Performance-Reliability Tradeoff Analysis for Multithreaded Applications

Isil Oz*, Haluk Rahmi Topcuoglu†, Mahmut Kandemir‡, Oguz Tosun*

*Computer Engineering Department, Bogazici University, 34342, Istanbul, Turkey
{isil.oz, tosuno}@boun.edu.tr
†Computer Engineering Department, Marmara University, 34722, Istanbul, Turkey
haluk@marmara.edu.tr
‡Dept. of Computer Science and Engineering, Pennsylvania State University, University Park, PA 16802, USA
kandemir@cse.psu.edu

***Abstract*—Modern architectures become more susceptible to transient errors with the scale down of circuits. This makes reliability an increasingly critical concern in computer systems. In general, there is a tradeoff between system reliability and performance of multithreaded applications running on multicore architectures. In this paper, we conduct a performance-reliability analysis for different parallel versions of three data-intensive applications including FFT, Jacobi Kernel, and Water Simulation. We measure the performance of these programs by counting execution clock cycles, while the system reliability is measured by Thread Vulnerability Factor (TVF) which is a recently-proposed metric. TVF measures the vulnerability of a thread to hardware faults at a high level. We carry out experiments by executing parallel implementations on multicore architectures and collect data about the performance and vulnerability. Our experimental evaluation indicates that the choice is clear for FFT application and Jacobi Kernel. Transpose algorithm for FFT application results in less than 5% performance loss while the vulnerability increases by 20% compared to binary-exchange algorithm. Unrolled Jacobi code reduces execution time up to 50% with no significant change on vulnerability values. However, the tradeoff is more interesting for Water Simulation where nsquared version reduces the vulnerability values significantly by worsening the performance with similar rates compared to faster but more vulnerable spatial version.**

***Index Terms*—Multi-Core Architectures and Support, Reliable Parallel and Distributed Algorithms**

I. Introduction

As technology scales and transistor sizes in modern architectures continuously decrease, transient error rates increase. Soft errors are a kind of transient errors that result from a fault in a single-bit and their rates keep increasing with current design trends [1]. Recent research addressed the soft error problem from both hardware and software perspectives [2], [3], [4], which indicate that the reliability is an important concern in computer architecture.

Chip multiprocessors (CMPs), which have been viewed as the most promising option for the next generation computing platforms, are more error prone due to their smaller transistor sizes and more aggressive power modes [5], [6]. While prior research on CMPs explored performance and power related issues, reliability problem has relatively taken less attention.

Although many performance metrics for multicore architectures have been suggested in the literature [7], [8], there is a lack of metric that quantifies the reliability of parallel architectures. Program Vulnerability Factor (PVF) measures the vulnerability of a program to the hardware faults [9], but it is a metric only for single-threaded applications. A recent study [10] has proposed *Thread Vulnerability Factor* (TVF) which defines the vulnerability for multithreaded applications running on multicore architectures. TVF measures the vulnerability of a thread (which is one of the threads of a multithreaded application) to transient hardware faults by considering the codes of the threads that communicate with that thread as well as the code of the thread itself.

Our main goal in this work is to present a performance-reliability analysis to provide the characteristics of different multithreaded applications with alternative design choices. We consider three data-intensive programs including *Fast Fourier Transform (FFT)*, *Jacobian Calculation*, and *Water Simulation* in our evaluation. While we evaluate two different parallel implementations of FFT algorithm, two different algorithms of the water simulation benchmark are selected. We also apply loop transformations to the standard parallel Jacobi implementation. While we use execution cycles as performance metric, TVF metric is employed to quantify vulnerability. Our analysis compares different implementations of the same program by considering performance and reliability aspects.

Our major contributions can be summarized as follows:

- We perform a performance-reliability analysis of different multithreaded applications running on multicore architectures. We evaluate TVF values and gather execution clock cycles of a set of parallel programs with different versions. Specifically, we demonstrate the performance and vulnerability behavior of two different parallel implementations of FFT algorithm, two different algorithms of water simulation and four versions of Jacobian calculation each of which has an application of a loop transformation.

- We present the performance and vulnerability values of parallel programs and discuss the effect of design choices on the system performance and reliability by comparing different implementations. We observe that the choice is clear for FFT algorithm which has different vulnerability values with almost equal performance, and for Jacobian calculation which, in contrast, has no significant difference in vulnerability values but distinct performance characteristics. However, the choice is not clear for water simulation which demonstrates a performance-reliability tradeoff with higher performance but higher vulnerability as well.

