Crosstalk Aware Register Reallocation Method for Green Compilation

Sheng Xiao[1, 2, *], Jing Selena He[3], Xi Yang[4], Yazhe Wang[1] and Lu Jin[1]

Abstract: As nanoscale processing becomes the mainstream in IC manufacturing, the crosstalk problem rises as a serious challenge, not only for energy-efficiency and performance but also for security requirements. In this paper, we propose a register reallocation algorithm called Nearby Access based Register Reallocation (NARR) to reduce the crosstalk between instruction buses. The method includes construction of the software Nearby Access Aware Interference Graph (NAIG), using data flow analysis at assembly level, and reallocation of the registers to the software. Experimental results show that the crosstalk could be dramatically minimized, especially for 4C crosstalk, with a reduction of 80.84% in average, and up to 99.99% at most.

Keywords: Crosstalk, energy efficiency, register reallocation, green compilation.

1 Introduction

With the progress of technological development, the size of embedded devices becomes smaller and smaller, so the bus lines lay out more and more intensively, making the crosstalk a more and more serious challenge for the circuit design. The increments of crosstalk not only influence the scalability and performance of the embedded system, but also consume more power, making the device more vulnerable to overheat and to malicious attacker. The additional power consumption of crosstalk can particularly be used by attackers through the Differential Power Analysis to get the security and hidden information of the system, such as revealing hidden hardware faults on integrated circuits, accessing cryptographic keys, and getting the actual executing codes of the microprocessors [Mangard, Oswald and Standaert (2011); Zhang, Fang, Li et al. (2016); Park, Xu, Jin et al. (2018); Liu, Yarom, Ge et al. (2015)]. Furthermore, the extra power needed increases noise and decreases the lifetime of the embedded device, therefore compromising the current "green compilation" pursuit.

[1] Information Science and Engineering Department, Hunan First Normal University, Changsha, 410205, China.

[2] Computer School, Wuhan University, Wuhan, 430072, China.

[3] Department of Computer Science, Kennesaw State University, Kennesaw, 30144-5588, USA.

[4] Hunan Guangyi Experimental Middle School, Changsha, 410205, China.

[*] Corresponding Author: Sheng Xiao. Email: sxiao@hnfnu.edu.cn.

Received: 29 January 2020; Accepted: 23 February 2020.

Crosstalk is a traditional problem for circuit design and many efforts have been done for it. Circuit designers proposed sorts of methods to reduce the crosstalk between couple buses, such as Codec [Duan, Calle and Khatri (2009); Shirmohammadi, Mozafari and Miremadi (2017)], buffer insertion [Halak and Yakovlev (2010)], Shielding [Mutyam (2009)], gate sizing [Gupta and Ranganathan (2011)], and so on. Lucas et al. [Lucas and Moraes (2009)] evaluated different crosstalk fault tolerant approaches for Networks-on-chip (NoCs) links such that the network can maintain the original network performance even in the presence of errors. Their results demonstrated that the use of CRC coding at each link should be preferred if minimal area and power overhead were the main goals. Cui et al. [Cui, Ni, Miao et al. (2017)] proposed an enhanced code based on the Fibonacci number system (FNS) to suppress the crosstalk noise below 6C level, in which both the redundancy of numbers and the non-uniqueness of Fibonacci-based binary codeword were utilized to search the proper codeword. Experimental results showed that the proposed technique decreased about 22% latency of TSVs comparing with the worst crosstalk cases. Shirmohammadi et al. [Shirmohammadi and Sabzi (2018)] propose DR coding mechanism, which uses a novel numerical system in generating code words that minimizes overheads of codec and is applicable for any arbitrary width of wires. Experimental results show that worst crosstalk-induced transition patterns are completely avoided in wires using DR coding mechanism. Jiao et al. [Jiao, Wang and He (2018)] proposed a crosstalk-noise-aware bus coding scheme with groundgated repeaters. This approach minimized the routing overhead as well as power consumption of data bus systems. The routing overhead was reduced by 12.31% with the new bus coding scheme compared to the conventional data bus with shielding wires. Furthermore, the power leakage and worst-case active power consumptions were reduced by 12.5% and 18.26%, respectively, with the new crosstalk-noise-aware data bus system compared to the previously published bus coding system in an industrial 40nm CMOS technology. Ohama et al. [Ohama, Yotsuyanagi, Hashizume et al. (2017)] proposed a selection method of adjacent lines for assigning signal transitions in test pattern generation. The selection method could reduce the number of adjacent lines used in test pattern generation without degrading the quality of test pattern that could excite the fault effect. Bamberg et al. [Bamberg, Najafi and Garciaortiz (2019)] presented a 3D CAC method which was based on an intelligent fixed mapping of the bits of existing 2D CACs onto rectangular or hexagonal TSV arrangements. Their method required less hardware and reduced the maximum crosstalk of modern TSV and metal wire buses by 37.8% and 47.6%, respectively, while leaving their power consumption almost unaffected. However, these methods either need extra hardware unit support or must increase the area of chip, making them unfavorable for the development of advanced embedded devices requiring portability and minimized cost.

With the existing Selective Shielding method, Weng et al. [Weng, Lin, and Shann (2010)] proposed a co-hardware/software register relabeling combination to reduce the crosstalk of instruction bus. Kuo et al. [Kuo, Chiang and Hwang (2007)] adapted also a combination approach with instruction rescheduling, register renaming, *NOP* instruction padding, and instruction opcode assignment. They proposed the software method to eliminate the 4C crosstalk. These methods are either based on current hardware support or having limitations, illustrated in the next section, to reduce the crosstalk.

