CHAPTER 3 PERFORMANCE-DRIVEN CROSSTALK ELIMINATION AT POST-COMPILER LEVEL

One effect of the process scaling down is that coupling capacitances have grown reciprocal in the square of the scaling factor. This crosstalk effect will not only increase the power consumption but also lengthen the propagation delay. The coupling capacitance between adjacent neighboring wires, such as buses, on the same metal layer induces a very large fraction of the total capacitance. As a result, how to solve the crosstalk problem on buses has become an important issue. Most of the previous proposed crosstalk-eliminating techniques except [17] were all performed at logic level. They have no knowledge on transition sequences and hence assume that all possible sequences will appear on the bus. Hence, the area overhead for codec logic to eliminate crosstalk is very high. We found that the data sequences on an instruction bus are known during the compile time. We are able to know the transition sequences on the bus before execution. This motivates us to study how to eliminate crosstalk effect on an instruction bus for performance improvement using compiler optimization algorithms. Since the elimination techniques, instruction rescheduling and register renaming, are performed at post-compiler level, no hardware (area) overhead is needed.

The post-compiler optimization algorithms proposed in this thesis is based on pre-designed and low-crosstalk instruction op-codes. In this thesis, we will also present how to assign instruction op-codes for reducing the number of crosstalk sequences. A similar work [38] has been proposed for low power instruction bus

designs. The authors of [38] proposed using run-time profiled results to guide the op-code assignment for application-specific processors. A similar approach, namely, utilization of profiled results, is adopted in our low-crosstalk instruction op-code assignment.

3.1. Crosstalk model and problem definition

In this section, we first describe the crosstalk model and then give the problem definition of eliminating crosstalk effect on an instruction bus.

3.1.1 Crosstalk model

In this section, we first give some preliminaries of the definition of crosstalk sequences [14]. Three types of cross-coupling capacitances of two adjacent bit-lines, b_i and b_{i+1}, are defined in [14] as shown in Table 3.1. First, if the two bit-lines remain the same or swing in the same direction from time t_j to t_{j+1}, no coupling capacitance ($0 \cdot C$) is defined. Second, if one bit-line remains the same from time t_j to t_{j+1} and the other swings, one coupling capacitance ($1 \cdot C$) is defined. Third, if the two bit-lines swing in different directions from time t_j to t_{j+1}, two coupling capacitances ($2 \cdot C$) are observed.

Table 3.1: Three types of coupling capacitance of two adjacent bit-lines.

		b_i at time $t_j \rightarrow t_{j+1}$			
		0→0	0→1	1→0	1→1
b_{i+1} at time $t_j \rightarrow t_{j+1}$	0→0	$0 \cdot C$	$1 \cdot C$	$1 \cdot C$	$0 \cdot C$
	0→1	$1 \cdot C$	$0 \cdot C$	$2 \cdot C$	$1 \cdot C$
	1→0	$1 \cdot C$	$2 \cdot C$	$0 \cdot C$	$1 \cdot C$
	1→1	$0 \cdot C$	$1 \cdot C$	$1 \cdot C$	$0 \cdot C$

Extending two bit-lines to three bit-lines, b_{i-1}, b_i, and b_{i+1}, we have two more types of crosstalk effects for b_i. When b_i swings, b_{i-1} (b_{i+1}) remains the same and b_{i+1}

(b_{i-1}) swings in a different direction to b_i, three coupling capacitances (*3·C*) are defined. Similarly, when b_i swings, b_{i-1} and b_{i+1} swing in a different direction to b_i, *4·C* crosstalk is defined. Figure 3.1 shows two data sequences that result in *3·C* and *4·C* crosstalk on b_i. It is intuitive that when *3·C (4·C)* crosstalk is incurred during the bus transmission, three (four) effective coupling capacitances need to be charged for the signal transmission. Thus, the large coupling capacitances have great effects on the bus transmission delay [14].

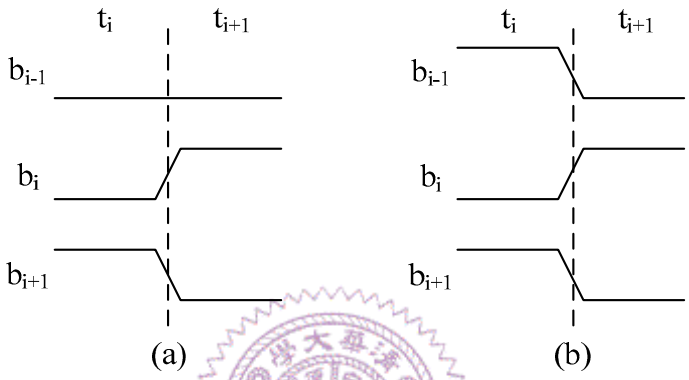

Figure 3.1: Examples of (a) *3·C* and (b) *4·C* crosstalk sequences.

3.1.2 Problem definition

In this section, we present the problem definition. We first define how a processor communicates with an instruction memory. We assume that a Harvard architecture is adopted i.e., the instruction memory and data memory are separated. As shown in Figure 3.2, the processor and instruction memory communicate with each other by an instruction bus and an address bus.

According to the report from NTRS [6], the processor speed will achieve *2* (or *3*) Ghz in 0.07 um technology. That means the instruction fetch stage should be no longer than *0.5* (or *0.33*) ns. On the other hand, the author of [7] pointed out that an optimized delay of system buses will take *0.67* ns in 0.07 um technology based on the report from NTRS. In another word, the fetch stage will be the bottle neck of a

processor, taking into account the delay time of instruction bus, memory access, and address bus in the fetch stage. For example, for some high-performance processors [68][69][70], the instruction fetch stage has been divided into more stages than other pipeline stages. For [68][69], the instruction fetch stage has been decomposed into two stages to adapt the access time. For Analog Devices' BF-561 [70], the instruction fetch stage has been decomposed into three pipeline stages, sending address, accessing, and sending instruction data. Hence, the reduction of delay time of instruction bus, memory access, and address bus becomes very important. In this thesis, we will study the delay reduction of instruction bus by eliminating crosstalk.

The data sequences on an instruction bus are known during the compile time. Hence, it is feasible to analyze the sequence of program codes and then utilize the bit information of instructions to avoid undesirable bit sequences that incur *3·C* and *4·C* crosstalk sequences. Based on this observation, our problem is defined as follows: Given a program (machine code), apply post-compiler optimization to generate a *4·C* (*3·C*-and-*4·C*) crosstalk-free program.

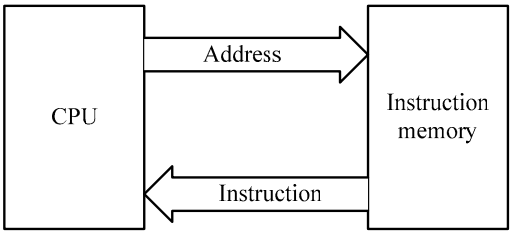

Figure 3.2: The targeted instruction memory architecture.

3.2. Crosstalk elimination in compiler optimization

This section presents the *3·C* and *4·C* crosstalk eliminating algorithms at the post-compiler level. Section 3.2.1 gives an overview of the crosstalk eliminating algorithms. Section 3.2.2 describes the algorithm to reschedule instructions. Finally,

Section 3.2.3 presents the algorithm to rename registers.

