Computer Systems
Science and Engineering

Tech Science Press

DOI: 10.32604/csse.2023.027081
Article

Read-Write Dependency Aware Register Allocation

Sheng Xiao[1,*], Yong Chen[2], Jing He[3] and Xi Yang[4]

[1]School of Computer Science, Hunan First Normal University, Changsha, 410205, China
[2]School of Information Engineering, Nanjing Audit University, Nanjing, 211815, China
[3]Department of Computer Science, Kennesaw State University, Kennesaw, 30144-5588, USA
[4]Hunan Huayi Experimental Middle School, Changsha, 410205, China
*Corresponding Author: Sheng Xiao. Email: sxiao@hnfnu.edu.cn
Received: 11 January 2022; Accepted: 06 April 2022

Abstract: Read-write dependency is an important factor restricting software efficiency. Timing Speculative (TS) is a processing architecture aiming to improve energy efficiency of microprocessors. Timing error rate, influenced by the read-write dependency, bottlenecks the voltage down-scaling and so the energy efficiency of TS processors. We proposed a method called Read-Write Dependency Aware Register Allocation. It is based on the Read-Write Dependency aware Interference Graph (RWDIG) conception. Registers are reallocated to loosen the read-write dependencies, so resulting in a reduction of timing errors. The traditional no operation (Nop) padding method is also redesigned to increase the distance value to above 2. We analyzed the dependencies of registers and maximized the average distance value of read and write dependencies. Experimental results showed that we can reduce all read-write dependency by Nop padding, as well as the overhead timing errors. An energy saving of approximately 7% was achieved.

Keywords: Read-write dependency; timing speculative; energy efficiency

1 Introduction

In the software industry and in social assessment of "carbon peaking and carbon neutralization" energy consumption is an important indicator. Many researchers have studied the parameter from different aspects [1–4]. Timing speculation (TS) was a concept recently proposed to apply to energy-efficient microprocessors [5]. The TS processing breaks through the restriction of traditional circuits on timing constraints, and makes possible a more effective software design for timing error detection and recovery. The TS uses modules such as enhanced latches, checkers, and restorers to dynamically detect and recover errors [5,6]. At the same time, it allows the occurrence of occasional timing errors and executions under lower voltage of power supply, achieving thus a higher efficiency in energy consumation. However, the error recovery operations consume a high proportion of energy. Reducing error recovery operations is thus critical for the energy efficiency improvement. The read after write dependency (RAW) between data is the main cause of an increased error rate. Therefore, reducing the RAW dependency becomes the key component of efforts to improve the TS processing.

To reduce timing errors, we can use a compiler. The compiler is a piece of system software. Many optimization schemes are used to generate optimized code for different architectures with a compiler for the error reduction purposes. In our previous work [7], we used register reallocation to reduce crosstalk, resultig in energy-saving and code-security improvement. Recently, works on compilation-based TS processing were also published [8,9]. Hoang et al. used compiler transformations to replace long-delay operations with faster ones, thereby reducing the number of timing errors [8]. Sartori et al. [9] evaluated the relationship between current compilation optimizations and TS processing before the optimization. They found that the closer the read-write dependency, the greater the likelihood of timing errors.

Many research efforts have been done for timing error detection and recovery. Tziantzioulis et al. [10] proposed a model for voltage-scaling induced timing errors, called b-HiVE. This model incorporated the timing-error detection and recovery attributes. The corresponding impact of these incorporated parameters on the overall model accuracy was demonstrated. For accurately computing DTS and activity information, Based on a concept of event propagation Zhang et al. [11] proposed EventTimer, a dynamic timing analysis engine. Assare et al. [12] proposed a framework to estimate the number of timing errors experienced by an application running on timing-speculative processor. Temperature- and aging- induced timing errors in the joint accelerator-algorithm interactions were investigated by Paim et al. for illustrating the runtime impacts of these errors [13]. They demonstrated the runtime behavior of three advanced block-matching video encoder algorithms in a joint SAD-accelerating operation, run based on 14 nm-FinFET technology. Tsiokanos et al. [14] provided a novel cross-layer framework. It addressed the lack of a holistic methodology for the understanding of the full system impact of hardware timing errors, such as when these errors propagating from the circuit-level through the microarchitecture, up to the application software. Ainsworth et al. [15] did a transformation of ParaMedic to ParaDox. The transformation resulted in a high performance in both error-intensive and scarce-error scenarios, with correct executions even in either undervolted or overclocked cases. With ExHero, a fully automated framework developped by Tsiokanos et al. [16], a dynamic timing analysis was performed based on the historical execution data of a number of in-flight instructions. Shin et al. [17] experimented a one-cycle error correction, by gating only the main latch in each stage of the pipeline that precedes a failed stage. PreFix, a method developped by Soman et al. [18], could handle hardware errors, keep running a faulty core and execute instructions. However, the above method require the support of additional hardware devices, so increasing the system costs.