Register allocation is also an important component of compilers, many techniques have been proposed, such as Graph-base register allocation [Florea and Geliert (2016); Odaira, Nakaike, Inagaki et al. (2010)], Linear scan register allocation [Poletto and Sarkar (1999); Wimmer and Franz (2010)], tree-based register allocation, and others [Lozano, Carlsson, Blindell et al. (2019), Su, Wu and Xue (2017); Chen, Lueh and Ashar (2018)]. Tabani et al. [Tabani, Arnau, Tubella et al. (2018)] propose a new register renaming technique that leverages physical register sharing by introducing minor changes in the register map table and the issue queue. Experimental results show that it provides 6% speedup on average for the SPEC2006 benchmarks in modern out-of-order processor. Kananizadeh et al. [Kananizadeh and Kononenko (2018)] propose a new class of register allocation and code generation algorithms that can be performed in linear time. These algorithms are based on the mathematical foundations of abstract interpretation and the computation of the level of abstraction. They have been implemented in a specialized library for just-in-time compilation. The specialization of this library involves the execution of common intermediate language (CIL) and low level virtual machine (LLVM) with a focus on embedded systems. But most of these proposed methods were aiming at increasing the performance with litter spill codes, while the crosstalk between instructions was seldom considered.

We propose here a software method Nearby Access based Register Reallocation (*NARR*) to reduce the crosstalk. Though similar to the graph color register allocation method, it is distinguished by combining the frequency of near neighbor access to assign the registers. Our register reallocation approach is not only a software-only method requiring no modifications in hardware, but also improves performance in reducing the instruction bus crosstalk since it deeply analyzes the data flow.

Our contributions of this paper in crosstalk-reducing by software can be summarized:

-A proposed new register reallocation algorithm called *NARR* to reduce the crosstalk. It is a pure software method without any hardware modifications and can theoretically get extra power saving and security enhancing by reducing the crosstalk that register renaming and other software techniques could fail to.

-A modified interference graph called Nearby Access Aware Interference Graph (*NAIG*) is designed and implemented with the help of assemble-level data flow analysis and profiling information that make the register reallocation feasible and more easily.

-Implemention of the algorithm in an evaluation of crosstalk improvement. Results show that the *NARR* is an efficient algorithm in reducing crosstalk, especially in 4C class of crosstalk.

The following Section 2 illustrates the background and motivation firstly, and then introduces our new crosstalk aware register allocation algorithm (*NARR*). Section 3 presents the performance evaluation using a benchmark from Mibench. Section 4 draws some conclusions and highlights future directions.

2 Methods and materials

2.1 Crosstalk overview

Crosstalk is the noise signal for one circuit or channel of a transmission system caused by the other circuits or channels that are usually parallel to the effected one. The strength of crosstalk is often subject to the following factors: wire length, wire width, switching pattern of nearby wires, and so on. For better evaluating the delay and energy caused by crosstalk, researchers have established the crosstalk delay model and energy model as Eqs. (1) and (2) respectively [Moll, Roca and Isern (2003); Duan, Calle and Khatri (2009); Mutyam (2009)].

$$\tau_j = k \cdot V_{dd} \cdot C_{eff,j} \tag{1}$$

$$E_{total} = C_L V_{dd}^2 \left[\left((1+\lambda)\Delta_0 - \lambda\Delta_1 \right) b_{t+1,0} + \sum_{j=1}^{n-2} \left((1+2\lambda)\Delta_j - \lambda\left(\Delta_{j-1} + \Delta_{j+1}\right) \right) \right.$$

$$\left. b_{t+1,j} + \left((1+\lambda)\Delta_{n-1} - \lambda\Delta_{n-2} \right) b_{t+1,n-1} \right] \tag{2}$$

where k is a constant determined by the driver strength and wire resistance, $C_{eff,j}{=}C_L{\cdot}abs(\Delta_j{+}\lambda{\cdot}\Delta_{j,j-1}{+}\lambda{\cdot}\Delta_{j,j+1})$ is the effective capacitance, $\Delta_j{=}b_{t+1,j}{-}b_{t,j}$ is the transmission value of j^{th} line, ΔV_j is the voltage change on the j^{th} line, C_L is the load capacitance, C_C is the coupling capacitance, and $\lambda = C_C / C_L$.

The above formulas show that the different transmission pattern can influence the effect of crosstalk significantly due to their different effective capacitance. According to the effective capacitance of different switching patter of nearby wires in continuous cycles, the crosstalk is classified into six classes shown in Tab. 1. (The symbols -, ↑, ↓, *x* stand for no, positive and negative, any transitions, respectively.)

Currently, research work focuses on how to eliminate the 3C and 4C classes of crosstalk and also to reduce those of other classes. But the established techniques are more or less based on to modification of the integrated circuit that could increase the overhead of the system and therefore can't be used for some cost-constraint embedded systems. There are some software researches attempting to reduce the crosstalk between instruction data buses. Some approaches need to insert extra "*NOP*" instructions. Others can't make use of the full power of changing register because insufficient amount of program information such as data flow is analyzed. In the next section, we will illustrate the limitations of register renaming techniques, as well as how to overcome these limitations by using register reallocation, with a simple example.