II. Background and Related Work

Since the reliability becomes an important concern in modern architectures, it is crucial to define suitable metrics to evaluate the reliability of the system.

Architectural Vulnerability Factor (AVF) has been defined as the probability that a fault in a processor will induce an error in the program output [11]. In the AVF analysis, the bits in the hardware structure (called as ACE bits) which have visible effect in the program output have been distinguished from the bits (called as un-ACE bits) which may affect the program flow or performance, but have no effect in the final output. ACE bits have been used in calculating the AVF of the hardware structure.

Although AVF is well-defined and useful metric, it is difficult to evaluate the vulnerability of a program to a hardware fault since it is a hardware-based metric. To define the vulnerability of a software to transient errors, Program Vulnerability Factor (PVF) has been proposed [9]. This metric measures the vulnerability of a program to the hardware faults in a microarchitecture independent way and provides the relative vulnerability of the programs to be able to make decisions about the reliability of these programs. While calculating the PVF, microarchitectural resources that are hardware dependent are not considered only architectural resources are taken account by the software's view.

While PVF is a software-based metric, it is only useful for single-thread applications. In a recent work, Thread Vulnerability Factor (TVF) has been proposed as a reliability metric for multithreaded applications on CMP architectures [10]. TVF measures the vulnerability of a thread to transient hardware faults by considering the thread itself and the threads that communicate with that thread via memory operations. TVF has two components including *LVF* and *RVF*, the former holds the vulnerability caused by the code of the target thread while the latter represents the vulnerability induced by the communication between the target thread and the other threads in the application.

In our performance-reliability analysis, we use the TVF metric to evaluate the relative vulnerability of multithreaded applications. TVF can be calculated as follows:

$$TVF(T_i) = [\, w_L \times LVF(T_i)\,] + [\, w_R \times RVF(T_i)\,],$$

where $LVF(T_i)$ is the local vulnerability factor for thread i and $RVF(T_i)$ is the remote vulnerability factor for thread i which iterates over all threads that sends data to thread i. w_L and w_R are weight values for local and remote vulnerability factors respectively.

TVF is evaluated for three hardware components including register file, ALU unit, and memory (cache). The local vulnerability factor (LVF) term is calculated in a similar fashion to PVF [9], where the vulnerability values of the resources in the thread are combined and normalized by considering their vulnerable intervals. The remote vulnerability factor (RVF) term represents the vulnerability impact of the threads that interact with the target thread in a multithreaded application. RVF of thread i can be calculated as follows:

$$RVF(T_i) = \frac{\sum rVF(T_i, T_j)}{RV},$$

where $rVF(T_i, T_j)$ represents the remote vulnerability factor for thread i induced by thread j which sends data to thread i. $RVF(T_i)$ iterates over all threads that sends data to thread i. This is because these threads can pass a corrupted value to thread i. The RV parameter represents the total number of remote memory accesses on thread i.

Figure 1 represents a data sharing between three threads in a multithreaded application, where TVF value of each thread can be calculated as follows:

$$\begin{aligned} TVF(T_1) &= LVF(T_1) \\ &= LVF(A+B). \end{aligned}$$

$$\begin{aligned} TVF(T_2) &= [\, w_L \times LVF(T_2)\,] + [\, w_R \times RVF(T_2)\,] \\ &= [\, w_L \times LVF(T_2)\,] + \\ &\quad [\, w_R \times (\frac{rVF(T_2,T_1) + rVF(T_2,T_3)}{2})\,] \\ &= [\, w_L \times LVF(C+D+E)\,] + \\ &\quad [\, w_R \times (\frac{LVF(A) + LVF(F)}{2})\,]. \end{aligned}$$

$$\begin{aligned} TVF(T_3) &= LVF(T_3) \\ &= LVF(F+G). \end{aligned}$$

To calculate TVF of a thread, the vulnerability of each thread on which that thread is dependent should be considered to capture the effect of these threads on the vulnerability of the thread investigated. Since T_1 is not dependent on other threads, its TVF is calculated using only its instructions. However, the portion of T_1 should also be considered to obtain TVF of T_2. Since T_2 reads data written by T_1 at the end of the code fragment shown as A, the vulnerability of A affects the vulnerability of T_2. Similarly, the code portion F of T_3 affects the vulnerability of T_1 as well. To calculate TVF of T_3, only the code segment of the thread itself is considered as in T_1 since there is no remote thread.

