Soft Error Mask Analysis on Program Level

Lei Xiong, Qingping Tan, Jianjun Xu
School of Computer
National University of Defense Technology
Changsha 410073, China
leixiong@nudt.edu.cn, eric.tan.6508@gmail.com, jjun.xu@gmail.com

***Abstract*—Computer hardware which consist billions of transistors could fail because of soft errors with the improving of semiconductor technology, and these failure could result in incorrect program execution. However, soft errors could be masked. Most methods of SIFT (Software-Implemented Fault Tolerance) do not take mask into account. In fact, these parts of program which could mask soft errors don't need to be added redundancy instructions with SIFT methods. These parts of program are analyzed in this paper. We equal logical errors of soft errors to errors in program. In our analysis, we focus on two parts of program. One is idle program which is related to control flow. The probability for them to be executed is low. The other is dynamically dead codes. From static view, dynamically dead codes include statically dead codes and statically partial dead codes. By control flow graph, we analyze the condition which could mask soft errors of these parts of program. Finally, we design an experimental framework to demonstrate our analysis. Those codes which we analyze show their ability to mask soft errors.**

Keywords-soft errors; error mask; fault tolerance; control flow; dynamically dead codes

I. INTRODUCTION

Due to improvement of semiconductor technology, transistors are getting smaller and faster. Those transistors yield performance enhancements, but their lower threshold voltages and tighter noise margins make them less reliable. When facing energetic particles striking the chip, microprocessors which consist billions of transistors are susceptible to transient faults. Transient faults which are also called soft errors are intermittent faults. These faults do not cause permanent damage, but may result in incorrect program execution by altering transferring signals or stored values [1][2][3].

When soft errors appear, system could not be failure because of soft error mask. For an instance, when the hardware component which is not used encounters soft error, it can't affect system state. Soft error mask can be classified several classes based on its effects on different level, such as mask on architecture level and mask on program level. The mechanism of soft error mask on program level is that program errors which are caused by soft errors of hardware are masked on the level of program with control flow and data flow.

SIFT methods protect program to tolerate soft errors. Most of SIFT methods are implemented by compiler. The compiler which are implemented tolerance algorithm compile common program to redundancy program. These methods copy data which are used in program, and compute every part of program twice to make sure the right result. Most methods of SIFT do not take soft error mask into account. However, it is obviously that there is no need to copy those data which are related to those parts of program which can mask soft errors. If we can distinguish these parts of program which could mask soft errors from the whole program, we can ignore these parts of program with a fault tolerance method. Then, the method can lead to higher performance.

This paper gives an analysis of soft error mask on the level of program. Based on our analysis, we show these parts of program which could mask soft error. In accordance with those methods that protect program statically, these parts of program are analyzed from a static approach. Those parts of program which are idle program and dynamically dead codes can mask program errors which are caused by soft errors. Idle program which have low probability to be executed can't activate their program errors. Dynamically dead codes on the contrary are executed, but their results don't come to be used. To meet the trade-off between reliability and performance, we give a measurement to these parts of program to decide if these parts of program need to be protected with our method. To demonstrate our analysis, we give some experiments by the method of simulation. Our rest structures are as follows. In Section 2, we show two parts of program which could mask error. Section 3 analyzes conditional branches which is one of mask parts in program. Section 4 shows dynamically dead codes which is another mask part of program. We give an experiment frame in Section 5. We show results of experiments and discuss these results in Section 6. Section 7 is related work. In last Section, we make a conclusion.

II. SOFT ERRORS AND ITS MASK ON PROGRAM LEVEL

Radiation of cosmic rays has an effect on semiconductor chip. This event can cause a single event upset (SEU), and SEU cause soft errors of hardware. These hardware include storage, bus, cache and functional unit related to instruction pipeline. Soft errors of hardware could lead to errors of program during execution. These program errors could

 ISBN:978-1-61208-113-7

propagate in program with control flow and data flow. We consider program errors which are caused by soft errors as logical errors of soft error. Based on the mechanism that soft errors on hardware affect program execution, we choose to show program behaviors on the low level. It is convenient to show the effect of soft errors to program on low level, such as assembly language or machine language.