Table 1: Classification of crosstalk

Class	$C_{eff,j}$	Transition Patterns$\Delta_{j-1}\Delta_j\Delta_{j+1}$
0	0	x-x
0C	C_L	↑↑↑,↓↓↓
1C	$C_L(1+\lambda)$	_↑↑, _↓↓,↑↑_,↓↓_
2C	$C_L(1+2\lambda)$	↓↑↑,↑↓↓,↑↑↓,↓↓↑,_↑_,_↓_
3C	$C_L(1+3\lambda)$	_↑↓, _↓↑,↓↑_,↓↑_
4C	$C_L(1+4\lambda)$	↓↑↓,↑↓↑

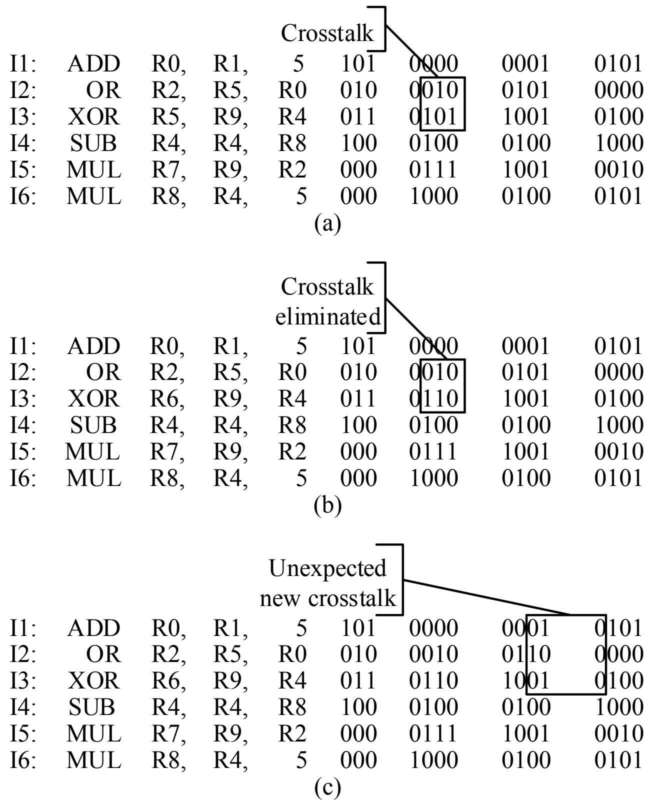
2.2 Crosstalk case study

Register renaming is used as software-only or software/hardware combined technique for reducing the crosstalk. Since the lifetime of registers is not analyzed and the results of register allocation are not used, Register renaming alone can’t alleviate the limitation of register allocation which aims to use a minimal number of registers to generate a good program performance, therefore losing potential improvement in crosstalk reduction.

Considering the example instruction lists in Kuo et al. [Kuo, Chiang and Hwang (2007)], as illustrated in Fig. 1(a), we can see that the instruction scheduling fail to reduce the crosstalk of instruction buses between *I*1 → *I*2 → *I*3. From the *I*2 and *I*3 lines, we can see that *R5* is lastly used of its previous definition in *I*2. And for saving registers, register allocator assigns *R5* for saving the results of *I*3 which causes the 4C crosstalk between *I*2 and *I*3.

From this piece of codes, *R6* is not used and we assume that *R6* is available too. If the register allocator uses the *R6* to replace *R5* for saving the results of *I*3, the crosstalk will be eliminated (shown in Fig. 1(b)). However, if we use the register renaming to rename *R5* with *R6*, the crosstalk between *I*2 and *I*3 will be eliminated, but the new 3C crosstalk will occur between *I*1, *I*2 and *I*2, *I*3 (see Fig. 1(c)).

This example allows us to see the potentially better capability of register allocation in crosstalk reduction, compared to register renaming and to other software techniques.



```
I1:  ADD  R0, R1,  5  101 0000 0001 0101
I2:   OR  R2, R5, R0  010 0010 0101 0000
I3:  XOR  R5, R9, R4  011 0101 1001 0100
I4:  SUB  R4, R4, R8  100 0100 0100 1000
I5:  MUL  R7, R9, R2  000 0111 1001 0010
I6:  MUL  R8, R4,  5  000 1000 0100 0101
                 (a)
```

```
I1:  ADD  R0, R1,  5  101 0000 0001 0101
I2:   OR  R2, R5, R0  010 0010 0101 0000
I3:  XOR  R6, R9, R4  011 0110 1001 0100
I4:  SUB  R4, R4, R8  100 0100 0100 1000
I5:  MUL  R7, R9, R2  000 0111 1001 0010
I6:  MUL  R8, R4,  5  000 1000 0100 0101
                 (b)
```

```
I1:  ADD  R0, R1,  5  101 0000 0001 0101
I2:   OR  R2, R5, R0  010 0010 0110 0000
I3:  XOR  R6, R9, R4  011 0110 1001 0100
I4:  SUB  R4, R4, R8  100 0100 0100 1000
I5:  MUL  R7, R9, R2  000 0111 1001 0010
I6:  MUL  R8, R4,  5  000 1000 0100 0101
                 (c)
```

Figure 1: Register assignments example

2.3 Crosstalk aware register allocation, optimization process outline

In order to get an effective optimization for the total program such as the library of system, our optimization process utilizes the disassemble codes and the profiling results as inputs. Then *NAIG* constructor is used to build the *NAIG* from the disassemble codes and set the weight of it. Finally, the *NARR* processor analysis the *NAIG* to reallocation the register and generate the optimized code. The outline of the process is presented as followed (Fig. 2).

