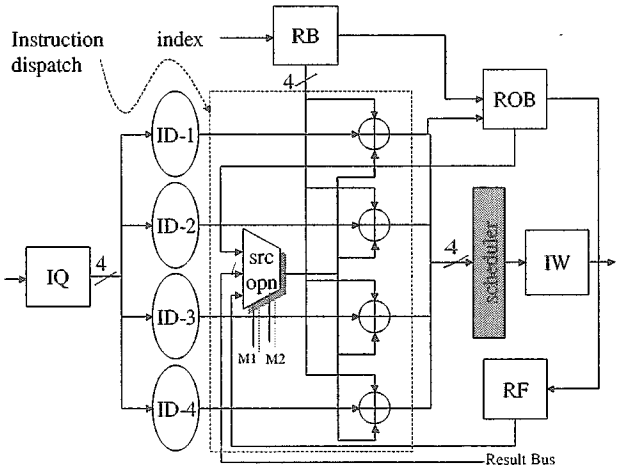

Figure 5: Scheduler for handling dispatching problems

The rules to handle all these conditions are shown as foll ws:

1. If I_1 is not reusable, then check I_2, I_3, I_4 to see

whether any of them has RAW data dependency relationship with $_1$, if so that instruction is dispatched to the instruction window with $_1$. Go to 3.

2. If I_1 is reusable, then go to 3.
3. If I_2 is n t reusable, then check I_3, I_4 to see whether any of them has RAW data dependency relationship with $_2$, if so that instruction is dispatched to the instruction window with $_2$. Go to 5.
4. If I_2 is reusable, then go to 5.
5. If I_3 is not reusable, then check I_4 to see whether $_4$ has RAW data dependency relationship with I_3, if so that instruction is dispatched to the instruction window with $_3$. Go to 7.
6. If I_3 is reusable, then go to 7.
7. Collect all the instructions that should be dispatched and send them to the instruction window

3.3 Trivial Computation Incorporated

To reduce the number of instruction dispatching to the instruction window, checking the trivial operation is another option. We adapt three conditions for triviality as illustrated in *table 1*.

Operation	Conditions for trivialit
Multiply x*	(x or y) = (0, 1, or –1)
Division x/y	(x = y, x = -y, or x = 0)
Square root $x^{1/2}$	(x = 0 or x = 1)

Table 1 Conditions for triviality.

The structure of the trivial operation incorporated with dynamic instruction reuse in the instruction dispatchin stage is depicted in *Figure 6*. We can see in the *Figure 6* that the instruction is decoded and get the source from reorder buffer, result bus, or register file. The value of source operand is then compared with the reuse buffer for the reuse test and with the constant 0 and 1 for the trivial operation test. After these 2 tests, we can determine the source select from trivial operation or reuse test to the scheduler and to the reorder buffer.

All these we will have experimental evaluation with simulation latter — the reuse scheme part and with trivial computaion will be presented in the next chpater.

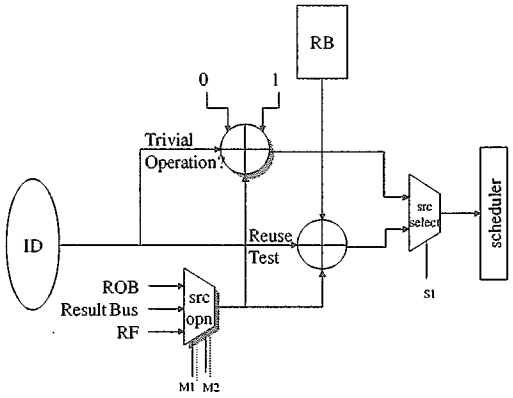

Figure 6: Trivial operation test incorporated with dynamic instruction reuse test

3.4 Reuse Buffer

It is obvious that the most important part to incorporate dynamic instruction reuse with the current superscalar architecture is the reuse buffer.

We know that the general registers would be used for instructions are integer type, long integer type, single precision type and double precision type. We assume that the integer and single precision type register are 4-byte long and long integer and double precision types are 8 byte long. According to this assumption, we know that if we want to use a general reuse buffer, we have no idea which entry should be reserved for the 4-byte one or the 8-byte one. In this point of view, the source operand fields and the result operand field are at least 8-byte long, that is to say, it is waste!

Here we calculate the size of reuse buffer in 2 cases

I. General RB

If we use the general RB for all the instructions, we need all the fields together to determine. But in fact the

load/store instruction and non-load/store instructions would use exclusive fields, one way to reduce this waste is t compact them together, but determination of the fields would be more difficult.