Based on our ground, when facing on energetic particle striking, inter-arrival times for raw faults in hardware components are independent and exponentially distributed with density function $\lambda e^{-\lambda t}$ [4]. When time t, in program, we assume that the instruction whose execution is related to time t is in the location of l. The number of functional units which are related to execution of the instruction is denoted as n. Density functions of these hardware components are denoted as $\lambda_1 e^{-\lambda_1 t}$, $\lambda_2 e^{-\lambda_2 t}$, $\cdots$, $\lambda_n e^{-\lambda_n t}$. The event that result of the instruction execution is right equals to the event that these n functional units are reliable. We denote the error probability of the instruction in the location of l as $P(l)$. Then

$$P(l) = 1-(1-P(u_1))(1-P(u_2))\cdots(1-P(u_n)) \quad (1)$$

and

$$P(l) = \int_0^t (\lambda_1 + \lambda_2 + \cdots + \lambda_n) e^{-(\lambda_1+\lambda_2+\cdots+\lambda_n)t} dt \quad (2)$$

We denote $\lambda_S = \lambda_1 + \lambda_2 + \cdots + \lambda_n$, so

$$P(l) = \int_0^t \lambda_S e^{-\lambda_S t} dt \quad (3)$$

We transform this formula:

$$P(l) = \int_0^t \lambda_S e^{-\lambda_S t} \frac{dt}{dl} dl \quad (4)$$

Since execution time of most instructions is single cycle, except some special instructions which need additional instruction cycles. To simplify our model, we assume that the average execution time of every instruction is the same. As a result, time and number of executed instructions keep linear relationship. So

$$P(l) = \int_0^l \lambda_S e^{-\lambda_S l} dl \quad (5)$$

We can see from this formula that error locations in program are almost exponentially distributed.

If soft errors of hardware lead to wrong results of instructions and wrong results don't affect program outcome, we take the situation as soft error mask. Soft error mask is a dynamic conception which is related to execution of program. However, dynamic behaviors of program are based on their static program. In this paper, program is modeled statically. We discuss program from static approach. There are several situations can mask soft errors which are as follows.

1) Idle program can mask errors. We define idle program as the part of program which are not executed during one execution. These program parts which can't be executed, they are pure idle program. The quantity of this program is small. Additionally, those parts of program which have probability to be executed can be idle program. These parts of program are not executed during one execution. Soft errors on hardware which are related to these parts of program are masked.
2) Dynamically dead instructions can mask errors in program. The results of dynamically dead instructions may not be used. They include two parts of program in static program. One is statically dead codes, their results are not used after these code. So their errors can be masked. The other one is partially dead codes. Partially dead codes are related to control flow. On some paths results of these codes are used, but on some other paths their results are dead. If results of these codes are dead definitely, their errors are masked.

III. ERROR MASK OF IDLE PROGRAM

Idle program are parts of program which can't be executed in one execution. We use control flow graph to describe the control flow of program. Its node represents basic block which contains sequential instructions, its edge represents a possibility of control flow path transition. Edge decides the block which follows the last block.

A. Idle Program

We assume the process that one node chooses the next node is a Markov process. Markov process is a type of stochastic process in which the outcome of a given trial depends only on the current state of the process. A system consisting of a series of Markov events is called a Markov chain [8]. As Figure 1 shows, we assign every basic block a label, such as A_1, A_2, $A_3, \cdots, A_n$, and we divide them into several groups based on their position. We denote them as G_i. For example, A_1 is the first node, $G_1 = \{A_1\}$, A_2, A_3, A_4 can be the next node of A_1, so we group them together, $G_2 = \{A_2, A_3, A_4\}$. Like this, we group A_5, A_6 together, A_7, A_8, A_9 are single group, $G_3 = \{A_5, A_6\}$, $G_4 = \{A_7\}$, $G_5 = \{A_8\}$, $G_6 = \{A_9\}$. We denote the basic block which is chosen to be executed as A_{ci} according to G_i, and $A_{ci} \in G_i$. Moreover, we denote the set of chosen basic block as $S_{ci} = \{A_{c1}, A_{c2}, A_{c3}, \cdots, A_{c(i-1)}\}$. According to the definition of Markov chain above, for a program, $S_{ci} (i = 1,2,3,\cdots,n)$ is a Markov chain. Every S_{ci} is only decided by the state of $S_{c(i-1)}$. It is independent of other states. We denote the probability of every branch as P. If the branch between basic block A_i and basic block A_j, we denote P_{ij} as probability of the branch. We define the probability of a branch as the ratio of the branch executing times to all executing times of source basic block. If the branch is definitely executed, the probability is 1, but we also describe it as P_{ij}.