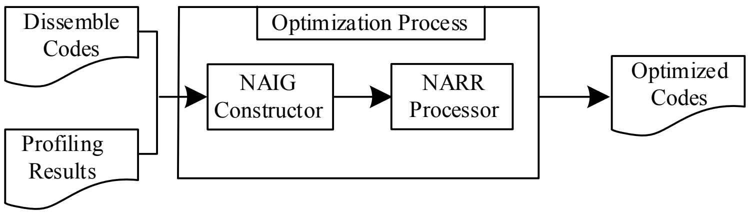


Figure 2: Outline of the optimization process

From this outline, we can see that the kernel of the optimization is *NAIG* construction and *NARR* process. The detail will be presented in the following two subsections.

2.4 NAIG construction

The goal of this work is to reduce the crosstalk on instruction data bus. So the more frequently access patterns of registers pairs are, the more important the registers are. For better illustrating the nearby accesses frequency feature combining with the register allocation, we enhanced the original Interference graph that was widely used for register allocation and constructed the new nearby access aware interference graph, called *NAIG*.

NAIG is a weighted undirected graph that can be represented by a four tuple $G=(V, E_I, E_N, W_E)$. Where $v \in V$ represents a variable or constant of the program, $e(u, v) \in E_I$ expresses that the node u and node v cannot share the same register, $e'(u', v') \in E_N$ expresses that the node u and node v may be nearby access and the weight $w(e) \in W_E$ represents the frequency of such access pattern $e(u, v)$.

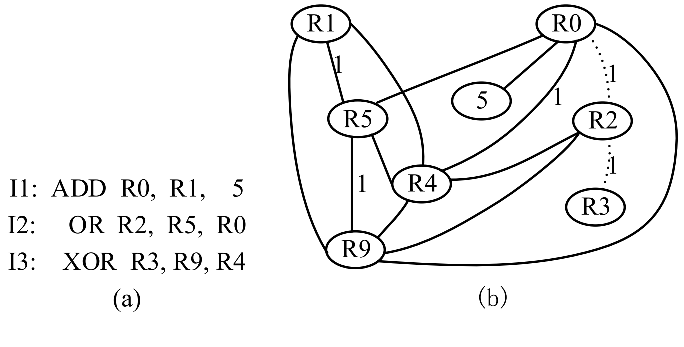


Figure 3: Example of NAIG

For building the *NAIG*, we get the disassemble code as input and suppose that there are unlimited registers, the same as the registers called virtual registers in many compilers.

Firstly, we change the disassemble codes to the *SSA* form for each basic block that makes sure the registers are defined only once (Algorithm 1, line 1-4). Then, we use the methods described in to construct the data flow of each basic block and get the lifetime of each register in each instruction (line 5-6). The interference graph can be constructed of analysis the live register in each instruction (line 7-15). After getting the interference graph, we can use the profiling results to add the weight of edges (line 16-21). And then we return the constructed *NAIG* at last (line 22). The detailed construction algorithm is expressed in Algorithm 1. In this program, the *CFG* is control flow graph for the program and each node $v \in V'$ represents a basic block that contains number of in order executed instructions. The *Liveregi* expresses the register defined before the instruction i and will be used after the instruction that called the live register.

To reduce the cost of spill node is a very complex work because it changes the source order of instruction by inserting extra spill codes that will make the *NAIG* rebuilt. Luckily, in our algorithm, we can avoid to generate the spill codes since the source code is allocated successfully and we can always eliminate the spill code by assigning the spill node with its original one. The detailed reallocation algorithm is presented in next section.

Figs. 3(a) and 3(b) are the changed *SSA* representation and corresponding *NAIG* for the first three instructions of Fig. 1(a), respectively.

Algorithm 1 *NAIG* construction algorithm.

```
Input:
  source disassemble codes, $S$;
  the profiling result, $P: V_P \rightarrow V_P \rightarrow W_E$;
Output:
   $NAIG(V, E_I, E_N, W_E)$;
1: $CFG(V', E') := ConstructCFG(S)$
2: for each $v \in V'$ do
3: Translate $v$ to SSA form
4: end for
5: $DS = DateFlowAnalysis(CFG)$
6: $Livereg_i = GetLivergi(DS)$ based on methods in[1]
7: for each $v \in V'$ do
8:   for each $i \in v$ do
9:     for each $j \in Livereg_i$ && $i \neq j$ do
10:        $NAIG.V.add(i)$
11:        $NAIG.V.add(j)$
12:        $NAIG.E_I.add(i,j)$
13:     end for
14:   end for
15: end for
16: for each pair $< r_i, r_j, w_{i,j} > \in p$ do
17:   $NAIG.V.add(i)$
18:   $NAIG.V.add(j)$
19:   $NAIG.E_N.add(i,j)$
20:   $NAIG.W_E.add(w_i,j)$
21: end for
22: return $NBTI$;
```

2.5 NARR algorithm

Based on the above *NAIG*, we implement our new *NARR* algorithm as follow: firstly, we construct the *NAIG* for each function of the program (Fig. 3); then, we sort the edge of *NAIG* by the decreased weight order (Algorithm 2, line 1). Since the heavily weighted edge represents that the nodes own the edge are more frequently nearby access in instruction date buses, we expect it in the same register or the least crosstalk registers. At the same time, we expect the lowest spill code which will not only lose the performance of the system, but also increase the undetected crosstalk by this algorithm, so we make sure that no additional new spill codes will be emerged in our algorithm.