We assume that we have 2^k-entry RB and the address is 32-bit long. The tag field should record the rest of th e program counter, so we should reserve 32 -k bits for it. These two source operand value fields should reserve 64 bits for all instruction, and so does the result field. The Address field we have the 32 bits, which is the same as we assume above. The memory Valid bit only needs one bit. The total amount of one field is 257-k bits, which depicted in *Figure 7*.

II. Speciall -designed RB

We have mentioned above that the load/store instructions and non -load/store instructions use the exclusive fields, if we depart them into two classes, we may save some buffer space. As we can see in the *Figure 8* that the upper one is for non -load/store instructions and the lower one is for the load/store instructions. The total size can be easily added up. But there is something needs to be concerned: how may entries should we reserve for load/store instructions and how many entries should be reserved for the other? This problem should be considered by experimental evaluation and statistic results.

With this information, we can d istribute them apart, but our focus is not on this point, in our simulation we use a general RB for our simulations.

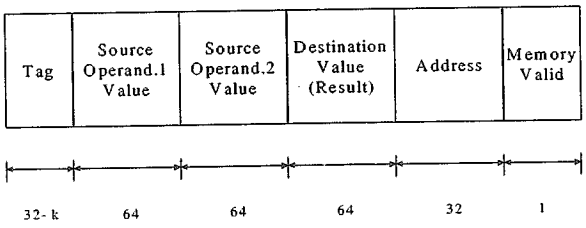

Figure 7: Size allocation of every field of a general RB entry Specially-designed RB

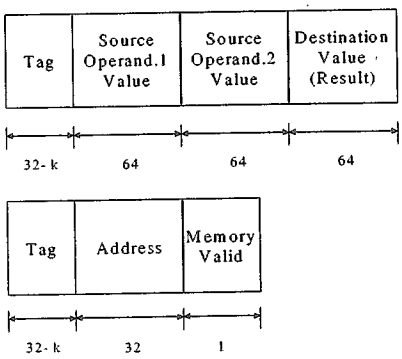

Figure 8 Size allocation of every field of a specially-designed RB entry

3.5.1 Source Operand Value Comparator

While comparing the value of source operand with that of reuse buffer, a comparator is necessary. The comparator can be easily handled by *xor* (*exclusive-or*) gate. If the source operand are 32-bit long, then the comparator need 32 *xor* gates, while all the 32 bits are compared, we collect the result of *xor* gates and '*or*' them together. If the result is 0, then we can make sure that the source operand value of current instruction is equal to that of entry in RB.

Of course, if the source operands are 64-bit long, we need the double size of the *xor* gates for the comparator, and the logic diagram of the comparator is illustrated in *Figure 9*.

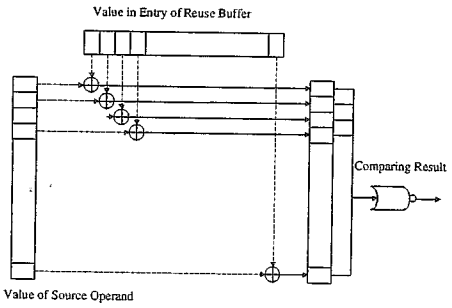

Figure 9: Comparator of current instruction source operand value with reuse information

4. Experimental Evaluation

Our simulator is built on top of the SimpleScalar toolset [5] an execution-driven simulator based upon the MIPS -I ISA. A complete documentation can be retrieved at the web site in University of Wisconsin-Madison: http://www.cs.wisc.edu/~mscalar/simplescalar.html. We also adapt this document for this section.

4.1 Simulation Design Consideration

In our simulator, we put our focus on total instruction count and total number of execution clock cycles reduced by instruction reuse and by hybrid method of dynamic instruction reuse and trivial computation. The main purpose of trivial operation is to reduce the number of execution clock cycle rather than the reduction of instruction count, so the simulation i focusing only on the number of clock cycles. We would like to see how the IPC be affected by the dispatch mechanism. We would also see the effect of replacement policy of the RB.

We would also have simulation on the effect of replacement policy on the RB. In this part, we would use the simpler version of SimpleScalar toolset fo the sack of speed consideration. In this simple version, all we simulation is based on the instruction count.

Of course, reuse scheme can reuse multiple dependent results in a single cycle (the maximum length of a dependence chain reused in a cycle is equal to the read bandwidth of RB, which is 4 in the simula ted configuration). This configuration of the RB, though aggressive, allows us to study the concept of instruction reuse without been limited by any particular implementation.