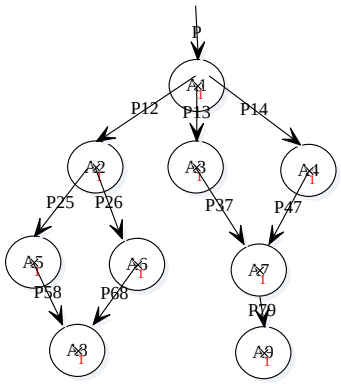

Figure 1. Control flow graph

B. *Probability Tree and Its Effects to Soft Error Mask*

As control flow graph shows, there is a probability for a node to next node. And next node also has probability to choose its next node. With control flow, every branch has its probability to be executed. We call it as probability tree. The probability of every branch is basic unit of this tree.

We analyze the probability tree from top to down. We assume that the real probability of the first node is P. We assume that the branch is from basic block i to basic block j. The real probability of this branch can compute like this: $P_{ij(real)} = P_{(basicblock)i} * P_{ij}$. The real probability of this branch from basic block i to basic block j is the multiplicative result of probability of basic block i and probability of this branch. The execution probability of basic block i is the sum of branch probability which points to the basic block i. This is: $P_{(basicblock)i} = \sum P_{*i(real)}$. We know the real probability of the first basic block, so we can compute all the real probability of probability tree from top to down.

To evaluate the trade-off between performance and reliability of a tolerance computer system, we must not over care the reliability of the computer system. So we have reasons to ignore protection to some parts of program. We give a threshold probability t. The threshold probability is suitable for system requirement that meets the trade-off between performance and reliability. We propose a new software-implemented fault tolerance method which pursues the trade-off between performance and reliability of system. In our method, branch probability which is less than t is ignored to be protected. In control flow graph, if the node which is pointed by a branch is an end node, then there is no other basic block next this basic block, we ignore the destination node to be protected. If the node is not an end node, there is a sub tree began with the destination node, we ignore this sub tree to protect. For a static program, after the application of our method to tolerate fault, the program which we will protect is transformed. To the transformed program, we do not need to adjust the probability of branches. For example, in Figure 1, if $P_{12} < t$, we ignore A_2 with its next branches and nodes to be protected. As a result, the program slice that we will protect is transformed. The program slice is described as Figure 2.

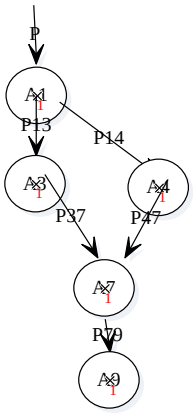

Figure 2. Control flow graph of program slice

IV. ERROR MASK OF DYNAMICALLY DEAD CODES

Dynamically dead instructions are those instructions whose results are dead when program runs. They include two classes of static codes. One is statically dead codes. The saved results of this kind of codes are not used before the saving place was written again. Shown as Figure 3, I2 is statically dead instruction. The other one is partially dead instructions. There are more than one path after this instruction. On some execution paths the saved results of this kind of instruction were not used before the saving place was written again. The instruction is dead on those execution paths. While, on other execution paths, the saved results of this kind of instruction are used. On these execution paths the instruction is alive. Partially dead instruction is also related to conditional branch. Figure 4 shows an instance of partially dead instruction. I2 assigns a value to variable X. After I2, there are two branches. Variable X is used on the left branch, but it is not used before assigning a new value to variable X of I5 on the right branch. So instruction I2 is alive if left path is chosen, and instruction I2 is dead if right path is chosen.