On the software side, there were also some studies considering the compiler optimization of TS processors. Hoang et al. [8] found that some code sequences demand more circuit timing deadlines than others. Furthermore, by selectively replacing these codes with instruction sequences which are not only semantically equivalent but also able to reduce activities on timing critical circuit paths, we can trigger fewer timing errors and hence reduce recovery costs. Sartori et al. [9] advocated that binaries for timing speculative processors should be optimized differently from those generated for conventional processors, to maximize energy-saving benefits of timing speculation. There were also some of the methods overusing hardware resources, shown in references [19–21]. Meixer et al. [19] established a code conversion method to avoid using faulty processor components. Reddi et al. [20] applied compiler optimization to reduce the stress on the power delivery system. Hari et al. proposed a system software-guided method for detection of hardware faults happended during the lifetime of a processor [21]. All of these findings indicate that software approaches are valuable to improve the performance of TS processor.

In this paper, we propose a new read-write dependency reduction method to improve the performance of TS architecture. A heuristic register reallocation procedure was set up to reschedule

the registers, based on the dynamically monitored distance values of read-write dependencies in each basic block. Nop fill methods were then used for further distancing the read-write dependencies.

2 Materials and Methods

2.1 Motivation

2.1.1 Timing Speculation Overview

Timing speculation (TS) can be used to improve energy efficiency through voltage reduction. In order to solve the accidental errors, the corresponding error detection and recovery circuit have been redesigned for the TS processors. Fig. 1 shows the relevant logic of a typical TS processor like Razor, or called Razor trigger. In the first clock cycle, the circuit meets the timing requirements, no timing error occurs, and the Error signal remains low.

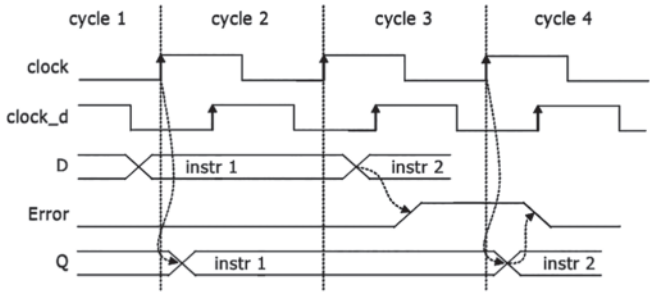

Figure 1: Typical TS processor pipelining [5]

When the combinational logic exceeds the expected delay due to sub-critical voltage scaling, timing errors will be detected. As shown in the third cycle of Fig. 1, due to the excessive scaling of voltage, the effective data of the shadow latch are inconsistent with the data in the main trigger, resulting in a system error and an error signal at high level. In a general pipeline system, once an error occurs in a certain stage, the subsequent pipeline work needs to be re-executed. Only in this way can the whole system work normally. However, in the TS processor, if an error is found, the correct value is reserved in the shadow latch, so the error in a certain stage can be accurately recovered. Subsequently, the content in the shadow latch can be directly used for processing. Therefore, timing errors can be prevented from affecting the entire pipeline. For example, in cycle 4, the recovery logic works. The shadow latch is used to recover the error data, thus ensuring that the error will not remain in the subsequent stages.

The TS processor effectively reduces the impact of occasional timing errors on the system through the error detection and recovery circuit. However, these functional units still requires a large amount of overhead. How to minimize the verhead is an important issue to be solved for the TS processor.

2.1.2 RAW in Pipeline

Reducing power consumption is an important aspect of processor design. TS processors can make the system work in a low power state through error recovery logic. However, its error recovery logic needs to be supported by additional energy consumption. Too many errors will make the TS processor consume more energy. Therefore, reducing TS processor errors is the key to improving TS processor performance. According to the research in [9], the error rate of TS processor is related to the dependence on data read and write. If the read write dependency can be eliminated, most timing inference errors will be eliminated.