4.2 Simulation Environment and Configuration

The base simulator models in detail a 4-way dynamically-scheduled processor with its first level of instruction and data cache memory. The parameters for the base out-of-order simulator are listed in *Table2*.

We extended this base simulator to incorp orate the RB and the three instruction reuse schemes described earlier. The RB is integrated with the processor pipeline as described in section 4. In our simulations, the RB is capable of supporting 4 reads, 4 writes, and 4 independent invalidations simultaneously. We also assume that all RB accesses – read, write or invalidate – complete in one cycle.

In [6], we can find that the RB size works best in 1024 entry, and here we assume that the size of RB for our simulation is based on 1024-entry RB.

Instruction Fetch	8/4/2 instruction per cycle. Aggressive: can fetch beyond multiple branches and across cache line boundaries. Fetch stops only on I-cache misses.
Instruction cache	16K-bytes, direct mapped, 32-byte cache line, 6 cycles miss latency.
Branch Predictor	2048 BTB entries with 2-bit saturating counters.
Speculative execution Mechanism	Out of order issue/commit of 4/2 operations per cycle, 32 -entry ROB, 32-entry load/store queue. Maximum of 8 unresolved branches. Loads execute only after the entire preceding store addresses are known. Values bypassed to loads from matching stores ahead in load/store queue.
Architecture registers	32 integer, hi, lo, 32 floating point, fcc.
Function units	4-integer ALUs, 2-load/store units, 4-FP adders, 1-integer MULT/DIV,1 -FP MULT/DIV
Functional unit latenc (total/issue)	Integer ALU-1/1, load/store 1/1, integer MULT 3/1, integer DIV 20/19, FP adder 2/1, FP MULT 4/1, FP DIV 12/12, FP SQRT 24/24.
Data cache	16K 2-way set associative, 32bytes block, 6 cycles mi ss latency. Dual ported, non-blocking interface, one outstanding misses per register.

Table 2 Configuration of our simulation of superscalar architecture with RB

As to the replacement policy, we simulate on both 128 and 1024 entries to see how the replacement policy affects the overall performance.

We have eight benchmarks for the simulation, four of them for integer and the rest for floating-point. *Table 3* shows the average IPC of our base benchmarks. We know that while the instruction be reused, the ins truction would not be sent to the instruction window for the later execution so the IPC is less than the base.

Benchmark / Bandwidth	Compress	Gcc	li	m88ksim
8/4/4	1.717245	0.920209	1.482089	1.641188
4/4/4	1.717789	0.916131	1.495469	1.640979
4/2/2	1.194288	0.75326	1.071679	1.343738
2/2/2	1.194363	0.752487	1.075449	1.369123
Benchmark / Bandwidth	Swim	su2cor	hydro2d	Fpppp
8/4/4	1.231185	1.108873	1.098249	0.731751
4/4/4	1.228325	1.104982	1.092656	0.730234
4/2/2	1.102393	0.966969	0.958151	0.624598
2/2/2	1.108793	0.966552	0.954863	0.623829

Table 3 IPC of each base benchmark

4.3 Simulation Results

4.3.1 Instruction Reused

From *Figure 10*, we can see that the improvement be incorporating trivial computation is little. The most improvement is appeared in hydro2d benchmark, which has 2% of reduction on instruction count, as for other benchmarks there are almost nothing help. As we can see in [4], the percentage is a little better than what we have experimented. The reason is that those instructions that can be classified into trivial computation might be reused in the reuse buffer, so the effect is not so good as we can see in[4].

4.3.2 Speedups

Figure 11 shows the speedups by total number of execution clock cycles obtained due to dynamic instruction reuse. The percentage of average reduction of total number of clock cycles is from 12.3% to 14.6%.

Figure 12 shows the speedups obtained due to dynamic instruction reuse and trivial computation by total number of execution clock cycles. The percentage of average reduction of total number of clock cycles is from 12.5% to 15.2%. *Figure 13* shows the speedup contributed from trivial computation, and we can see that the contribution is not too much. The reason is explained in previous section.

4.3.3 IPC Reduction

We know that while the instruction can be reused, then the instruction would not send to the instruction window for

the later execution, so some of the previously scheduled instructions would not be executed, and the IPC must be less than the base. Here we can see in *Figure 14* that the reduction of five in eight benchmarks is less than 4%, and one in eight is more than 10%.