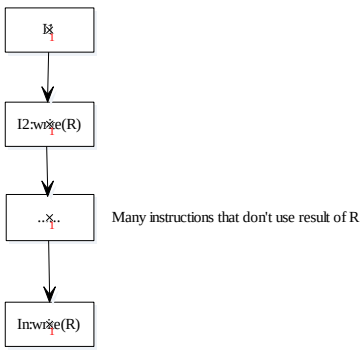

Figure 3. Statically dead instructions

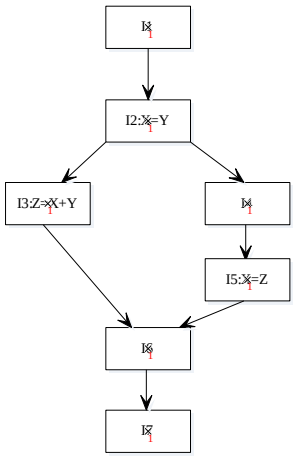

Figure 4. Partially dead instructions

If an instruction is partially dead instruction, it must be in a basic block which contains this instruction. Then the execution probability of this instruction equals to the execution probability of the basic block, and the latter has been computed in Section 3. Then $P_{PDI} = P_{basicblock}$. After this partially dead instruction, there are many paths, some make this instruction live, and some make this instruction dead. We can compute the sum probabilities of which make the partially dead instruction live. It equals to the sum of branch probability which can make the instruction live, and the sum probability of partially dead instruction to live is $\sum P_{live}$. So the final probability of this partially dead instruction to live is P_{1-live} , $P_{1-live} = P_{PDI} * (\sum P_{live})$. We assume a threshold probability as p . The variable p also meets the extent of trade-off between performance and reliability. To every partially dead instruction, if $P_{1-live} < p$, we ignore protection to this partially dead instruction. However, if $P_{1-live} > p$, we still have to protect it with redundancy instructions. As Figure 4 described, if $P_{1-live} < p$, the program slice, which we will protect, is transformed. The program slice is as Figure 5.

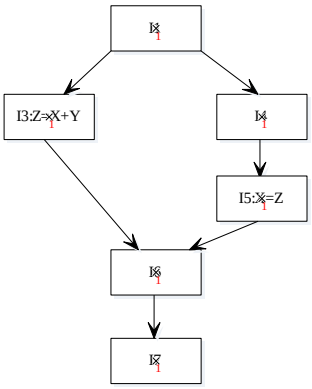

Figure 5. Control flow graph of program slice

.

V. EXPERIMENTAL FRAMEWORK

To demonstrate our analysis, we design an experimental framework. Our experimental framework includes SPEC2000 benchmarks, simplescalar tool set, and static fault injection tool. We choose some integer benchmarks from the SPEC2000 benchmark suite: gzip vpr, gcc, mcf, crafty, parser, perlbmk, gap, vortex, bzip2, twolf. Their sources are all written by C language. These benchmarks were compiled using a modified version of GNU GCC2.7.2.3 at the –O2 optimization level. To evaluate the mask ability of benchmark, we run benchmark on simplescalar tool set. Simplescalar tool set is a simulation tool for simulating computer architecture of a processor. We use sim-safe functional simulator to run our benchmark. Sim-safe function simulator is safer compared with other functional simulator such as sim-fast. We implement a static fault injection tool to inject fault into benchmark program. This tool can inject fault to determined location in program, and it can also inject fault to any random location in program.

The experimental framework is described as Figure 6. Firstly, we compile benchmarks to assembly codes. Then, we inject faults into assembly codes with fault injection tool. Thirdly, we run this program on simplescalar tool set with sim-safe simulator. Finally, we get simulation results.

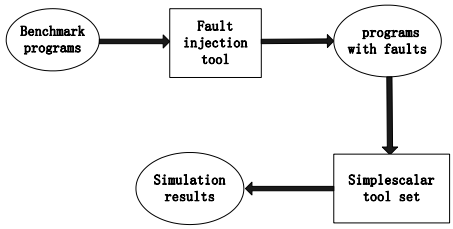

Figure 6. Experimental framework

VI. RESULT AND DISCUSSION

We give three error injection experiments. First, we inject errors into determined locations which contain idle program and partial dead codes. Second, we randomly inject error into locations which contain idle program and partially dead codes. Third, we randomly inject error into locations which are all over the whole program. The first experiment is to show that if the result of program can be still right when these code encounter faults. The second experiment is also to show this aim when faults are random. The third experiment show results under the real situation that faults turn on random location of program. In Section 2.1, we have known error locations in program are exponential distributed. In our experiment, we generate data by exponential distribution with Monte Carlo simulation.

 ISBN:978-1-61208-113-7

The statistics of conditional branch in SPEC2000 benchmark is as table 1. We can see from table 1 that the quantity of conditional branch instructions is much enough to be paid attention. Take benchmark bzip2 for example, the whole number lines of its code is 4650 lines, the number of its "if" statement is 245, and the number of its "switch" statement is 10. One "if" statement is at least one line, and one "switch" statement is at least two lines. These conditional branch instructions are at least 265 lines of code. Moreover, if we only care 80% of them, the rest codes which we need not to protect is at least 53 lines. However, these codes could be double or even more than 53 lines. If the scale of program is larger than bzip2, these codes could be more.