3.2.1 An overview of the proposed algorithms

Figure 3.3(a) shows the crosstalk eliminating flow of the proposed algorithms that consists of four steps. The input of the proposed flow is a binary executable program after all the compiler optimizations are performed.

Step 1: We first decompose the input program into basic blocks. Then, we apply two optimization algorithms, rescheduling and renaming, on basic blocks one by one.

Step 2: We reschedule the instruction order of each basic block. By analyzing the dependency graph of instructions in the target basic block and instructions in the adjacent basic blocks, we change the topological order of instructions to eliminate the crosstalk sequences. We model this problem as a traveling salesman problem (TSP). The details will be discussed in Section 3.2.2. The process priority of each basic block is determined by its execution frequency. The higher the execution frequency of the block is, the higher the priority of the block is.

Step 3: We rename register indexes to eliminate the crosstalk sequences. The details will be discussed in Section 3.2.3.

Step 4: After applying Step 2 and Step 3, if there still exists uneliminated crosstalk sequences in and between basic blocks, we will insert redundant instructions to eliminate crosstalk sequences. From the definitions in Section 3.1.1, we can see that an instruction of all zero (or all one) bits adjacent to any other instruction will never incur $3 \cdot C$ and $4 \cdot C$ crosstalk. On the other hand, to maintain the original program behavior, the inserted instruction should have no effect. Fortunately, for many processors, NOP instructions with all zero (or all one) bits are defined. For example, in *Alpha*, the instruction with all zero bits is defined as a NOP instruction. Similarly,

in *ARM*, all one bits can be defined as a NOP instruction. Therefore, in this step, we insert NOP instructions to eliminate crosstalk sequences.

Figures 3.3(b), (c), (d) and (e) show some simple examples of Step 2, Step 3 and Step 4 of the crosstalk eliminating algorithms. Figure 3.3(b) shows a basic block generated in Step 1. In Figure 3.3(b), there are four *4·C* crosstalk sequences (marked by gray rectangles). In Figure 3.3(c), we apply the instruction rescheduling algorithm (Step 2) to the basic block and inter-change *I4* and *I5*. As a result, the *4·C* crosstalk between *I3* and *I4* and *I5* and *I6* is eliminated. In Figure 3.3(d), we further apply the register renaming algorithm (Step 3) to the basic block and rename *R2* (*0010*) to *R3* (*0011*). The crosstalk between *I2* and *I3* is also eliminated. Since the crosstalk sequence between *I1* and *I2* is unable to be eliminated in using the instruction rescheduling and register renaming, in Figure 3.3(e), a NOP between *I1* and *I2* is inserted in Step 4 to eliminate the sequence.

Note that instruction rescheduling and register renaming will not cause any timing overhead. However, the insertion of the NOP instructions increases the total number of dynamic and static instruction count.

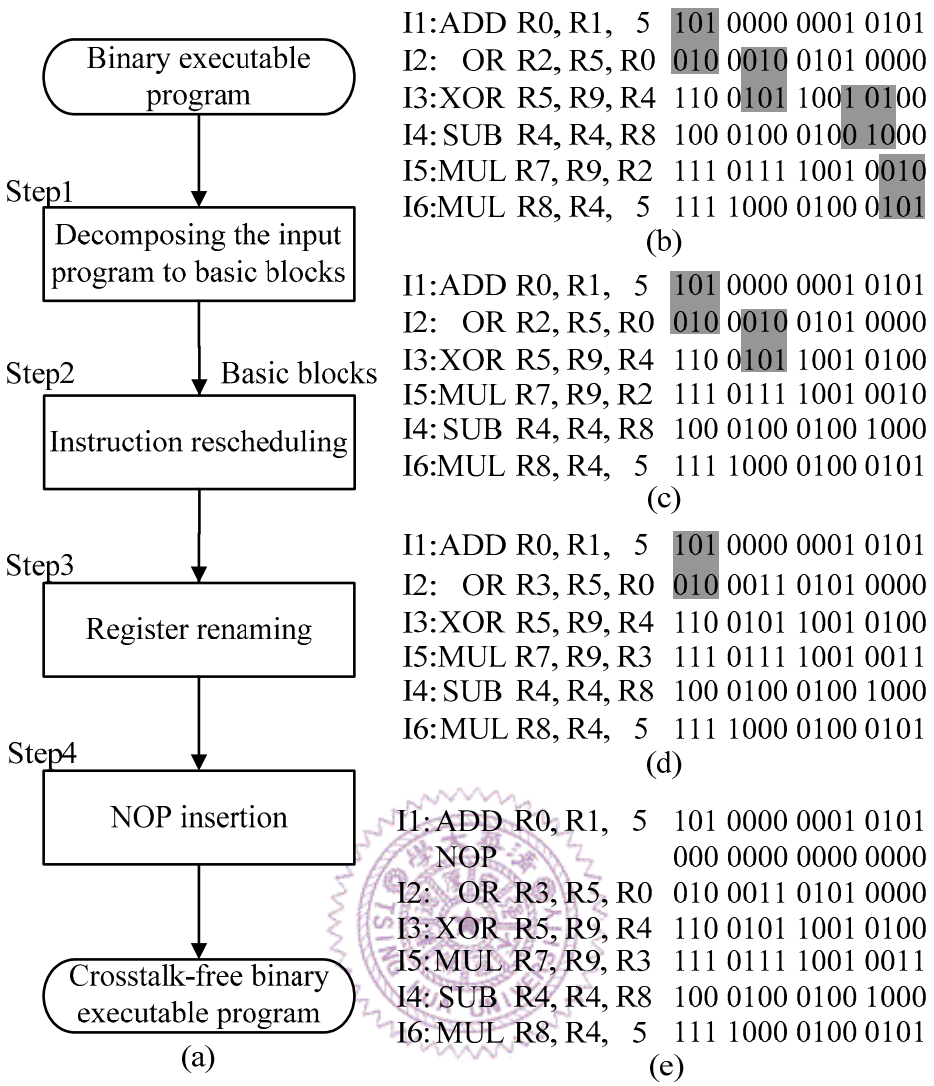

Figure 3.3: Overview of our proposed algorithms: (a) the algorithm flow and (b-e) the algorithm examples.

3.2.2 Instruction rescheduling

During the execution of a program, instructions are fetched and executed one by one. Under the constraint of instruction dependencies, we are able to reschedule the instructions. In this sub-section, we present how to reschedule the instructions in a basic block for eliminating crosstalk sequences. We first construct a control and data dependence graph (CDG) of the instructions in a basic block

Based on the generated CDG, a weighted and directed rescheduling graph, G_{rs}, is constructed. The directed graph ($G_{rs}=\{V_{rs}, E_{rs}, WE_{rs}\}$) represents the sequence relations of instructions. V_{rs} is a node set, E_{rs} a directed edge set, and WE_{rs} weights on the edges. A node v in V_{rs} represents an instruction, an edge e from node v_i to node v_j represents that instruction v_i is able to execute before v_j, and the weight on the edge represents the crosstalk cost between two instructions. The weight on each edge from v_i to v_j is defined as the number of fields in which crosstalk exists:

$$Edge\_Weight_{ij} = \alpha \times crosstalk_{unchangeable}(v_i, v_j) + \beta \times crosstalk_{changeable}(v_i, v_j),$$

where $crosstalk_{unchangeable}(v_i,v_j)$ represents the crosstalk incurred by the unchangeable fields of bit-codes, such as op-code fields and immediate values fields in an instruction, and $crosstalk_{changeable}(v_i,v_j)$ by changeable bit-codes, such as register indexes. The weight is a linear combination of the two terms. In our experiment, since the first term, $crosstalk_{unchangeable}(v_i,v_j)$, is unchangeable, we will set α larger than β.