Then, we analyze the ordered edges one by one to finish the register allocation for each node (lines 2-36). For each edge $e(u, v) \in E_I$, we first check whether a node is assigned. If any one of nodes u is assigned for register r_i, we will choose the register other than r_i but with minimal crosstalk to assign it for v (lines 5-7). If both nodes are not assigned, we first assign any of them to one register and then find the other suitable register as the previous case for the other one (lines 15-19). If the two nodes are assigned with the same register, we will try to change one assigning into another register (lines 6-14). For the edge not in E_I, we first try to assign the two nodes in the same register. If it is not reasonable, we can handle it as the edge in *EI* (lines 22-34). The program detail is shown in Algorithm 2.

Algorithm 2 *NARR* algorithm.

```
Input:
  the $NAIG(V,E_I,E_N,W_E)$ for each function of program;
  the available registers $R=\{ r_0, r_i, \dots, r_n \}$
Output:
  the allocation map M: $V \rightarrow R$ for each node $V$ in NAIG;
1: $E'$ := sort $E_N$ by decreased order in $W_E$
2: while $E' \neq \emptyset$ do
3:   $e(u,v)$ :=pop the first element of $E'$
4:   if $e \in E_I$ then
5:     if only one node ( assuming for $u$ ) is assigned for register $r_i$ then
6:       get the register $r_j \neq r_i$ with minimal cost $crosstalk(r_i,r_j)$
7:       $M.add(v,r_j)$
8:     else if both $u$, $v$ are assigned for the same register $r_i$ then
9:       if one of this two node (assuming for $u$) can be changed to other register
           Set $R'$ without violating the $IG$ of current analysis then
10:          $r_j$:= $r_k$ where $r_k \in R', e(n,u) \in E_N$ and satisfy
               $min \sum_{n=1}^{m} \omega_{n.u} \cdot crosstalk\ (M(n), r_k)$
11:          $M(u)$ :=$r_k$
12:      else
13:        assign the two nodes foe original registers.
14:      end if
15:    else if both $u,v$ are not assigned for any register then
16:      $r_i$:= get the random register that node $v$ can be used.
17:      $M.add(v, r_i)$
18:      get the register $r_j \neq r_i$ with minimal cost $crosstalk(r_i, r_j)$
19:      $M.add(u,r_j)$
20:    end if
```

```
21: else
22: if only one node (assuming for $u$) is assigned for register $r_i$ then
23:   if $r_i$ without violate the conflict of other assignments till now then
24:     $M.add(v,r_i)$
25:   else
26:     assign as line 5-7
27:   end if
28: else if none of node is assigned then
29:     if exist an register $r_k$ can be used for both node without violate the
        conflict of the other assignments till now then
30:       $M.add(v,r_k)$, $M.add(u,r_k)$
31:     else
32:       assign as line 15-19
33:     end if
34:   end if
35:  end if
36: end while
37: return $M$
```

2.6 Experimental setup for performance evaluation

The experiment is built up in Fedora 12 combined with Windows 7 home basic version. The test cases are selected from the MIBench [Guthaus, Ringenberg, Ernst et al. (2001)].

The compiler tools are arm-linux-gcc 4.4.3, and combined with objdump 2.19.51 to get the disassemble codes. The sim-profile tool for arm is used as profile tool to get the access frequency of instructions. The whole experimental framework is shown in Fig. 4.

Firstly, we use the arm-linux-gcc to compile the source code to binary codes in Fedora 12 environment. Then, we disassemble and get profile information for the binary codes respectively. After getting the disassemble codes and profile information, we use them as input for the *NARR* processor to get the crosstalk aware optimized binary codes. Finally, we compare the source binary codes to the optimized binary codes to evaluate the performance of *NARR,* and analyze the improvement details in crosstalk reduction, especially for *3C* and *4C* ones.