TABLE I. STATISTICS OF CONDITIONAL BRANCH IN SPEC2000

Spec name	Counts of "if" statement	Counts of "switch" statement
gzip	400	2
vpr	870	38
gcc	12805	524
mcf	80	1
crafty	1098	29
parser	624	0
perlbmk	4174	183
gap	2921	23
vortex	3018	49
Bzip2	245	10
twolf	1378	5

The percentage of statically partial dead code with SPEC2000 is as Figure 7. We can see from the figure that partial dead code is an important part of program. Among 11 SPEC2000 benchmarks, the percentage of statically partial dead code varies from 0.1 to nearly 7. In additional to idle program, these parts of program take a lot of account of the program. With our software-implemented fault tolerance method to tolerate soft errors, there could be a dramatic enhancement to system performance.

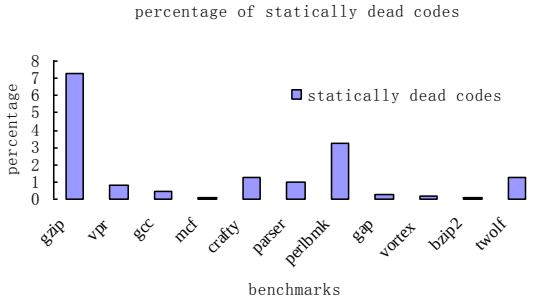

Figure 7. Percentage of statically partial dead codes in SPEC2000

We choose one of the SPEC2000 benchmark gzip to show our experimental results. For the ability of soft errors mask, gzip can present the behavior of all programs. Gzip is a data compression program. It uses Lempel-Ziv coding (LZ77) as its compression algorithm. We choose 1000 files to be compressed with gzip, and then we calculate the probabilities of conditional branches. We choose 20% as a threshold probability. It means we only care about those conditional branches whose probability is above 20%. Based on the frequency of soft errors and our program length, we statically inject 5 faults into the program in every experiment. For each experiment, we test 10 times. Then we calculate the average results. Results of Experiment are as Figure 8.

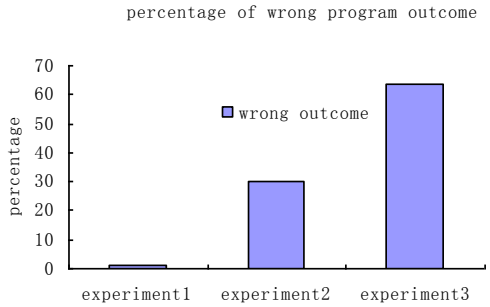

Figure 8. Simulation results of gzip with fault injection

Experiment 1 shows fault tolerance ability of program part which has low probability to be executed. Experiment 2 shows fault tolerance ability of program part which has probability to be executed. Experiment 3 shows fault tolerance ability of the whole program. From results of experiment 1, when there are errors in conditional branches or statically partial dead code whose probability to be executed is low, execution of the program can not be affected. This result matches our analysis. From Figure 8 of experiment 2, the effects of conditional branches and statically partial dead code to the program outcome is limited, these effects are weakened by control flow. From experiment 3, we can see that the ability of program to tolerate errors is considerable during its execution. Except these codes which are related to control flow can tolerate errors in program, there are still other codes can tolerate errors, such as errors are masked from logical operation.

VII. RELATED WORK

Soft error mask has been studied on the level of architecture. Mukherjee distinguishes these bits which do not affect program outcome as unACE bits [1]. If there is a fault on unACE bits, there is no effect to the system because unACE bits can't affect committed architectural state. Mukherjee also proposes AVF (Architectural Vulnerability Factor) to measure reliability of hardware component. AVF is the probability that a user-visible error will occur given a

 ISBN:978-1-61208-113-7

bit flips in a storage cell. LI et al. have shown AVF of hardware component is a variable because of soft error mask [4]. Mukherjee et al. show several situations of unACE bits on micro-architecture and architecture level [1].

There are some algorithms of soft error tolerance on software level, such as EDDI, SWIFT, and so on [2][3][5][6]. They are different from the method implemented by hardware duplication. Hardware methods protect hardware structure and duplicate hardware structure which can be corrupted. However, software methods protect programs by duplicating instructions and adding checking instructions into program [5] [6]. We take EDDI for example. It is implemented by compiler. The compiler compile source program to executable program containing redundancy code. EDDI divides program into many basic blocks. Based on basic block, it partition basic block by store instruction, this is called storeless basic block. Inside a storeless basic every instruction is duplicated, and before store instruction every result is checked [5]. However, EDDI, SWIFT protect program on the uniform way. No matter whether these sections of program can mask error or not, they protect them on the same way. Although these methods can get a good reliability of system, they sacrifice the performance of system.