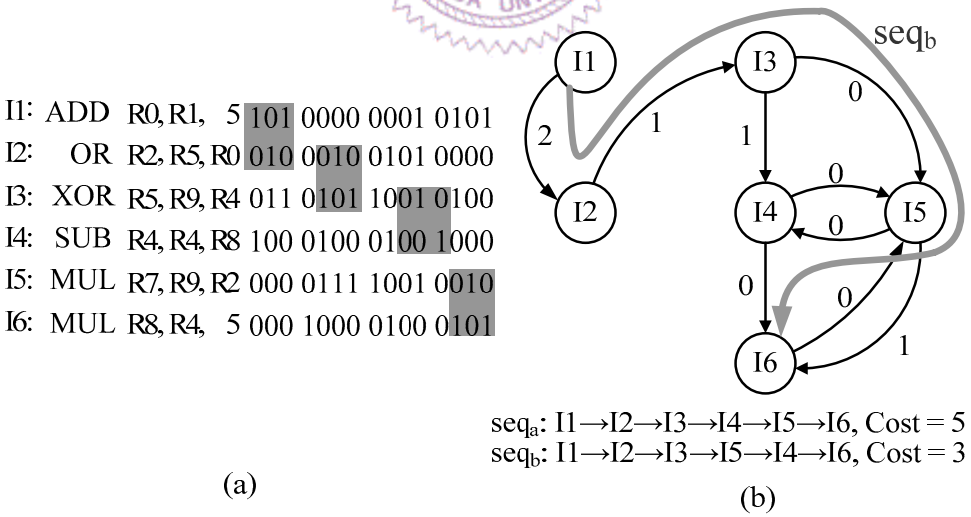

Figure 3.4: A rescheduling graph example: (a) the instruction set and (b) the rescheduling graph.

Figure 3.4 shows a rescheduling graph example. Since there are six instructions in Figure 3.4(a), there are six nodes in G_{rs} (Figure 3.4(b)). The *I1* is the last instruction of the predecessor and *I2-6* are instruction in the target basic block. Since *I1* is executed before *I2*, there is an edge from v_{I1} to v_{I2}. Similarly, since there is no dependency between the two instructions *I4* and *I5*, there is an edge from v_{I4} to v_{I5} and an edge from v_{I5} to v_{I4}. As to weights on edges, let us take the edge from v_{I1} to v_{I2} as an example and α and β are set to *2* and *1*, respectively. Because there is one $4 \cdot C$ crosstalk between *I1* and *I2* incurred by the unchangeable op-code field, the weight on the edge from v_{I1} to v_{I2} is *2* (2×1). Similarly, the weight on the edge from v_{I2} to v_{I3} is 1 (1×1) because the $4 \cdot C$ crosstalk between *I2* and *I3* appears in the field of register index which is changeable.

Once the directed graph is constructed, a TSP algorithm [39] is applied to obtain an instruction sequence with the least sum of edge weight of

$$Weight\_Cost = \sum_{\substack{\text{for each edge in} \\ \text{the sequence}}} Edge\_Weight_{ij}.$$

In order to consider the crosstalk sequences that are incurred between basic blocks, we develop a rule to select the first instruction of the traveling salesman path. First, for a basic block to be rescheduled, the instruction which does not incur any crosstalk sequence with the last instructions in its predecessor basic blocks has the highest priority to be the starting node of the traveling salesman path. Second, if there is no such instruction, the second priority will be given to the instruction whose crosstalk sequences with adjacent instruction is incurred from register-naming fields. The reason behind this heuristic is that a later step of register renaming may eliminate the crosstalk sequences. Finally, if none of the previous cases is met, any instruction which does not violate dependency relations is selected.

Based on the obtained sequence, a new instruction order is determined. The least weight path corresponds to a legal execution sequence which has the least number of fields with crosstalk. It means the least number of registers needs to be renamed and the least number of NOP needs to be inserted. Take Figure 3.4 as an example. We find a path seq_b has less *Weight_Cost* than the original path seq_a. As a result, new instruction sequence is determined as seq_b (Figure 3.3(c)).

3.2.3 Register renaming

The naming of register also affects the transmission sequences on an instruction bus. Frequently, there are unused register indexes after register naming. It means if one register index incurs crosstalk sequences, we are able to rename it to a different one for eliminating the incurred sequences. In this section, we will introduce how to rename register indexes to eliminate crosstalk sequences.

For renaming register indexes, we model the relations between register indexes as a weighted and undirected graph, which is called an adjacency graph, G_a. An adjacency graph (G_a=$\{V_a, E_a, WE_a\}$) represents the adjacency relations of two symbolic registers that are adjacent in the program code where V_a is a node set, E_a an edge set, and WE_a weights on the edges. A node v in V_a represents a symbolic register, an edge e between node v_i and node v_j represents that symbolic register v_i and v_j are adjacent and the weight on the edge represents the frequency of their adjacencies. Figure 3.5 gives an example of an adjacency graph. In Figure 3.5(a), there are seven symbolic registers and an immediate value, *5*. Hence, there are eight nodes in the graph as shown in Figure 3.5(b). The edges between nodes and the weights on the edges are constructed by tracing the code. In Figure 3.5(a), the two dots show that *S0* and *S2* are adjacent twice. Hence there is an edge between node *S0* and *S2* and the weight of the edge is set to two.

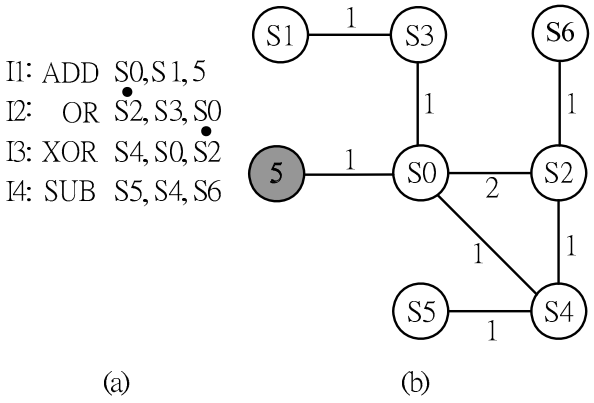

Figure 3.5: An adjacency graph example: (a) the instructions in a basic block and (b) the corresponding adjacency graph.

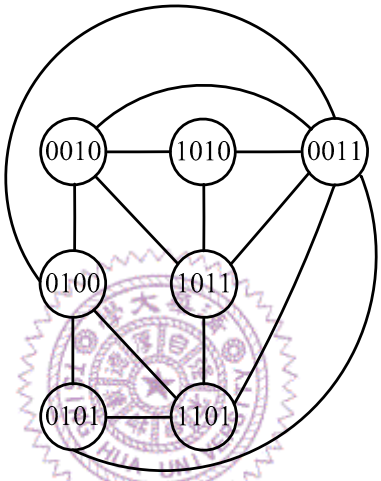

Figure 3.6: A constraint graph example.