The read/write dependency is mainly caused by the segmented execution of the pipeline. In a five level pipeline, as shown in Fig. 2, its second stage is to read data from a register, and its fifth stage is to write data to a register. When the last instruction needs to read the register value written by the previous instruction between two consecutive instructions, a RAW dependency will be generated. More generally, if *Dis (i, j)* is to represent the distance between instruction i and instruction j, and instruction j needs to read the data written to the register by instruction i, then when *Dis (i, j)* meets Eq. (1), there is a dependency between them after reading and writing. In Eq. (1), *DISRAW* is the distance between *RegRead* and *WriteBack* stages.

$$Dis\,(i,\ j)\ < DISRAW \tag{1}$$

Figure 2: Pipeline of StrongArm

2.2 Read-Write Dependency-Aware Register Reallocation (RWDRR)

Aiming to realize the active optimization of the whole program such as the system library, we make the optimization process get the disassembly code and the analyzed results as inputs. The Read-Write Dependency-Aware Interferogram (RWDIG) constructor is then used to build the RWDIG from the disassembly code and to set its weights. We make finally the RWDRR processor analyze the RWDIG to relocate the registers and to generate optimized code. The core of the optimization is to build the RWDIG and RWDRR programs. The details are described in the following subsections.

2.2.1 RWDIG Construction

In order to conduct a more comprehensive analysis, the code of the static library is also included in the analysis. We take the binary code as the entry point, and obtain the register level information through decompilation. Then, with the help of the method described in [22], a read-write correlation sensing interferogram (RWDIG) is constructed. RWDIG is a directly weighted graph. It can be represented by quadruple $RWDIG = (V, E_I, E_N, W_E)$, 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 can't share the same register, $e'(u', v') \in E_N$ expresses that the node u and node v may be the same register and the weight represents the minimal number of instructions of such read-write dependency if the node u and v are assigned to the same register.

To improve the pertinence of the program, we have modified the calculation of weights. In [22], the weight value of this graph only represents the number of times that read/write dependencies exist, but in fact, TS processors are only sensitive to read/write dependencies that generate forwarding. If read after write dependencies do not generate forwarding, it will not adversely affect timing errors of TS processors. Therefore, we correct the calculation of the weight to: if two nodes are allocated to the same register, the number of times that they generate forwarding between them.

For the RWDIG setup, we take the disassembly code as input and assume that there are infinite registers, called virtual registers in many compilers. First, we change the disassembly code to a static single assignment form for each basic block, ensuring that the registers are defined only once (lines 1–4). Then, we construct the data stream for each basic block using the method described in reference [23] and obtain the lifetime of each register in each instruction (lines 5–6). The interference map can be constructed by analyzing the active registers in each instruction (lines 7–15). Once the interferogram is obtained, we can use the contour analysis results to add weights to the edges (lines 16–19). Then we

finally return the constructed RWDIG (line 22). The detailed construction is presented in Algorithm 1. In this program, the CFG is the control flow graph, and each node represents a basic block containing the number of instructions executed in order. The $Lfreg_i$ represents a register that is defined before the instruction and will be used after the instruction that called the active register.

Algorithm 1: *RWDIG* Construction.

Input:
S: the disassemble of source codes
M: the profile inform
Output:
$RWDIG(V,E_I, E_N, W_E)$
1:Consturct the CFG by S
2: **for** each node v of CFG **do**
3: Tranform *v* to *SSA*
4: **end for**
5: *do the dataflow analysis for the transformed CFG*
6: $Lfreg_i$ = get the life info by step 5 for each node in CFG
7: **for** each node v in CFG **do**
8: **for** each i in v **do**
9: **for** each m, k in $Lfreg_i$ and m $\neq$ k **do**
10: *V.add(m)*
11: *V.add(k)*
12: *EI.add(m, k)*
13: **end for**
14: **end for**
15:**end for**
16: **for** each weight info <m, k, $w_{m,k}$> in M **do**
17: E_N *.add(m, k)*
18: W_E*.add(*$w_{m,k}$*)*
19: **end for**
20: **return** $RWDIG(V,E_I, E_N, W_E)$;

2.2.2 RWDIG Based RWDRR

Based on the above modified RWDIG, we have implemented a new RWDIG based read/write correlation aware register relocation (RWDRR) algorithm. We take the RWDIG of each process as the input, and sort according to the weight of the edge in E_N. (line 1). Because the larger the weight value of the E_N side is, the higher the frequency of the two nodes connected to the side is, the easier it is to generate more read-write dependencies. Therefore, we try to allocate the two points connected to the E_N side to different registers to reduce read-write dependencies (lines 2–36). But different from the reference [22], when there are multiple registers to choose from, we try to reuse the used registers to leave more available registers for conflict resolution. The detailed program is shown in Algorithm 2:

Algorithm 2: *RWDIG* Construction.