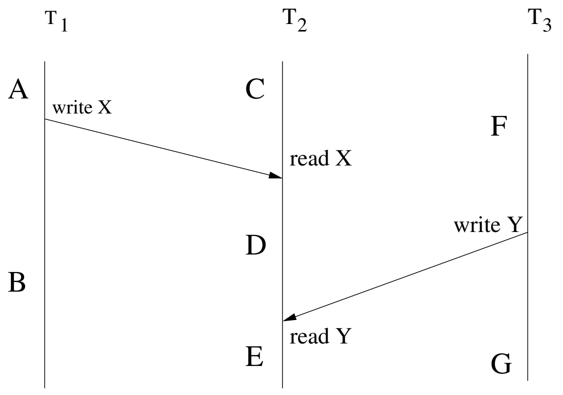

Fig. 1. An example for representing data sharing between threads

III. Experimental Setup

In this section, we present the details of our benchmark applications considered in our experimental evaluation, which is followed by a brief information about our simulation platform.

A. Benchmark Applications

To evaluate performance and reliability behaviors of multithreaded applications, we select three data-intensive applications: Fast Fourier Transform, Jacobi, and Water Simulation. Note that FFT and Jacobi are used in both high end and low end computing.

1) Fast Fourier Transform: Fast Fourier Transform (FFT) is an algorithm to compute Discrete Fourier Transform (DFT), which is a mathematical transform requiring complex number calculations and used in various applications including time series, partial differential equations, and digital signal processing [12]. There are several forms of FFT algorithm; in this work, our focus is on one-dimensional, unordered, radix-2 FFT. The pattern of combination of input array elements used in this calculation is represented by a butterfly network. In a parallel algorithm, it is important to assign input elements into threads by considering the communication cost which affects both performance and reliability. We consider two parallel algorithms defined in [13]: *binary-exchange* and *transpose* algorithms. These two algorithms have different thread communication pattern.

In the binary-exchange algorithm, the array elements which are denoted as their binary representations are mapped to cores such that elements with indices having the same $d = logp$ most significant bits are mapped into the same core where p is the number of cores in the system. The elements that belong to different cores are combined during first d iterations, while the elements that belong to the same cores are combined in the remaining iterations. The communication between the cores only exists along these first d iterations.

The transpose algorithm which involves a matrix transposition operation requires smaller amount of communication among cores. The input array with size n is arranged in an $\sqrt{n}$ x $\sqrt{n}$ two dimensional array in row major order where FFT calculation can be performed by applying FFT over the rows and then applying FFT over the columns. These array elements are mapped to p cores such that each core stores $\sqrt{n}/p$ rows. FFT over the rows requires no communication among the cores. After transposition, FFT over the rows (old columns) is computed to complete FFT operation. The communication between the cores is needed for only transposition operation. The input array in our calculations has 2^{18} double numbers.

2) Jacobi Kernel: Stencil computation represents a common kernel for engineering applications including multimedia processing, quantum dynamics, and electromagnetics [14]. The performance optimization of computations has been extensively studied and several code transformations have been developed to improve data locality in the calculations [15], [16], [17].

To illustrate the effect of the loop transformations, we consider 2-dimensional Jacobi code which updates the contents of grid elements by using neighbor elements in two consecutive loop iterations. Our parallel implementation is simply parallelization of two loops in the calculation.

The loop transformations used in our performance-reliability analysis are *loop unrolling* which reduces the loop overhead with smaller number of iterations, *loop fusion* which replaces multiple loops with a single one, and *loop interchange* which exchanges the order of two iteration variables. We consider 512x512 grid size for Jacobi computations in our experiments.

3) Water Simulation: The Water Simulation is N-body molecular dynamics application which simulates forces and potential energy of water molecules. The method of molecular dynamics is widely used to analyze the atomic structures in materials science, biochemistry, and biophysics [18].