After constructing an adjacency graph, an initial solution is determined and then a simulated annealing based algorithm is performed to further improve the initial solution. Before constructing an initial solution, we represent all possible register index candidates as a constraint graph, $G_c = (V_c, E_c)$, where V_c is a node set and E_c an edge set. A node c in V_c represents a register index and an edge between node c_i and c_j represents that the two indexes have no $3 \cdot C$ or $4 \cdot C$ crosstalk with each other. Figure 3.6 shows a constraint graph with seven index candidates where the edges represent that there is no $4 \cdot C$ crosstalk between the two indexes. For example, an edge between register 2 (*0010*) and register 10 (*1010*) represents that there is no $4 \cdot C$ crosstalk between the two nodes.

After constructing a register-adjacency graph and a candidate-constraint graph, we apply a clique partitioning algorithm [40] to partition the two graphs, G_a and G_c. Note that a clique in G_a means that the nodes (the registers) in the clique are adjacent to each other in the program code while a clique in G_c means the nodes (the indexes) in the clique are crosstalk-free with each other. We assign partitioned clusters in G_c as bins and perform the bin-packing algorithm [41] to pack clusters in G_a into clusters in G_c. After packing, we assign the correspondent codes to each instruction. As for the groups that are not assigned, we will assign them randomly. Note that in order to eliminate the crosstalk sequences between basic blocks, special attention is paid to the first instruction of the basic block being processed. Indexes for the registers of the first instruction in the target basic block are assigned first and their indexes are selected so that no crosstalk sequence is incurred between this instruction and the last instructions in the predecessor basic blocks.

Figures 3.7(a) and (b) show the partition results in G_a and G_c, respectively. In G_a, four clusters, $C_{a,1}$, $C_{a,2}$, $C_{a,3}$, and $C_{a,4}$, are generated. In G_c, two clusters, $C_{c,1}$ and $C_{c,2}$, are generated. Then, we pack $C_{a,1}$ and $C_{a,2}$ into $C_{c,1}$ and $C_{a,3}$ and $C_{a,4}$ into $C_{c,2}$. After packing, indexes in G_c are assigned to G_a as shown in Figure 3.7(c).

In the second step, we perform a simulated annealing algorithm [42] to further improve the crosstalk cost function. The crosstalk cost is defined as:

$$\textbf{Crosstalk\_Cost}(G) = \sum_{e} we \times crosstalk_{sequence}(v_i, v_j),$$

where *we* represents the weight on each edge *e* and $crosstalk_{sequence}(v_i, v_j)$ is the crosstalk sequence incurred by the two codes, v_i and v_j.

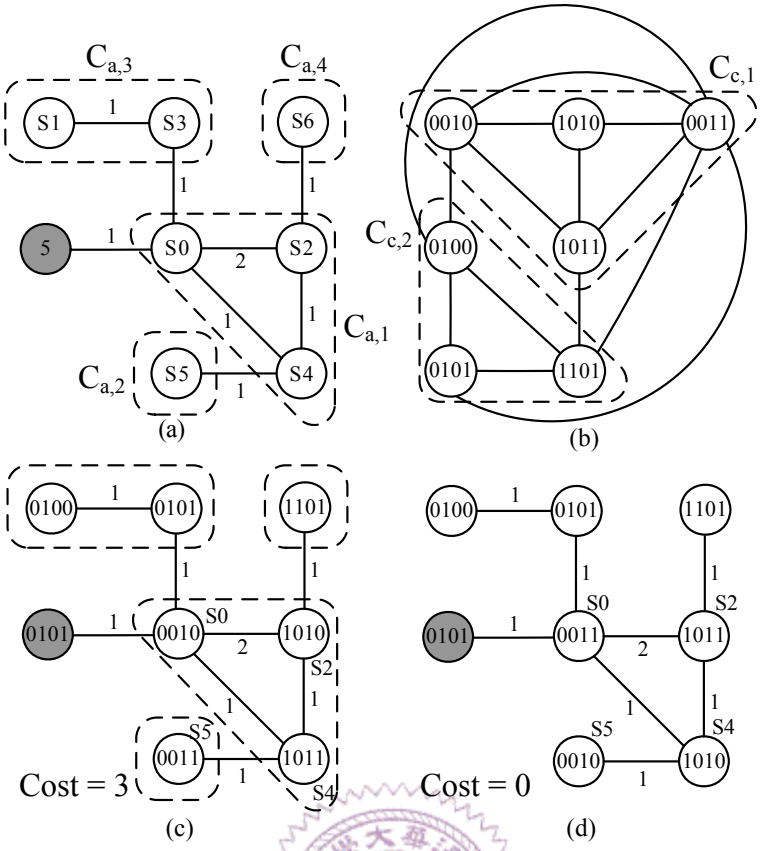

Figure 3.7: A register index renaming example: (a) the clique partition of adjacent graph, (b) the clique partition of constraint graph, (c) an initial assignment of register names and (d) the result of SA interchange.

In each temperature step of the simulated annealing algorithm, we selectively change two indexes in G_a or index candidates. In Figure 3.7(d), we interchange indexes of *S0* and *S5* and indexes of *S2* and *S4* in Figure 3.7(c). The result shows that the cost is improved.

3.3. Instruction op-code assignment

As mentioned in Section 3.2.2, crosstalk sequences on an instruction bus are affected by instruction op-code fields. Since op-codes are fixed, crosstalk sequences incurred by instruction op-codes are not avoidable (except the NOP insertion).

However, the behavior of a system design is usually statically known. Hence, it is feasible to analyze the application programs and utilize their characteristics to

design an instruction op-code in a processor design that will result in less crosstalk. In this section, we present our low-crosstalk instruction op-code assignment algorithm for minimizing crosstalk. First, we collect a set of instruction-execution sequences from target application programs. According to the sequences, we transform the sequence into a transition graph. Then, we utilize the candidate assignment algorithm proposed in Section 3.2.3 to assign op-code candidates to instructions.

To assign op-code candidates to instructions, we model the instruction-execution sequences as a weighted graph G_{ins}. Similar to G_a, $G_{ins} = (V_{ins}, E_{ins}, WE_{ins})$, where V_{ins} is a node set, E_{ins} an edge set, and WE_{ins} execution frequencies on the edges.

Figure 3.8 shows an example of constructing the weighted graph from an execution sequence.