Input:

The $RWDIG(V, E_I, E_N, W_E)$ for every processor;

The registers that can be used $R = \{r_0, r_1, \ldots, r_n\}$

Output:

The results of registers allocation for each node V in *RWDIG*;

1:$E' =$ *sort the edges* in E_N by W_E in decrease
2: **for each** $e \in E'$ **in ordered do**
3: **if** $e \in E_I$ **then**
4: **if** neither node is assigned for any registers **then**
5: Get the most used two registers to assigned for the two nodes
6: **else if** both nodes are allocated the same register **then**
7: Change any one of the node with the minimal used register
8: **else if** only one node is allocated to the register r_i **then**
9: search the register $r_j \neq r_i$ to allocate for node v.
10: **end if**
11: **else**
12: **if** neither node is allocated to any register **then**
13: do it as the above line 5
14: **else if** only one node is allocated to register r_i **then**
15: do it as the above line 9
16: **end if**
17: **end if**
18:**end while**

2.3 Nop Padding Method

To loosen effectively the read-write dependency, we use the Nop approach to pad the instructions, with a minimal distance value preset at 2. The details are shown in Algorithm 3. First, the register in the original assembly instruction is replaced according to the result obtained by RWDRR (line 1–5). Next, the CFG is constructed (line 6). And we traverse each instruction in each node n of the CFG in turn (line 7–21). If the instruction i is the first instruction of the CFG node, we traverse each precursor node p of the node n in turn. If there is a read-write dependency between the last instruction of p, which expressed by p_{last}, and the current instruction i, we add a Nop instruction at the end of the corresponding precursor node p. If the instruction i is not the first of the CFG node, we judge whether it has read-write dependency with its previous instruction. If so, add a Nop instruction before the current instruction i.

Algorithm 3: *Nop* padding Construction.

Input:

M: the register allocation map for each node V in RWDIG

S: the disassemble of source codes

Output:

CFG': the CFG after Nop padding

1: **for** each instruction i in S

(Continued)

Algorithm 3: Continued

```
2:  for each register r in i:
3:    R = M(V(r))
4:  end for
5: end for
6: CFG(V',E') = Construct the CFG for S
7: for each node v in V':
8:  for each instruction i in v:
9:   if i is the first instruction:
10:   for each p in Parent (v):
11:    if p_last and i have Read-Write dependency
12:      add Nop instruction after p_last in p
13:    end if
14:   end for
15:  else:
16:   if i and i −1 have Read-Write dependency:
17:    add Nop instruction before i
18:   end if
19:  end if
20:  end for
21: end for
22: return CFG'
```

3 Results

3.1 Experimental Methodology

To evaluate the effectiveness of the method in this paper, we built an experimental process as shown in Fig. 3.

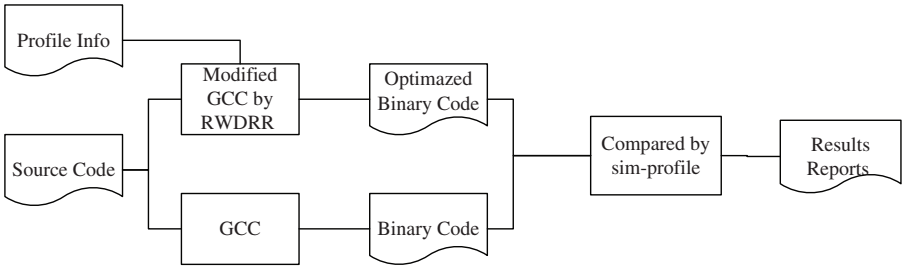

Figure 3: Experimental framework

In the experiment, we first use the GCC [23,24] compiler to obtain the binary under the O2 optimization option. Then, we modified the register allocation method of GCC by using our RWDRR and obtained the optimazed binary code. The source and the optimized codes are then compared and the performance of RWDRR are evaluated with the following parameters: average distance of read-write dependency, minimum distance of read-write dependency, impact of Nop padding and energy saving value. We select the MiBench [25] and Mediabench [26] as the benchmark, and the target

architecture is StrongARM. The results of read-write dependency rumbers obtained in the benchmark testing are shown in Table 1.