In *Figure 15*, we can see that the reduction is not s serious than that in *Figure 14*. We think that the main reason for this is that the main purpose of trivial computation is to reduce the execution cycle rather than instruction count, so the effect of trivial computation will increase the IPC. That's why the IPC is a little better than that with only dynamic instruction reuse.

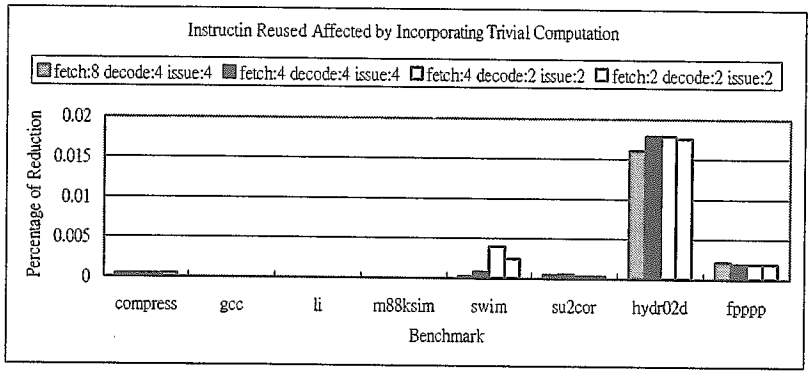

Figure 10: Percentage of instructions reused affected by incorporating trivial computation for 1024 -entry RB

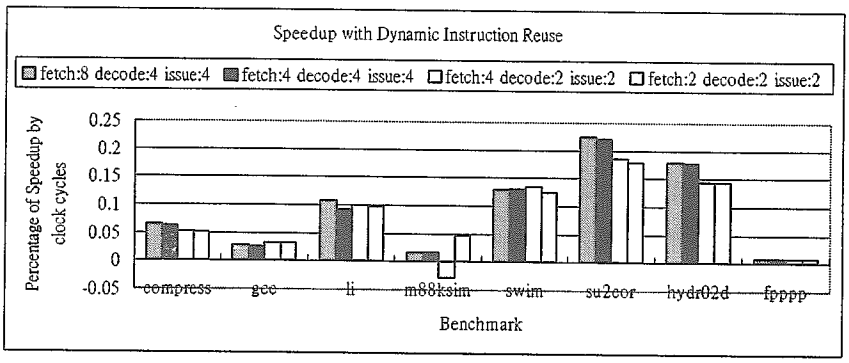

Figure 11 :Speedups obtained due to dynamic instruction reuse measured by total number of clock cycles

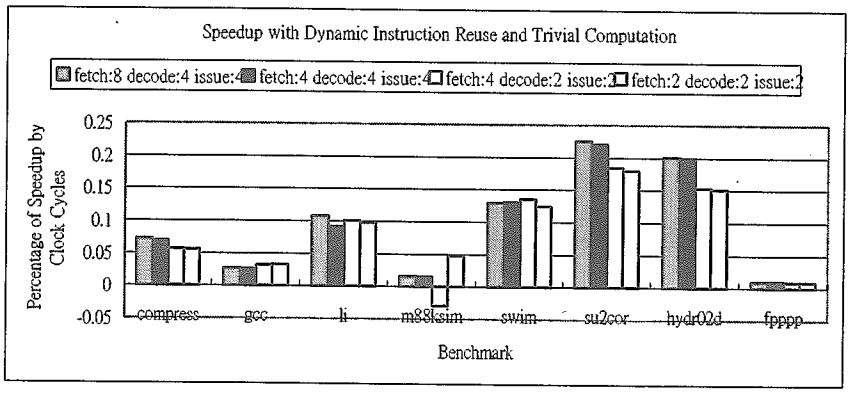

Figure 12: Speedups obtained due to dynamic instruction reuse and trivial computation measured by total number of clock Cycles

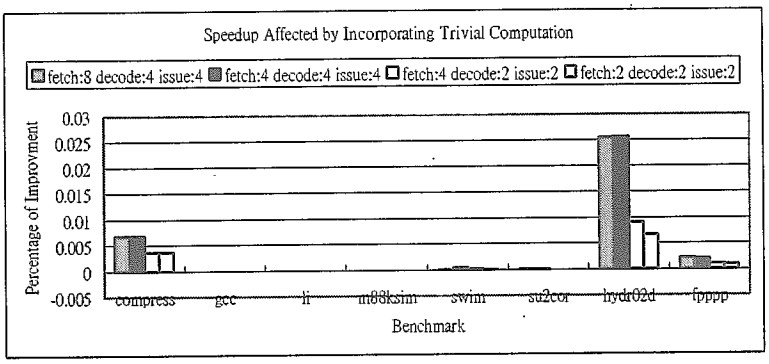

Figure 13 Speedups contributed from trivial computation compared to dynamic instruction reuse measured by total number of clock cycles