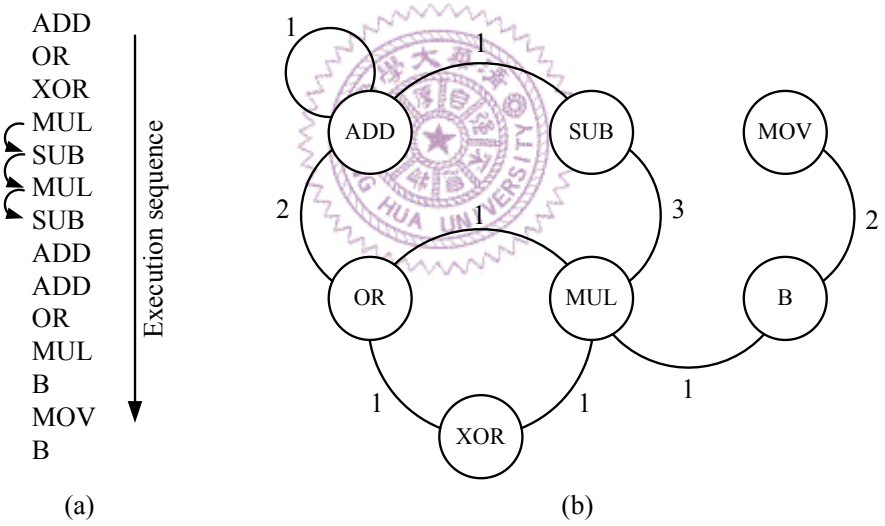

Figure 3.8: An instruction transition graph example: (a) an execution sequence and (b) the corresponding transition graph.

After generating an instruction transition graph, we apply the algorithm proposed in Section 3.2.3 to assign op-code candidates to each node (instruction) in the transition graph.

3.4 Experimental results

We have conducted five sets of experiments on two benchmark sets. In the first experiment, we demonstrate the crosstalk reduction of low-crosstalk instruction op-code assignment algorithm described in Section 3.3. In the second and the third experiments, we examine the effectiveness of our post-compiler optimization algorithms. In the fourth experiment, we examine the energy consumption after applying our post-compiler optimization algorithms and instruction op-code assignment algorithm. In the fifth experiment, we examine the static instruction count, cache miss rate, and algorithm running time. The two benchmark sets used in our experiments are *DSPstone* and *Powerstone*. The *DSPstone* is a benchmark set for basic DSP cores and *Powerstone* contains various general purpose programs. *SimpleScalar* (targeted on *Alpha*) is chosen as our experimental platform. In the *Alpha* instruction set, a six-bit op-code field is designed. For some instruction groups, sub op-code fields are also designed.

3.4.1 The op-code assignment algorithm

In this experiment, we first demonstrate how much the op-code affects the total crosstalk sequences between instructions. Figure 3.9 shows that in average 60.63% and 60.11% $4 \cdot C$ crosstalk sequences occurred in the six-bit op-code field on *DSPstone* and *Powerstone*, respectively. Figure 3.9 also shows a similar result when both $3 \cdot C$ and $4 \cdot C$ are considered (for showing the effect when the coupling capacitances larger than and equal to $3 \cdot C$). The results show that the six-bit op-code field has a significant crosstalk effect on an instruction bus.

Figures 3.10 and 3.11 show the average $4 \cdot C$ and $3 \cdot C$-and-$4 \cdot C$ crosstalk minimization results by applying the instruction op-code assignment algorithm on the

DSPstone and *Powerstone* sets, respectively. In Figures 3.10 and 3.11, the results show that our algorithm can eliminate almost all the *4·C* and *3·C*-and-*4·C* crosstalk sequences related to the six-bit op-code. In Figures 3.10(b) and 3.11(b), the results show that in average 65.93% and 60.19% *4·C* reductions can be achieved after the op-code assignment. In Figures 3.10 and 3.11, the results show the *3·C*-and-*4·C* crosstalk reduction that is not as good as the result of reducing *4·C* crosstalk only. The reason is that, in considering both *3·C* and *4·C* crosstalk sequences, more constraints are considered. When we only reduce *4·C* crosstalk, our algorithm needs only to avoid the sequence of (*010* and *101*) only. However, when reducing both *3·C* and *4·C* crosstalk, our algorithm needs to avoid the four more sequences, (*001* and *010*), (*010* and *100*), (*011* and *101*), and (*010* and *011*). This experiment demonstrates that our op-code assignment algorithm can effectively reduce the crosstalk sequences in op-code fields.

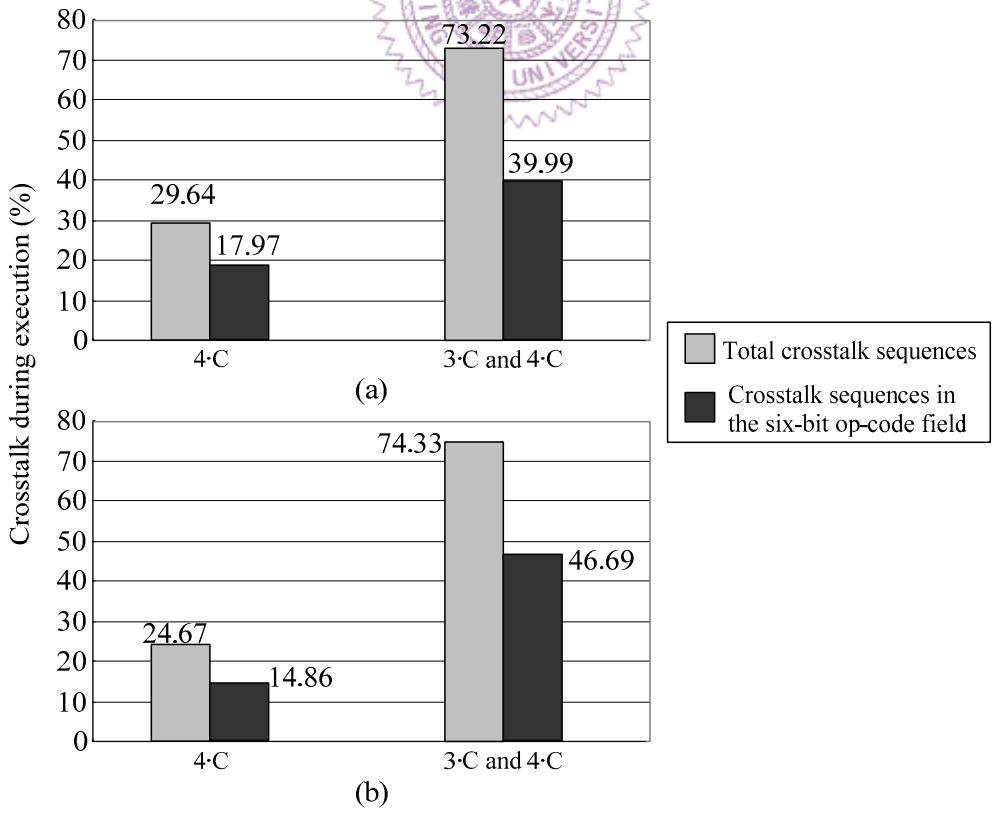

Figure 3.9: The crosstalk effect of the six-bit op-code field on (a) *DSPstone* and (b) *Powerstone*.