Table1: The number of Read-Write dependencies by GCC

Benchmarkes	Number	Benchmarkes	Number
basicmatch_large	345580	dijkstra_large	254954
basicmnatch_smnall	345497	dijkstra_small	122938
bitcnts_large	238603	patricia_large	299280
bitcnts_small	238599	patricia_small	127280
qsort_large	269550	ispell	399010
qsort_small	290855	search_large	276056
susan_s_large	404997	blowfish_en	298706
susan_e_large	213386	blowfish_de	81994
susan_c_large	347166	rijndael_en	242539
susan_s_small	387228	rijndael_de	223996
susan_e_small	185618	sha	400153
susan_c_small	335301	bin/rawcaudio(adpcm_c)	294799
jpeg-6a/cjpeg_large	557450	bin/rawcaudio(adpcm_d)	358832
jpeg-6a/djpeg_large	483977	crc	168817
jpeg-6a/cjpeg_small	537632	fft_i	316053
jpeg-6a/djpeg_small	500575	fft	316049
lame3.70/lamne_large	330314	bin/toast	6463
lame3.70/lame_small	347278	bin/untoast	6556

3.2 Experiment Results

3.2.1 Occurrence Times with One Distance of Read-Write Dependency

In some architectures, read-write dependencies can be eliminated by pass-through techniques. This requires that the distance value of the read-write dependency must be set over 1. The worst-case execution results of our proposed algorithm are evaluated by comparing the number of occurrences of read-write dependency distance obtained in GCC mode with that obtained in RWDRR condition. Table 2 shows the optimization results of our RWDRR. The obtained value of occurrences of read-write dependencies for GCC next distance, and that of occurrences of read-write dependencies for RWDRR next length (without Nop padding) for the same testing case set are listed side by side.

It is clear that there is a significant decrease in the number of occurrences of dependency (Table 2, column 3 *vs.* column 2), and as can be seen from the last row of Table 2, the average number of occurrences decreases from 536 to 184, without Nop Padding. If Nop padding is added, the number of occurrences is reduced to zero (data shown below), so RWDRR shows a positive effect on the benchmark results for the worst-case processing.

Table2: The comparsion result of worst cases

Benchmarks	GCC	RWDRR without Nop padding
basicmatch_large	421	241
basicmatch_small	334	220
bitcnts_large	507	295
bitcnts_small	777	59
qsort_large	307	143
qsort_small	648	287
susan_s_ large	588	211
susan_e_large	427	129
susan_c_large	143	87
susan_s_small	861	265
susan_e_small	980	108
susan_c_small	371	138
jpeg-6a/cjpeg_large	786	270
jpeg-6a/djpeg_large	783	219
jpeg-6a/cjpeg_small	740	76
jpeg-6a/djpeg_small	349	121
lame3.70/lame_large	649	174
lame3.70/lame_small	99	96
dijkstra_large	636	224
dijkstra_small	437	299
patricia_large	829	124
patricia_small	439	77
ispell	919	258
search_large	208	255
blowfish_en	879	248
blowfish_de	901	202
rijndael_en	196	166
rijndael_de	314	98
sha	942	251
bin/rawcaudio(adpcm_c)	245	257
bin/rawcaudio(adpcm_d)	433	117
crc	977	269
fft_i	294	71
fft	146	296
bin/toast	488	132
bin/untoast	245	167
average	536	184

3.2.2 Influence of Nop Padding

Nop padding increases the distance between read and write dependencies, but also adds extra idle time and space. Therefore, we analyze the impact of Nop padding on the whole system from both positive and negative aspects.

Positive Influence

In order to distance further the read-write dependency, Nop padding instructions are used. To evaluate its positive influence, we compared its performance by Eq. (2). Where Dis_{avg} is the averaged distance of read-write dependency without Nop padding, and Dis_{avgnop} is the averaged distance of Read-write dependency with Nop padding. The results are shown in Fig. 4.