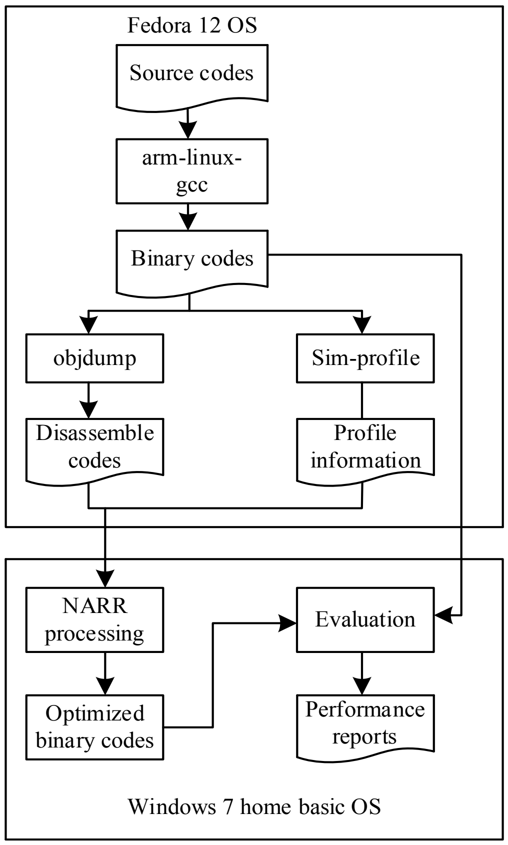


Figure 4: Experiment framework

3 Results and discussions

3.1 Performance of NARR

Fig. 5 presents the decreased percentage of 4C and 3C+4C crosstalk for *NARR,* compared with the results of GCC. From this benchmark, we can see that the 4C crosstalk has been significantly reduced. In the cases such as *stringsearch_large*, *stringsearch_small*, *dijkstra*, and *crc*, the reduction percentage of 4C crosstalk is higher than 95%, eliminating almost all 4C crosstalk of the program. And the average decrease rate is about 81%. And for 3C+4C crosstalk, we can see that most of them are also significantly reduced except *dijkstra* since the *dijkstra* has many conflicts between the 3C and 4C crosstalk. We force a crosstalk avoid priority for 4C, so the *dijkstra* benchmark result is not so good in 3C+4C condition. However, we still get an excellent reduction rate under the major benchmark tests for 3C+4C and the average reduction rate of all tested benchmarks is about 44%.

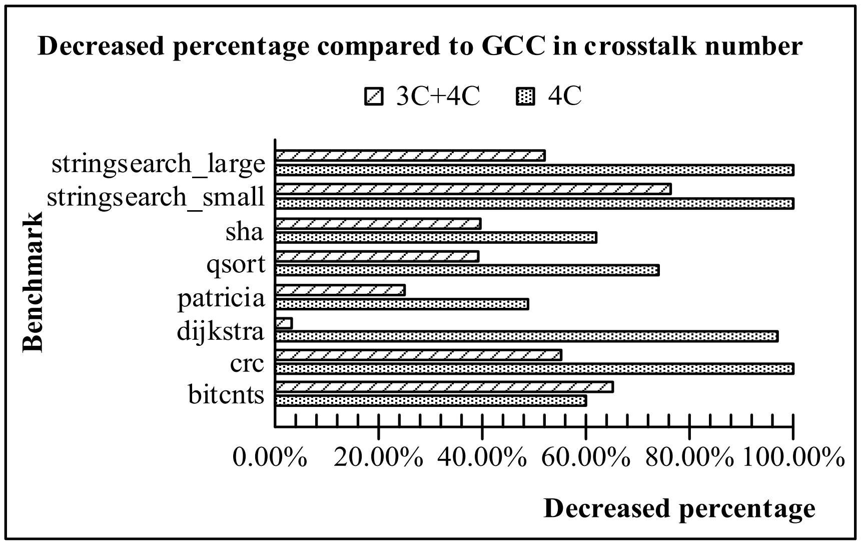


Figure 5: 4C and 3C+4C crosstalk reducing in crosstalk number

For better understanding the crosstalk avoid in instruction level, we also analyze the 4C and 3C+4C crosstalk in dynamic execution with profile recorded in Fig. 6. From this result, we can see that the 4C crosstalk shows again a good reduction and the average decreased percentage is 80.87%. The highest reduction rate is 99.99% for the *crc* test under that only two 4C crosstalk appeared in the program after optimization (shown in Tab. 2). And for 3C+4C crosstalk, the average reduction percentage is also 37.01%, similar to the results shown in Fig. 5. Special cases are, however, again a smaller reduction rate recorded in 3C+4C crosstalk condition for *stringsearch_large, patricia* and *dijjkstra* tests. The main reason could be that in an instruction, there may be some crosstalk in the same class such as 3C. So if the instruction frequently executes, the crosstalk data at instruction level will be less than those at crosstalk number level. However, getting crosstalk statistics at instruction level is reasonable since the program is executed at instruction level and the possible attackers might also try to work in the instruction level to get the most detailed information of the system.

Table 2: Crosstalk comparison of *GCC* and *NARR*

Benchmark	3C Crosstalk			4C Crosstalk		
	GCC	*NARR*	*NARR/ GCC*	*GCC*	*NARR*	*NARR/ GCC*
bitcnts	87750075	5625002	0.06410	33750073	2250001	0.0667
crc	159667218	26611206	0.1667	79833611	2	0
dijkstra	76644109	7562503	0.0987	63344776	86222	0.0014
patricia	590383	38600	0.0654	466640	19041	0.0408
qsort	1250019	400005	0.3199	1050022	100003	0.0952
sha	69017038	11570830	0.1677	39583124	4264487	0.1077
string search_small	16563	15185	0.9168	15509	57	0.0037
string search _large	3881159	352483	0.9247	705765	1334	0.0019