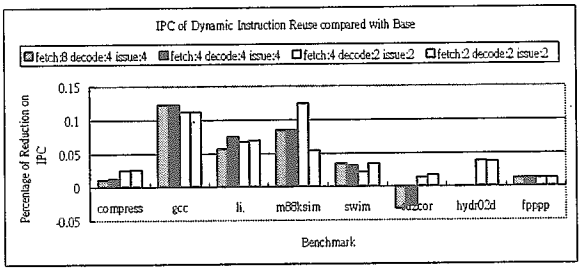

Figure 14: IPC reduction resulted from dynamic instruction reuse.

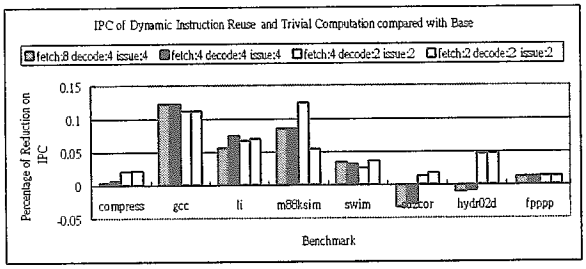

Figure 15: IPC reduction resulted from dynamic instruction reuse and trivial computation

5. Conclusions

In this paper we have introduced the concept and problem of dynamic instruction reuse and trivial computation incorporating into the current design of general superscalar architecture. According to the experimental result in [6], we observed that the size for RB would affect the effect of dynamic instruction reuse. Thus if the processor itself can provides enough area to locate the larger-size RB, not only the instruction count but also the total number of execution clock cycles would decrease significantly.

In our experiment, we show that the improving dispatch mechanism with dynamic reuse works and its improvement on instruction number is also significant. The other benefits of this mechanism is that it collapses the RAW data dependencies, and thus reduce the number of clock cycles that would be wasted without this mechanism.

The most important contribution of this dispatch mechanism is that it makes dynamic instruction reuse to be applied to the superscalar architecture and improve the instruction dispatch, which compares with the original design of superscalar architecture. Of course, additional hardware is necessary.

We also show that the replacement policy would not affect the efficiency of dynamic instruction reuse too much and thus would not improve significantly in the instruction dispatch stage.

The future work about this topic can be focused on the static reuse on the function calls, which is similar to the concept of *result cache* in [4]. Also, if we want to apply dynamic instruction reuse to other architecture, such as super-pipelining, VLIW, and etc, it remains a lot of problems to be solved. Our work focuses on the architecture of superscalar, because it is the most common architecture for the present time. Besides, the reuse schemes can be improved in some way, and exploit the reuse phenomenon to find out new schemes for better reuse efficiency.

If instruction reuse can be done in both static and dynamic ways, or even more the combination of these two

ways, the IPC (instruction per cycle) can be as the same as that without dynamic instruction reuse. That is to say, w need better scheduler to predict the reusability at compile time, and thus would help decrease resour ce waste at run time.

References

[1]. A. Aho, R. Sethi, and j. Ullman. "Compilers principles, techniques, and tools." Addison -Wesley, Reading, MA, 1986.

[2]. J. Smith and A. Pleszkun. "Implementing precise interrupts in pipelined processors." *IEEE Transactions on Computers*, 37(5): 562--573, May 1988.

[3]. Mike Johnson, "Superscalar Microprocessor Design", Prentice Hall, Englewo-od Cliffs, NJ , 1991.

[4]. S. E. Richardson. "Caching function results: Faster arithmetic by avoiding unnecessary computation. " Technical Report SMLI TR-92-1, Sun Microsystems Laboratories, Sept. 1992.

[5]. D. Burger, T. M. Austin, and S. Bennett. "Evaluatin Future Microprocessors: The SimpleScalar Tool Set." Technical Report CS-TR-96-1308, University of Wisconsin-Madison,July,1996.(URL:http://www.cs.wi sc.edu/~mscalar/simplescalar.html)

[6]. Avinash Sodani and Gurindar S. Sohi, "Dynamic Instruction Reuse", *Proceedings of the 24TH Annual International Symposium on computer Architectu-re,*