$$imp = \frac{Dis_{avg} - Dis_{avgnop}}{Dis_{avg}} \tag{2}$$

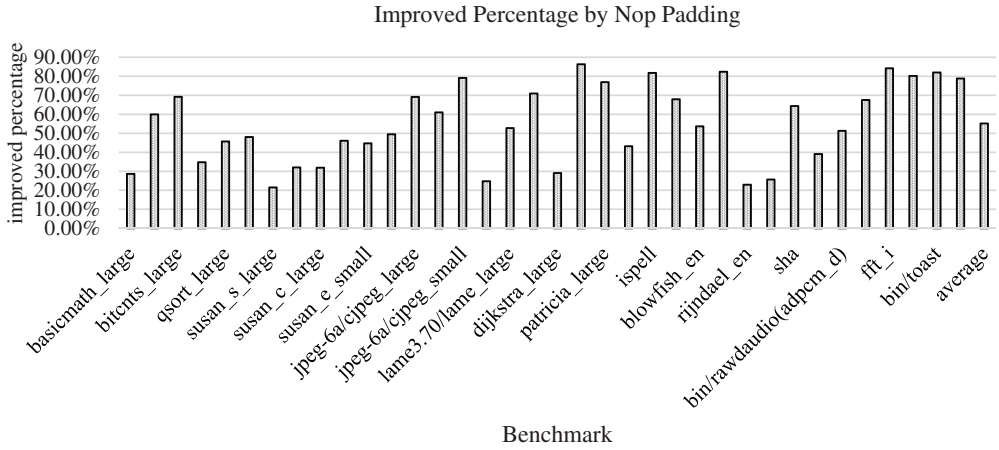

Figure 4: The improvement of Nop padding on occurrence numbers

It can be seen from Fig. 4 that the improvement percentage by Nop padding ranges from 20% to over 85% , such as for benchmark Dijkstra_large, for its averaged distance value of read-write dependency, with an averaged improvement value of about 55%, as shown in the last bar in the above figure. Therefore, the effect of Nop padding on the dependency distance is obvious and can be effectively used to eliminate the read-write errors.

Negative Influence

Although the Nop instruction itself is a null operation and consumes very low energy, it still wastes the corresponding clock cycles and increases the size of the code itself. In order to evaluate its impact, we have carried out analytical experiments. The results are shown in Figs. 5 and 6, respectively.

Fig. 5 shows the ratio of the time increased by the Nop instruction to the total program time. It can be seen from the figure that the maximum time cost is 2.26%, the minimum is only 0.3%, and the average increased cost is 1.29%.

Fig. 6 shows the ratio of the space added by the Nop instruction to the original size of the program. It can be seen from the figure that the space cost is slightly higher than the time cost, up to about 5%, and the average cost is 2.75%. But in general, the cost is acceptable.

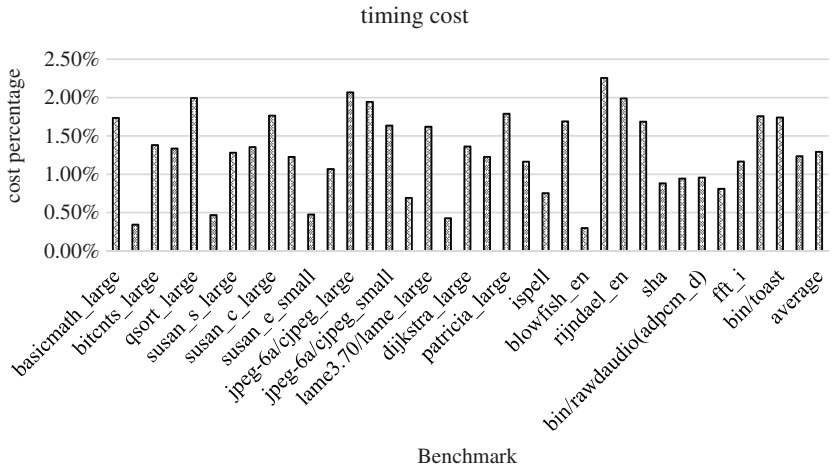

Figure 5: The time cost of the Nop padding

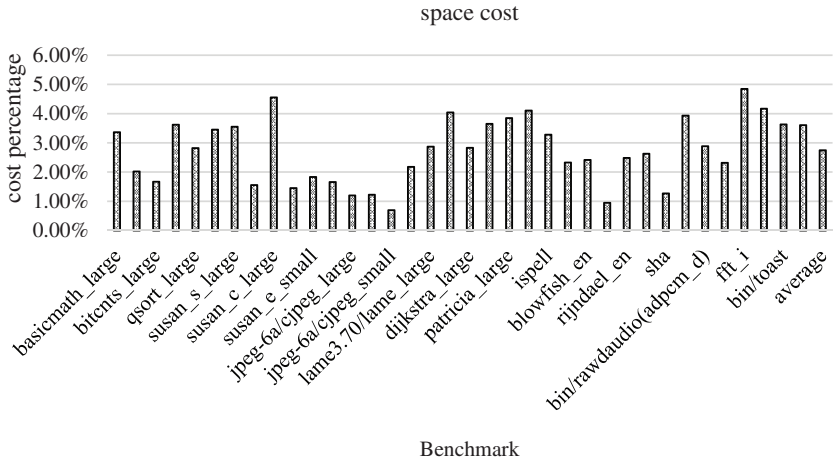

Figure 6: Space cost of the Nop padding

3.2.4 The Energy Saving Values

Impact of our RWDRR method on the system energy consumption were evaluated, by normalizing and calculating the energy consumption through Eq. (3): Where, E1 represents the system energy consumption before optimization, and E2 represents the energy consumption after optimization.

$$P = \frac{E1 - E2}{E1} \tag{3}$$

The obtained results (Fig. 7) show that the system energy consumption can be improved, with the maximum increase of nearly 14%, and the averaged improvement percentage is about 7%.