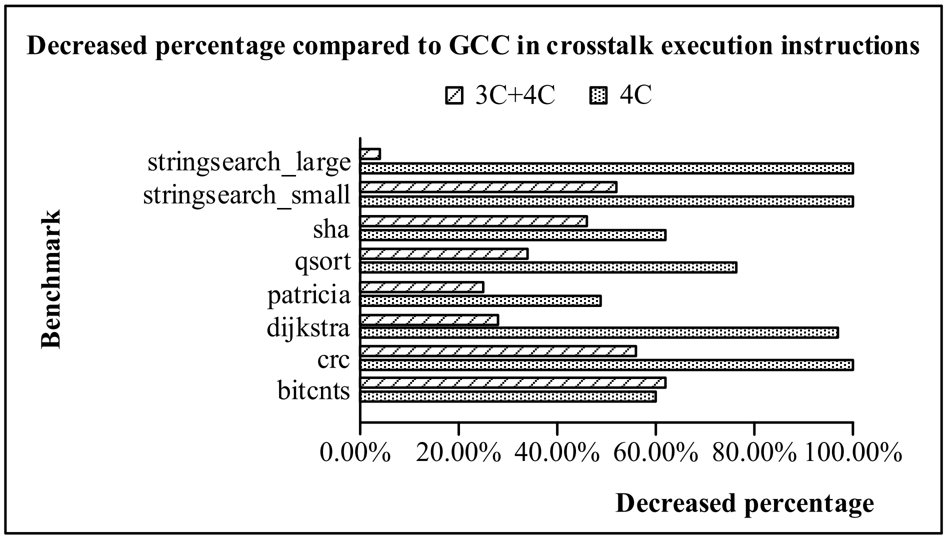


Figure 6: 4C and 3C+4C crosstalk reducing in execution instructions

Tab. 3 shows the evaluation results of adapting *NARR* to reduce the 4C and 3C+4C crosstalk in the aspect of the whole executed instructions. We can see that after *NARR*, the crosstalk percentage is significantly reduced for almost every benchmark tested, in both 4C and 3C+4C cases, in comparison with GCC. The average percentage of 4C crosstalk is reduced to a level of 0.89%, compared with the initially compiled result of 9.89% with GCC (with a relative reduction rate of 91% based on the GCC value). Furthermore, under specific tests such as *crc*, *dijkstra*, etc, we get nearly 0 crosstalk in 4C situation after *NARR*. And the 3C+4C crosstalk is also reduced from 40.77% for GCC to 25.85% after *NARR*, in average. So the *NARR* method is good for reducing the crosstalk, especially for the 4C case.

Table 3: 4C and 3C+4C crosstalk reducing for the whole execution instructions

Benchmark	4C execution in structions		3C+4C execution in structions	
	GCC	*NARR*	*GCC*	*NARR*
bitcnts	0.78%	0.31%	13.01%	5.02%
crc	7.69%	0.00%	53.85%	23.08%
dijkstra	3.08%	0.04%	34.33%	25.86%
patricia	1.51%	0.74%	24.61%	19.00%
qsort	11.27%	2.82%	46.48%	32.39%
sha	8.20%	3.02%	57.10%	31.07%
string search_small	23.10%	0.09%	48.29%	23.68%
string search_large	23.29%	0.09%	48.48%	46.73%
average	9.87%	0.89%	40.77%	25.85%

Tab. 4 shows all types of crosstalk decreased percentage compared to GCC by NARR. We can see that the overall crosstalk is also decreased largely. For the bitcnts and qsort, the reduction rate is up to more than 44%. The average reduction rate is also achieved to 24.24%. So our NARR method is not can get good performance for crosstalk.

Table 4: All types crosstalk reducing in execution instructions

Benchmark	*GCC*	*NARR*	*Decreased rate*
bitcnts	2781573971	1553071408	44.17%
Crc	2261952341	1809561893	20.00%
dijkstra	1606207102	1384640962	13.79%
patricia	12692958	11624053	8.42%
qsort	270214	150181	44.42%
sha	1168409001	935588937	19.93%
search_small_pro	451658	325107	28.02%
search_large_pro	10470186	8885138	15.14%
average	980253428.9	712980959.9	24.24%

4 Conclusions

Crosstalk is a challenge not only for acquiring power-efficiency and performance, but also for satisfying the security and green requirements of an IC design since the nanoscale manufacturing has become the mainstream now. The new method we proposed here in crosstalk aware register reallocation is to reduce the influence of crosstalk for the couple instruction buses. The method is a software-only technique without any needs to modify the traditional hardware. Our *NARR* method can result in a reduction of 80.87% for 4C crosstalk in average and up to 99.99% at most. The percentage of 4C and 4C+3C crosstalk at instruction level is also reduced from the control GCC average value, by 9% and 15% in crosstalk rate difference, respectively. It confirms so that our *NARR* algorithm is effective in reducing the crosstalk especially for the 4C class. Of course, we can combine in the future with the methods proposed in Kuo et al. [Kuo, Chiang and Hwang (2007)] such as instruction scheduling, *NOP* padding to reduce further the crosstalk interference.

Acknowledgement: This work was supported by the General Project of Humanities and Social Sciences Research of the Ministry of Education (16YJA740039); the Foundation Project of Philosophy and Social Science of Hunan (17YBA115).

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

Bamberg, L.; Najafi, A.; Garciaortiz, A. (2019): Edge effect aware low-power crosstalk avoidance technique for 3D integration. *Integration*, vol. 69, no. 1, pp. 98-110.

Chen, W.; Lueh, G.; Ashar, P. J.; Chen, K.; Cheng, B. (2018): Register allocation for Intel processor graphics. *Symposium on Code Generation and Optimization*, pp. 352-364.

Cui, X.; Ni, Y.; Miao, M.; Jin, Y. (2017): An enhancement of crosstalk avoidance code based on fibonacci numeral system for through silicon vias. *IEEE Transactions on Very Large Scale Integration Systems*, vol. 25, no. 5, pp. 1601-1610.