Except these soft errors mask on architecture level, there are some situations for soft error to be masked on the level of program. Based on software protection, we give an analysis of soft errors mask on program level. Our results of analysis are helpful to optimize these tolerance systems which are implemented by the method of software-implemented hardware fault tolerance. The purpose of our analysis is to meet the trade-off between performance and reliability. If these parts of program which can mask soft error are clear for us, there is no need to protect these parts of program. Therefore, redundancy instructions are reduced. Performance of system can be improved.

VIII. CONCLUSION

Model transistors of processor get smaller and faster, but their lower threshold voltages and tighter noise margins make them less reliable. When computer system expose in the space, its components may encounter soft errors because of high energy particle striking. Once soft error of computer component happened, it may affect the execution of program. However, soft errors of computer components may not lead to system failure, because these soft errors may be masked. This paper analyze soft error mask on program level. Based on the mechanism that soft errors affect system reliability, we compute error distribution caused by soft error in program. In our analysis, there are two kinds parts of program related to control flow can mask soft error. They are idle program which is related to conditional branches and dynamically dead codes. Based on control flow graph, the probability of branches and basic block to be executed are computed by our method. In our method, if the probability of basic block to be executed is less than threshold probability, we considered this basic block as idle program. These parts of program need not to be protected. Dynamically dead code includes statically dead code and partially dead code from static approach. In our method, partially dead code whose probability to live is less than threshold are ignored to be protected. We designed an experiment frame to demonstrate our analysis. In our experiment, we statically inject faults to program, and run program on a simulator. Experiment Results match our analysis.

REFERENCES

[1] S. Mukherjee, Architecture Design for Soft Errors, USA, Morgan Kaufmann Publishers, 2008.

[2] G.A. Reis, J. Chang, and N. Vachharajani, "Software-Controlled Fault Tolerance. ACM Transactions on Architecture and Code Optimization", Vol.V, No. N, 2005, pp. 1-28.

[3] G.A. Reis, "Software Modulated Fault Tolerance", A dissertation presented to the faculty of Princeton University, 2008.

[4] X. Li, "Soft Error Modeling and Analysis for Microprocessors", A dissertation presented to computer science in the graduate college of the University of Illinois, 2008.

[5] N. Oh et al., "Error Detection by Duplicated Instructions in Super-Scalar Processors", IEEE Transactions on Reliability, Vol. 51, No. 1, March 2002, pp. 63–75.

[6] G.A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "SWIFT: Software-implemented Fault Tolerance", In Proceedings of the 3rd International Symposium on Code Generation and Optimization. March 2005, pp. 243–254.

[7] A. Benso, S.D. Carlo, and G.D. Natale, "Static Analysis of SEU Effects on Software Applications", International test conference. IEEE, 2002, pp. 500-508.

[8] S. Sparks, S. Embleton, and R. Cunningham, "Automated Vulnerability Analysis Leveraging Control Flow for Evolutionary Input Crafting", Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007), 2007, pp.477-486.

[9] J. L. Hennessy and D. A. Patterson, Computer Architecture: A Quantitative Apprach, Third Edition, BeiJing, China Machine Press, 2002.

[10] L. David and I. Pusut, "Static Determination of Probabilistic Execution Times", Proceedings of the 12th 16th Euromicro Conference on Real-Time System, ECRTS, 2004.

[11] J. Singer, "Towards Probabilistic Program Slicing", Dagstuhl Seminar Proceedings 05451 Beyond Program Slicing, 2006.

[12] M. Weiser, "Program slicing", IEEE Transactions on software Engineering. August 1984, pp. 352-357.

[13] J.A .Butts and G. Sohi, "Dynamic Dead-Instruction Detection and Elimination", In 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). October 2002, pp. 199-210.

[14] B. Fahs, S. Bose and M. Crum, "Performance Characterization of a Hardware Mechanism for Dynamic Optimization", In 34th Annual International Symposium on Microarchitecture (MICRO). December 2001, pp. 16-27.

[15] A. V.Aho, R. Sethi, and J.D. Ullman, Compilers: Principles, Techniques and Tools, Addison-Wwsley, 1985.

[16] S.S. Muchnick, Advanced Compoiler Design Implementation, Elsevier, 1997.

[17] J. Xue, Q. Cai, and L. Gao, "Partial Dead Code Elimination on Predicated Region", software-practice and experience. 36, 2004, pp. 1655-1685.

 ISBN:978-1-61208-113-7