The SPLASH-2 benchmark suite [19] has two parallel *pthread* versions of water simulation application: *Water-nsquared* and *Water-spatial*. Water-nsquared is an improved version of the original Water program in SPLASH [20] and computes force and potentials in $O(n^2)$. The computation is performed over a number of time steps by using a predictor-corrector method. It improves the original version by using a locking strategy in the updates which stores a local copy of the particle accelerations as it calculates and sends to the shared copy at the end. Water-spatial is a more efficient method and uses an $O(n)$ algorithm. It divides the molecules into a grid of cells (processors) and employs spatial locality to calculate inter-molecular forces by considering only molecules in nearby cells. Our experiments for water simulation are conducted for Water-nsquared and Water-spatial programs by considering 512 water molecules and default parameters provided by the SPLASH-2 benchmark.

B. Simulation Platform

To evaluate performance and reliability of the multithreaded applications by measuring their execution times and thread vulnerability factors respectively, we use the Simics toolset [21]. Main characteristics of the simulated multicore are given in Table I.

IV. Results and Discussion

We measure execution clock cycles and calculate TVF of each thread in target applications with respect to L1 caches,

L1 data cache	16K/core 2-way cache
L1 cache latency	1 cycle
L2 shared cache size	4MB 4-way cache
L2 cache latency	10 cycles
Memory latency	200 cycles

TABLE I
PARAMETERS OF THE SIMULATED MULTICORE ARCHITECTURE

register file, and ALU units. Unless otherwise stated, we use equal local and remote vulnerability weights ($w_L = w_R = 0.5$). Our experiments are conducted by using the same number of threads as the core counts (assign one thread per core) in the target architecture.

Table II presents the vulnerability and execution time values of our benchmark applications for two, four, eight, and sixteen-thread (one thread per core) execution scenarios.

One can see from the *FFT* results that two parallel algorithms have different TVF values. As mentioned in Section III, the *binary-exchange* algorithm has more communication whereas the *transpose* algorithm does not have much thread communication after transpose operation. However, the *transpose* algorithm spends its execution cycles to transposition which requires more local resource usage. The effect of this distinct characteristics of the algorithms can be observed on the local vulnerability values as well as on the remote vulnerability values. LVF values for *transpose* algorithm which requires more computation are larger than LVF values for *binary-exchange*, especially for register resources (e.g., binary-exchange 1.27, transpose 1.56 for the 4-core case, binary-exchange 2.53, transpose 3.13 for the 8-core case). Since these parallel algorithms have different thread communication patterns, RVF values which result from communication of threads in a multithreaded application differ for these algorithms as well. The *binary-exchange* algorithm which requires thread communication for more iterations has larger RVF (especially for memory resource) values than the *transpose* algorithm which has thread communication only for one phase (before transposition) of the algorithm (e.g., binary-exchange 0.87, transpose 0.75 for the 8-core case and binary-exchange 1.64, transpose 1.43 for the 16-core case). On the other hand, the execution time values are not so distinct while *binary-exchange* performs well for all core cases.

The results of the *Jacobi* kernel reveal that loop fusion and loop unrolling improves the performance of the code by eliminating loop overhead. On the other hand, loop interchange has no significant effect on the execution time. While the *fusion* increases the vulnerability values (for both local and remote) which results in a tradeoff between performance and reliability, the *loop unrolling*, which has the largest performance gain, does not affect the vulnerability if compared with the original version. The unrolled version has similar vulnerability results, the values are even the same for some cases (1.00, 2.00, 4.03 for memory resources in the 2-core, 4-core and 8-core executions respectively).

The execution time values of the *water* simulation application are significantly different for two parallel versions. The difference becomes more clear as the core count rises due to

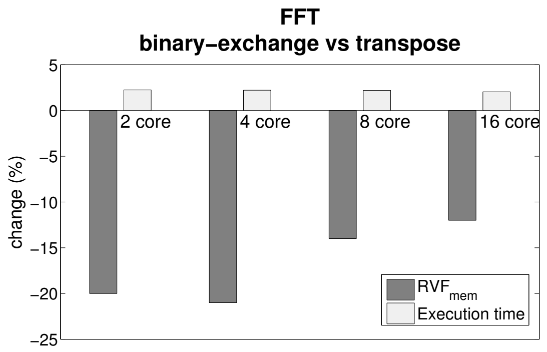

Fig. 2. The change in percentage of RVF values and execution time for versions of FFT

the communication overhead of the *nsquared* algorithm. Although there is more communication overhead in the *nsquared* algorithm, RVF values that result from remote read operations are larger in the *spatial* algorithm. Since cache miss rates are smaller for *nsquared* version due to the lack of spatial locality, LVF values with respect to memory (cache) resources are smaller as well. Therefore, RVF values calculated by using local vulnerability become larger for the *spatial* version which demonstrates that the faster *spatial* algorithm is more vulnerable to soft errors with larger vulnerability values (e.g., nsquared 0.58, spatial 0.64 for the 2-core case and nsquared 1.01, spatial 1.20 for the 4-core case).