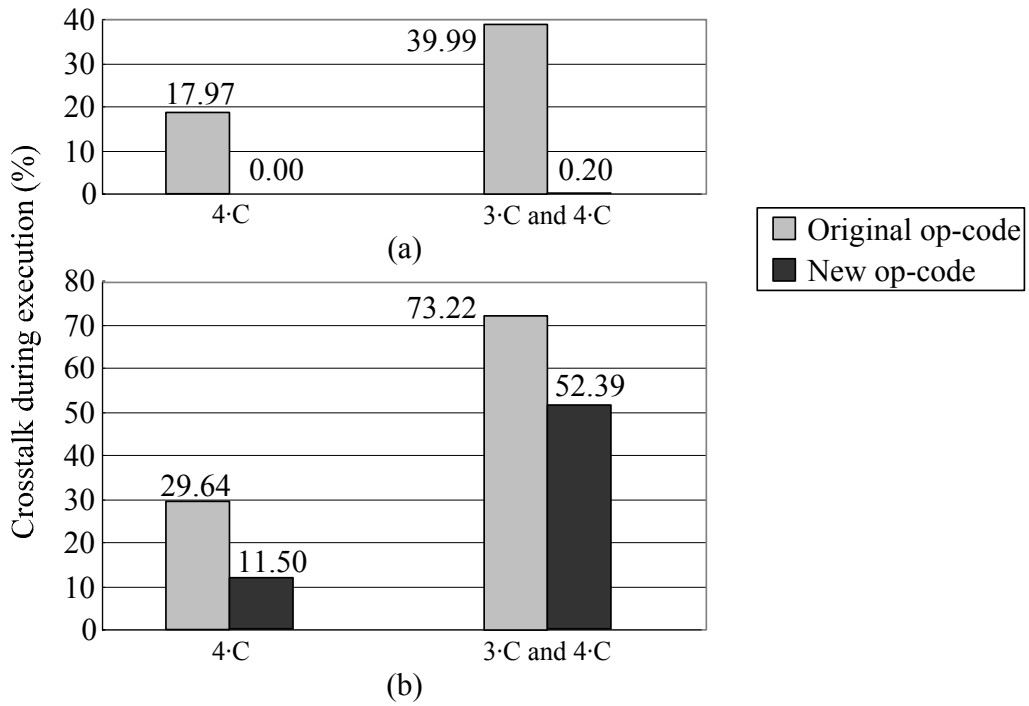

Figure 3.10: The *4·C* and *3·C*-and-*4·C* crosstalk reduction on *DSPstone*: (a) related to the six-bit op-code field and (b) related to both op-code and sub op-code field.

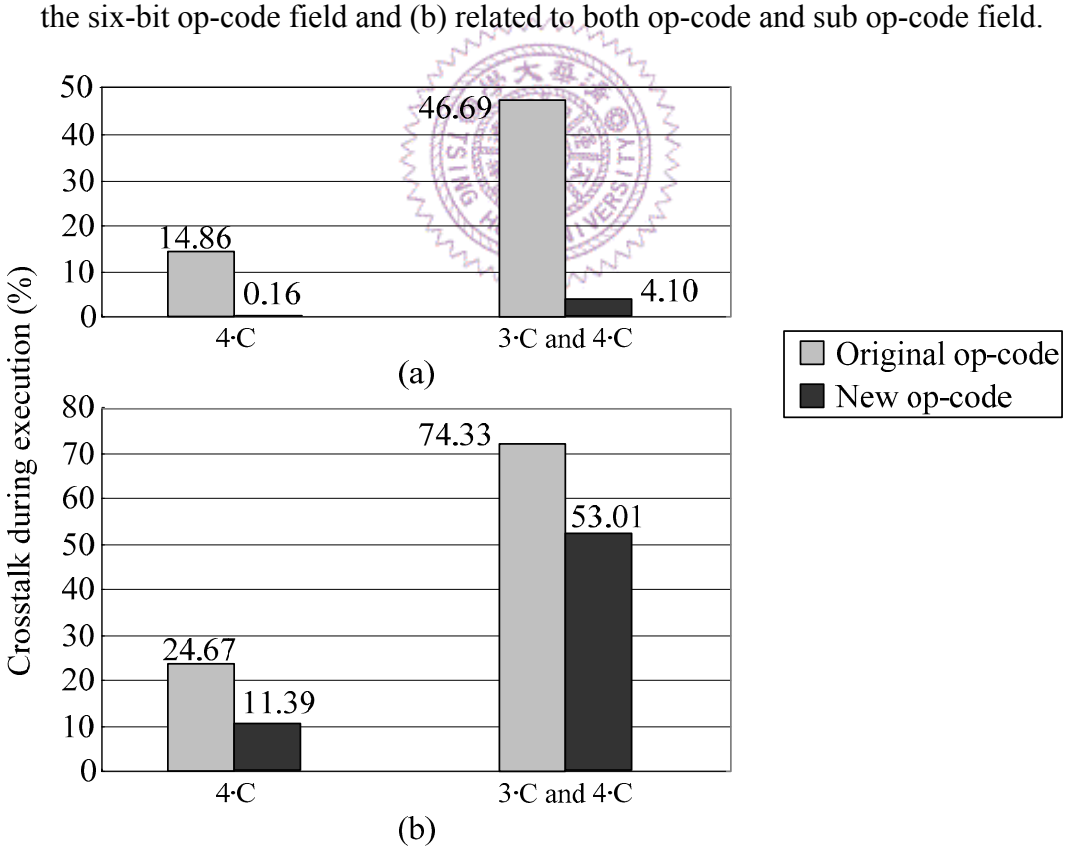

Figure 3.11: The *4·C* and *3·C*-and-*4·C* crosstalk reduction on *Powerstone*: (a) related to the six-bit op-code field and (b) related to both op-code and sub op-code field.

3.4.2 The result on the number of *4·C*/*3·C*-and-*4·C* crosstalk reduction

In this experiment, we demonstrate the effectiveness of our post-compiler optimization algorithms presented in Section 3.2 based on the low-crosstalk op-codes. The α and β in Section 3.2.2 are set to *2* and *1*, respectively.

Figure 3.12 shows the percentages of dynamic instruction count with *4·C* or *3·C*-and-*4·C* crosstalk sequences after applying our post-compiler optimization algorithms. The gray bar shows the percentage of the number of dynamic instruction incurring *4·C* (or *3·C*-and-*4·C*) crosstalk sequences to the total number of dynamic instruction in the original program. The black bar shows the results after applying our post-compiler optimization algorithms in Section 3.2. For eliminating *4·C* crosstalk sequences on *DSPstone*, Figure 3.12(a) shows that without applying our algorithms, an average of 11.50% dynamic instructions incur *4·C* crosstalk sequences. After applying our algorithms, only 0.52% (average) instructions incur *4·C* crosstalk. Figure 3.12(b) shows a similar overhead result (0.65%) on the *Powerstone*. The results for reducing both *3·C* and *4·C* crosstalk sequences are not as effective as reducing *4·C* crosstalk sequences only (average 24.20% and 23.15%, respectively). It shows that it is much harder to reduce both *3·C* and *4·C* crosstalk sequences than to reduce *4·C* crosstalk sequences only.