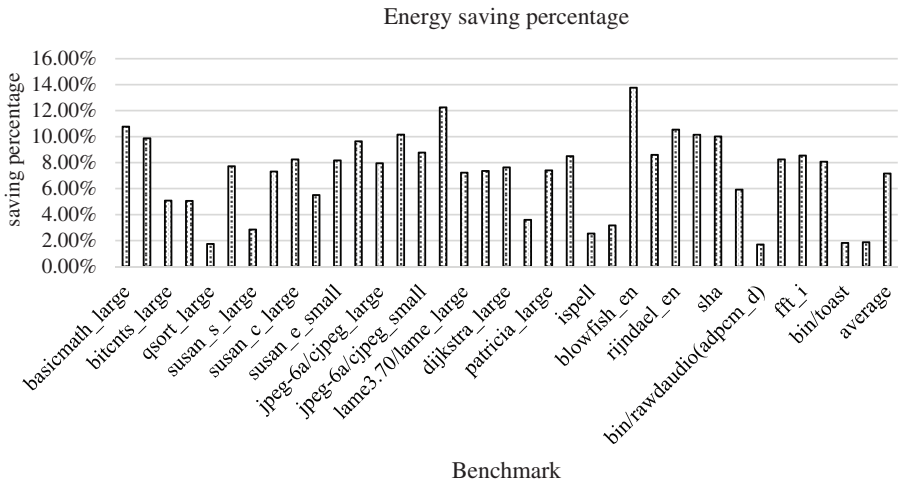

Figure 7: Energy saving of the RWDRR

4 Discussion

Experimental results showed that our new method can effectively increase the read-write correlation distance through an optimized speculative execution. This optimization method reduces the energy consumption caused by speculative error detection and recovery and is especially promising for applications running under low power supply voltage. As a software optimization approach, it requires no additional hardware, and no dependency on a specific architecture. The experimental results showed that our algorithms are capable of increasing the dependency distance value by 55%, and that the reduced timing error rate allows to achieve an average power savings of about 7%.

5 Conclusions

Software optimization on the TS platform showed promising features for improving energy efficiency in the microprocessor development. Reducing the read and write dependencies rsulted in a significant reduction of the timing error rate and thus the voltage of the TS processor. Optimized voltage down-scaling benefits microprocessor's power consumption. Our new method and experimental results demonstrated that the goal of reducing timing error rate and read/write dependancy of registers is achievable via software optimizations without additional hardware support.

Funding Statement: This work was supported by the Project of Hunan Social Science Achievement Evaluation Committee (XSP20YBZ090, Sheng Xiao, 2020).

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

References

[1] P. Hemalatha and K. Dhanalakshmi, "Cellular automata based energy efficient approach for improving security in iot," *Intelligent Automation & Soft Computing*, vol. 32, no. 2, pp. 811–825, 2022.

[2] S. Mahmoud and A. Salman, "Cost estimate and input energy of floor systems in low seismic regions," *Computers, Materials & Continua*, vol. 71, no. 2, pp. 2159–2173, 2022.

[3] M. Maharajan and T. Abirami, "Energy efficient qos aware cluster based multi-hop routing protocol for wsn," *Computer Systems Science and Engineering*, vol. 41, no. 3, pp. 1173–1189, 2022.

[4] H. G. Zaini, "Forecasting of appliances house in a low-energy depend on grey wolf optimizer," *Computers, Materials & Continua*, vol. 71, no. 2, pp. 2303–2314, 2022.

[5] D. J. Ernst, N. S. Kim, S. Das, S. Pant, R. R. Rao *et al.,* "A Low-power pipeline based on circuit-level timing speculation," in *Proc. IEEE/ACM*, Washington, DC, USA, pp. 7–18, 2003.

[6] B. Greskamp and J. Torrellas, "Paceline: Improving single-thread performance in nanoscalecmps through core overclocking," in *Proc. PACT*, Brasov, Romainia, pp. 213–224, 2007.

[7] S. Xiao, J. He, X. Yang, Y. Wang and J. Lu, "Crosstalk aware register reallocation method for green compilation," *Computers, Materials & Continua*, vol. 63, no. 3, pp. 1357–1371, 2020.

[8] G. Hoang, R. Findler and R. Joseph, "Exploring circuit timing-aware language and compilation," in *Proc. ASPLOS*, Newport Beach, CA, USA, pp. 345–356, 2011.