Duan, C.; Calle, V. H. C.; Khatri, S. P. (2009): Efficient on-chip crosstalk avoidance codec design. *IEEE Transactions on Very Large Scale Integration Systems*, vol. 17, no. 4, pp. 551-560.

Florea, A.; Geliert, A. (2016): E-learning approach of the graph coloring problem applied to register allocation in embedded systems. *Sixth International Conference on Innovative Computing Technology*, pp. 173-178.

Gupta, U.; Ranganathan, N. (2011): A utilitarian approach to variation aware delay, power, and crosstalk noise optimization. *IEEE Educational Activities Department*, vol. 19, no. 9, pp. 1723-1726.

Guthaus, M. R.; Ringenberg, J.; Ernst, D. J.; Austin, T.; Mudge, T. et al. (2001): MiBench: a free, commercially representative embedded benchmark suite. *IEEE International Symposium on Workload Characterization*, pp. 3-14.

Halak, B.; Yakovlev, A. (2010): Throughput optimization for area-constrained links with crosstalk avoidance methods. *IEEE Transactions on Very Large Scale Integration Systems*, vol. 18, no. 6, pp. 1016-1019.

Jiao, H.; Wang, R. R.; He, Y. (2018): Crosstalk-noise-aware bus coding with low-power ground-gated repeaters. *International Journal of Circuit Theory and Applications*, vol. 46, no. 2, pp. 280-289.

Kananizadeh, S.; Kononenko, K. (2018): Improving on linear scan register allocation. *International Journal of Automation and Computing*, vol. 15, no. 2, pp. 228-238.

Kuo, W. A.; Chiang, Y. L.; Hwang, T. T.; Wu, A. C. H. (2007): Performance-driven crosstalk elimination at postcompiler level-the case of low-crosstalk op-code assignment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 3, pp. 564-573.

Liu, F.; Yarom, Y.; Ge, Q.; Heiser, G.; Lee, R. B. (2015): Last-level cache side-channel attacks are practical. *IEEE Symposium on Security and Privacy*, pp. 605-622.

Lozano, R. C.; Carlsson, M.; Blindell, G. H.; Schulte, C. (2019): Combinatorial register allocation and instruction scheduling. arXiv: Programming Languages. https://arxiv.org/abs/1804.02452.

Lucas, A. H.; Moraes, F. (2009): crosstalk fault tolerant noc: design and evaluation. *IFIP IEEE International Conference on Very Large Scale Integration*, pp. 81-93.

Mangard, S.; Oswald, E.; Standaert, F. X. (2011): One for all-all for one: unifying standard differential power analysis attacks. *IET Information Security*, vol. 5, no. 2, pp. 100-110.

Moll, F.; Roca, M.; Isern, E. (2003): Analysis of dissipation energy of switching digital CMOS gates with coupled outputs. *Microelectronics Journal*, vol. 34, no. 9, pp. 833-842.

Mutyam, M. (2009): Selective shielding technique to eliminate crosstalk transitions. *ACM Transactions on Design Automation of Electronic Systems*, vol. 14, no. 3, pp. 1-43.

Odaira, R.; Nakaike, T.; Inagaki, T.; Komatsu, H.; Nakatani, T. (2010): Coloring-based coalescing for graph coloring register allocation. *IEEE/ACM International Symposium on Code Generation & Optimization*, pp. 160-169.

Ohama, Y.; Yotsuyanagi, H.; Hashizume, M.; Higami, Y.; Takahashi, H. (2017): On selection of adjacent lines in test pattern generation for delay faults considering crosstalk effects. *International Symposium on Communications and Information Technologies*, pp. 1-5.

Park, J.; Xu, X.; Jin, Y.; Forte, D.; Tehranipoor, M. (2018): Power-based side-channel instruction-level disassembler. *IEEE 55th ACM/ESDA/IEEE Design Automation Conference*, pp. 1-6.

Poletto, M, A.; Sarkar, V. (1999): Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, vol. 21, no. 5, pp. 895-913.

Shirmohammadi, Z.; Mozafari, F.; Miremadi, S. G. (2017): An efficient numerical-based crosstalk avoidance codec design for NoCs. *Microprocessors and Microsystems*, vol. 50, no. 1, pp. 127-137.

Shirmohammadi, Z.; Sabzi, H, Z. (2018): DR: overhead efficient RLC crosstalk avoidance code. *International Conference on Computer and Knowledge Engineering*, pp. 1-6.

Su, X.; Wu, H.; Xue, J. (2017): An efficient WCET-aware instruction scheduling and register allocation approach for clustered VLIW processors. *ACM Transactions in Embedded Computing Systems*, vol. 16, no. 5, pp. 1-21.

Tabani, H.; Arnau, J.; Tubella, J.; Gonzalez, A. (2018): A novel register renaming technique for out-of-order processors. *High-Performance Computer Architecture*, pp. 259-270.

Weng, T.; Lin, C.; Shann, J. J.; Chung, C. (2010): Power reduction by register relabeling for crosstalk-toggling free instruction bus coding. *International Computer Symposium*, pp. 676-681.

Wimmer, C.; Franz, M. (2010): Linear scan register allocation on SSA form. *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pp. 170-179.

Zhang, J.; Fang, L.; Li, L.; Zhang, Z. (2015): A novel approach to detecting hardware trojan horses. *International Symposium on Computational Intelligence and Design*, pp. 43-46.