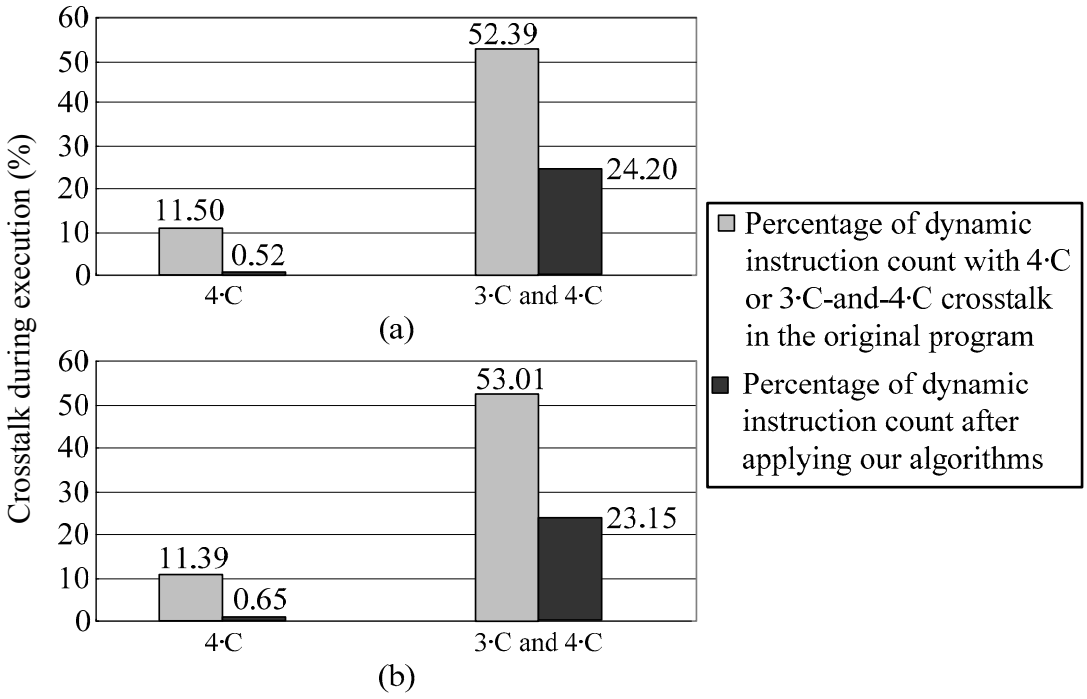

Figure 3.12: The *4·C* only and *3·C*-and-*4·C* reduction comparison on (a)*DSPstone* and (b) *Powerstone*.

3.4.3 The comparison on the instruction-fetch time

In high performance processor designs, pre-fetch unit [58][59] has been proposed to hide the long latency of memory access. The pre-fetch unit is designed to keep fetching instructions from the instruction memory while the program is executed. By improving the performance of instruction bus, we can improve the performance of pre-fetch unit.

Therefore, to investigate the actual timing improvement of a pre-fetch unit, we estimate the total instruction fetch time with/without applying our proposed optimization algorithms. First, we show how *3·C* and *4·C* crosstalk sequences affect the delay of bus transmissions. A set of simulation by *SPICE* [60] was performed. Figure 3.13 shows the worst-case delay among the bus signals under *3·C* and *4·C*. Two process technologies (*0.07* and *0.18 um*) were tested where wiring parasitic was

obtained from Berkeley Predictive Technology Model [61]. Wire length is set to 20 *mm*, which is obtained from [9] and [7]. The data sequences, (*010* and *101*) for *4·C* crosstalk and (*011* and *101*), (*110* and *101*), (*011* and *010*), and (*110* and *010*) for *3·C* crosstalk, are used as the test patterns. In Figure 3.13, we notice that eliminating *4·C* crosstalk sequences on a bus can speed up more than 20% of the bus transmission.

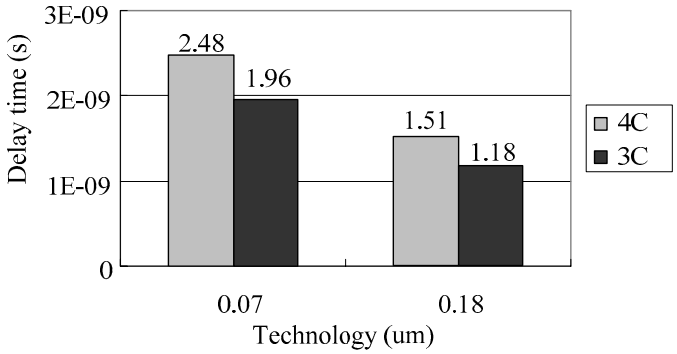

Figure 3.13: The worst-case delay of *3·C* and *4·C* crosstalk conditions.

In the next experiment, we examine the total instruction fetch time comparison of the original programs and the programs after applying our proposed optimization algorithms.

Since the instruction fetch time is composed of three parts [62]: (1) sending address (t_{addr}), (2) memory access (t_{access}), and (3) sending instruction data (t_{inst}), the fetch time of an instruction is estimated as below:

$$fetch\_time = t_{addr} + t_{access} + t_{inst}.$$

Take *0.07 um* as an example, from Figure 3.13, the transmission times on *20 mm* bus considering *3·C* and *4·C* are *1960* and *2482 ps*, respectively. In [63], by using the same *0.07 um* technology, the memory has *333 ps* delay. Hence, the instruction fetch times on a *20 mm* bus considering *3·C* and *4·C* are *4775* and *5297 ps*, respectively. Finally, the total fetch time of all instructions is then computed as below:

$$total\_fetch\_time = fetch\_time \times instruction\_count,$$

where the *instruction_count* is the dynamic instruction count of a program. Similar experiments are performed for the bus lengths of *5*, *10*, *15*, and *20 mm* [9][7] and for the technology of *0.18 um*. Figure 3.14 shows the average improvement of total instruction fetch time after applying our compiler optimization algorithms. The figure shows that after applying our algorithm, in the best case (*20 mm* bus length and *0.07 um* technology), the total instruction fetch times are improved on *DSPstone* and *Powerstone* up to 9.59% and 8.98%, respectively, and in the worst case (*5 mm* bus length and *0.18 um* technology), 1.17% and 0.77%.

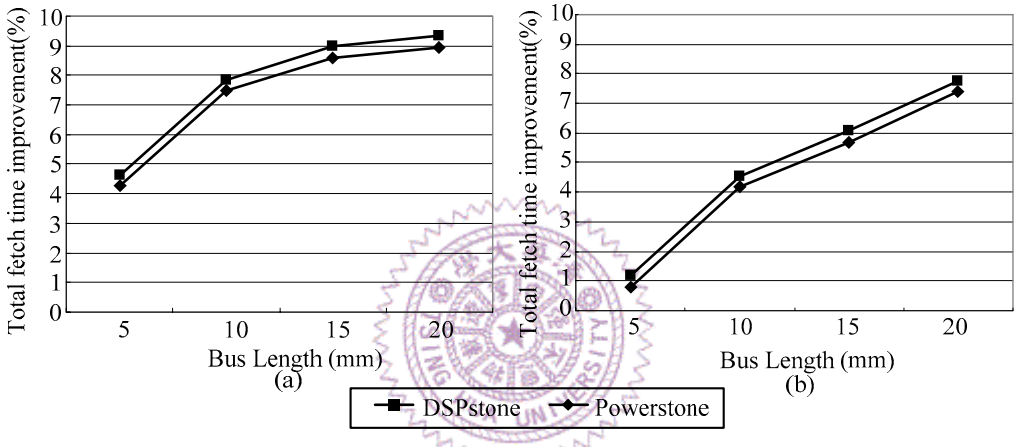

Figure 3.14: The average total fetch time improvement on (a) *0.07* and (b) *0.18 um*.