[9] J. Sartori and R. Kumar, "Compiling for energy efficiency on timing speculative processors," in *Proc. DAC*, SanFrancisco, CA, USA, pp. 1297–1304, 2012.

[10] G. Tziantzioulis, A. M. Gok, S. M. Faisal *et al.,* "B-HiVE: A bit-level history-based error model with value correlation for voltage-scaled integer and floating point units," in *2015 52nd ACM/EDAC/IEEE Design Automation Conf. (DAC)*, Moscone Center San Francisco, CA, IEEE, 2015.

[11] Z. Zhang, Z. Guo, Y. Lin, R. Wang and R. Huang, "EventTimer: Fast and accurate event-based dynamic timing analysis," in *2022 Design, Automation & Test in Europe Conf. & Exhibition (DATE)*, Antwerp, Belgium, pp. 945–950, 2022.

[12] O. Assare, R. K. Gupta, "Performance analysis of timing-speculative processors," *IEEE Transactions on Computers*, vol. 71, no. 2, pp. 407–420, 2022.

[13] G. Paim, H. Amrouch, L. M. G. Rocha, B. Abreu, E. A. C. da Costa *et al.,* "A framework for crossing temperature-induced timing errors underlying hardware accelerators to the algorithm and application layers," *IEEE Transactions on Computers*, vol. 71, no. 2, pp. 349–363, 2022.

[14] I. Tsiokanos, G. Papadimitriou, D. Gizopoulos and G. Karakonstantis, "Boosting microprocessor efficiency: Circuit- and workload-aware assessment of timing errors," in *2021 IEEE Int. Symp. on Workload Characterization (IISWC)*, Virtual Online, pp. 125–137, 2021.

[15] S. Ainsworth, L. Zoubritzky, A. Mycroft and T. M. Jones, "ParaDox: Eliminating voltage margins via heterogeneous fault tolerance," in *2021 IEEE Int. Symp. on High-Performance Computer Architecture (HPCA)*, Virtual Online, pp. 520–532, 2021.

[16] I. Tsiokanos and G. Karakonstantis, "ExHero: Execution history-aware error-rate estimation in pipelined designs," *IEEE Micro*, vol. 41, no. 1, pp. 61–68, 2021.

[17] I. Shin, J. J. Kim, Y. S. Lin and Y. Shin, "One-cycle correction of timing errors in pipelines with standard clocked elements. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, no. 2, pp. 600–612, 2016.

[18] J. Soman and T. M. Jones, "High performance fault tolerance through predictive instruction re-execution," in *2017 IEEE Int. Symp. on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, Cambridge, UK, pp. 1–4, 2017.

[19] A. Meixner and D. J. Sorin, "Detouring: Translating software to circumvent hard faults in simple cores," in *Proc. ICSTW*, Anchorage, AK, USA, pp. 80–89, 2008.

[20] V. Reddi, S. Campanoni, M. S. Gupta, M. D. Smith, G. Wei *et al.,* "Eliminating voltage emergencies via software-guided code transformations," *ACM Transactions on Architecture and Code Optimization*, vol. 7, no. 12, pp. 1–28, 2010.

[21] S. K. S. Hari, M. L. Li, P. Ramach, B. Choi and S. V. Adve, "Mswat: Low-cost hardware fault detection and diagnosis for multi-core systems," in *Proc. IEEE/ACM*, New York, NY, USA, pp. 122–132, 2009.

[22] S. Xiao, J. He, X. Yang, H. Zhou and Y. Yuan, "Timing error aware register allocation in ts," *Computer Systems Science and Engineering*, vol. 40, no. 1, pp. 273–286, 2022.

[23] Z. Zhou, Z. Ren, G. Gao and J. He. "An empirical study of optimization bugs in GCC and LLVM," *Journal of Systems and Software*, vol. 174, no. 3, pp. 110884, 2020.

[24] M. Shalabi, "Programs optimization in GCC compiler," *Scientific Annals of Computer Science*, vol. 9, no. 2, pp. 91–108, 2018.

[25] M. R. Guthaus, J. Ringenberg, D. J. Ernst, T. Austin, T. Mudge *et al.,* "Mibench: A free, commercially representative embedded benchmark suite," in *Proc. IEEE/WWC*, Washington, DC, USA, pp. 3–14, 2001.

[26] C. Lee, M. Potkonjak and W. M. Smith, "MediaBench: A tool for evaluating and synthesizing multimedia and communications systems," in *Proc. IEEE/ACM*, Saint Louis, MO, USA, pp. 330–335, 1997.