3.4.4 The comparison on the transmission energy

Although the main object of our algorithms is timing optimization, we would like to understand the effect on energy consumption on this bus. Figure 3.15 shows the average of the total switching number of capacitance on the instruction bus before and after applying the algorithms in Section 3.2 and Section 3.3 for all benchmarks. The C_L and C_X represent the capacitance between a wire and a ground, and the capacitance between two adjacent wires, respectively. We observe that our proposed algorithms, instruction rescheduling, register renaming, and op-code assignment, can reduce the switching number of C_X. However our algorithms do not guarantee the minimization of the switching number of C_L. When all the algorithms and NOP

instruction are applied, the reduction on switching number of C_X is 15%. The switching number of C_L is slightly increased by 2%.

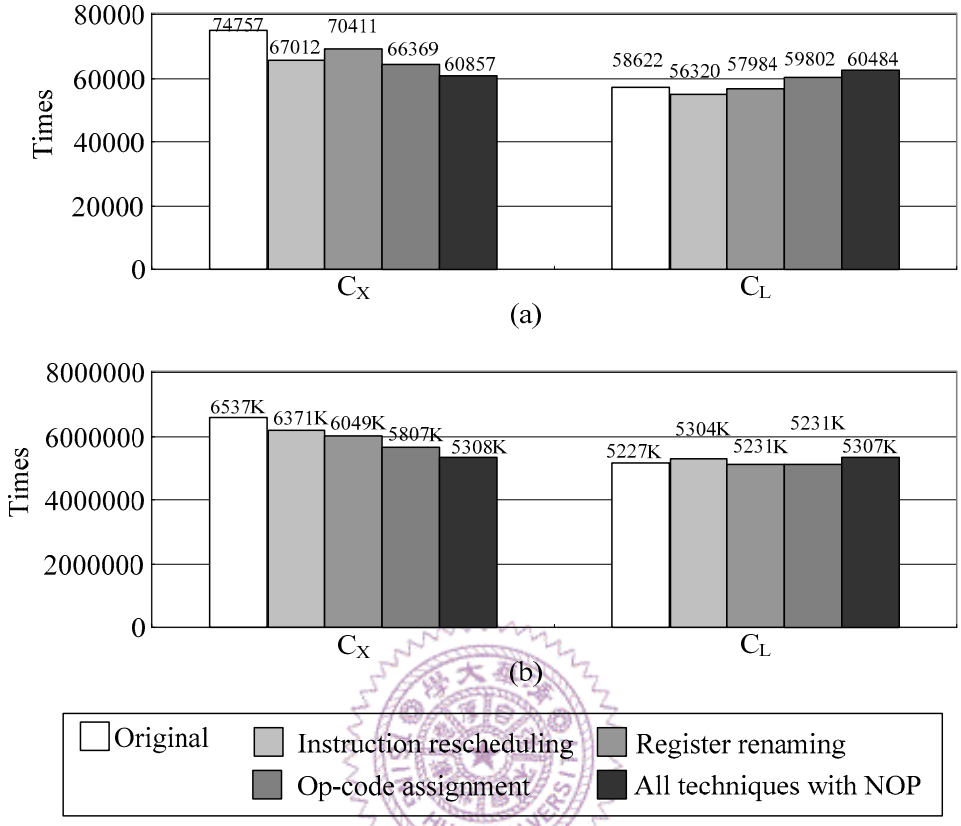

Figure 3.15: The average total switching number of capacitances.

Since this insertion of NOP will increase the power consumption of other pipeline stages. To understand the effect, we perform an experiment on the total power consumption by running the program through the whole pipeline stages. The power is calculated by *Wattch* [64] using 0.18 um technology. Table 3.2 shows the average power consumption before and after applying our algorithms in Section 3.2 and Section 3.3. We can see that the power overhead is 0.44% and 0.61% on *DSPStone* and *Powerstone*, respectively. The power overhead is very small because the number of added NOP instructions is much smaller than the number of instructions in the original program. (Shown in Section 3.4.2).

Table 3.2: The power consumption comparison.

	Original programs(W)	All algorithms with NOP (W)	Overhead
DSPStone	498801	501020	0.44%
Powerstone	13078528	13158921	0.61%

3.4.5 The experiments on the static instruction count, cache miss, and running time

To understand how our algorithms affect the static instruction count and cache miss, we simulate the programs with/without NOP instruction using *SimpleScalar*. The microarchitecutre configuration is shown in Table 3.3. The option to run the simulator is *sim-outorder* with the *in-order* instruction issue. The fetch width, decode width, and issue width are all set to be *1*.

Table 3.3: The microarchitecture configuration of *SimpleScalar*.

Parameter	Value
Branch Predictor	Bimodal predictor (talbe size: 2048) 2-level adaptive predictor (l1size: 1, l2size: 1024, hist_size: 8, xor: 0, mata_table_size: 1024)
L1 instruction cache	16KB, 1-way asc., 32B/line, LRU, 1 cycle lat.
L1 data cache	16KB, 4-way asc., 32B/line, LRU, 1 cycle lat
Instruction TLB	64 entries, 4-way asc., LRU, 30 cycle lat.
Data TLB	128 entries, 4-way asc., LRU, 30 cycle lat.
Memory	8 bytes bus width, 18 and 2 cycles for the first and inter chunks

Table 3.4 shows the average static-instruction count overhead. The results show the added NOP instructions increase the program size by 8%. Although the static instruction overhead is not small, Table 3.5 shows that the increase in cache miss rate is smaller than 1%. This is because our optimization algorithms take the execution frequencies of basic blocks into consideration. As a result, most NOP instructions are inserted in less frequently executed basic blocks.

Table 3.4: The static instruction overhead.

	Original programs	All algorithms with NOP	Overhead
DSPStone	17487.14	18822.93	7.63%
Powerstone	22131.40	24052.37	8.67%

Table 3.5: The cache miss rate comparison.

	Original programs			All algorithms with NOP		
	Access times	Miss times	miss rate	Access times	Miss times	miss rate
DSPStone	6005.14	168.57	2.8071%	6049.14	182.57	3.0181%
Powerstone	455004.85	250.11	0.0549%	457399.85	312.11	0.0682%

The last experiment is to address the running time of our post-compiler optimization algorithms. The post-compiler algorithms are implemented using *C++* language. The platform is *Intel Celeron* 1.3 Ghz with 256Mb ram. The operating system is *RedHat Linux* 7.3. Table 3.6 shows the average number of basic blocks (NB) and average running time of the proposed post-compiler optimization algorithms: instruction rescheduling (IR), register renaming (RR), clique partitioning (CP), bin packing (BP), and simulated annealing (SA). The results show that the running time takes only several seconds. The running time of instruction rescheduling is much larger than that of register renaming. That is because instructions in a basic block are usually larger than the number of registers to be re-assigned.

Table 3.6: The average number of basic blocks (NB) and the running time of instruction rescheduling (IR), register renaming (RR), clique partitioning (CP), bin packing (BP), and simulated annealing (SA).

	NB	IR (s)	RR (s)	CP (s)	BP (s)	SA (s)
DSPStone	3828	1.19	0.31	0.11	0.04	0.12
Powerstone	5017	3.12	0.99	0.28	0.17	0.35