Since LVF values have been counted for RVF calculation (RVF value of one thread is calculated by adding TVF values of threads communicating with the target thread), only RVF values are considered as the reliability metric by ignoring very small LVF values that are not counted for RVF calculation. We also examine the vulnerability values for the resource which has the largest difference between different cases. For instance, RVF values with respect to ALU and register resources are not significantly different for two FFT algorithms (e.g., 2.21% higher in transpose algorithm in the 2-core execution for the ALU resource, 3.18% higher in binary-exchange algorithm in the 4-core execution for the register resource). On the other hand, the change in RVF values with respect to memory (cache) resources is large enough to evaluate reliability measure in tradeoff analysis. To evaluate performance-reliability tradeoff between different versions of our applications, we demonstrate the change (in percentage) for both vulnerability values and execution time of the applications (see Figure 2, Figure 3, and Figure 4). The plots provide the behavior of the applications in terms of both reliability and performance if we choose a specific version.

Figure 2 reveals that if we use the *transpose* algorithm instead of the *binary-exchange* algorithm, we should sacrifice approximately 3% performance but gain 20% reliability with lower RVF values in the 2-core case. This observation is valid for any number of cores. We can say that one may prefer the *transpose* algorithm rather than the *binary-exchange* algorithm to work with much higher reliability by accepting little amount of performance loss.

There is similar observation for the versions of the *Jacobi*

			FFT		WATER		JACOBIAN			
			binary-exchange	transpose	nsquared	spatial	original	fused	interchanged	unrolled
2 core	LVF	ALU	0.61	0.61	0.46	0.48	0.92	0.95	0.92	0.96
		Register	0.62	0.76	0.84	0.91	0.51	0.51	0.51	0.51
		Memory	0.55	0.53	1.26	1.32	1.00	1.14	0.99	1.00
	RVF	ALU	0.44	0.45	0.43	0.42	0.90	0.91	0.90	0.94
		Register	0.37	0.38	0.51	0.52	0.40	0.41	0.40	0.39
		Memory	0.29	0.23	0.58	0.64	0.90	0.92	0.86	0.90
	Execution time		32.61	33.35	9.78	8.27	13.52	10.87	13.85	9.19
4 core	LVF	ALU	1.21	1.22	0.93	0.96	1.83	1.90	1.83	1.92
		Register	1.27	1.56	1.42	1.46	0.99	1.01	1.03	0.90
		Memory	1.00	1.06	2.36	2.69	2.00	2.28	1.98	2.00
	RVF	ALU	0.82	0.84	0.85	0.83	1.81	1.82	1.81	1.89
		Register	0.68	0.66	1.05	1.03	0.91	0.86	0.91	0.90
		Memory	0.52	0.41	1.01	1.20	1.82	1.84	1.73	1.82
	Execution time		16.32	16.68	5.46	4.49	6.80	5.42	6.93	4.63
8 core	LVF	ALU	2.43	2.45	1.86	1.92	3.67	3.80	3.67	3.84
		Register	2.53	3.13	2.63	2.77	2.02	1.99	1.89	2.06
		Memory	2.00	2.12	4.40	5.05	4.03	4.55	3.99	4.03
	RVF	ALU	1.66	1.63	1.69	1.64	3.66	3.64	3.63	3.80
		Register	1.16	1.19	2.08	2.05	1.78	1.87	1.87	1.85
		Memory	0.87	0.75	1.73	2.19	3.68	3.67	3.49	3.68
	Execution time		8.17	8.35	3.29	2.59	3.49	2.78	3.56	2.42
16 core	LVF	ALU	4.87	4.89	3.73	3.83	7.33	7.60	7.33	7.67
		Register	5.14	6.35	4.97	5.29	3.96	3.94	4.00	3.99
		Memory	2.86	4.23	8.55	9.73	8.13	9.08	8.06	8.13
	RVF	ALU	3.26	3.25	3.36	3.28	7.28	7.29	7.28	7.61
		Register	2.22	2.18	4.19	4.13	3.78	3.78	3.78	3.73
		Memory	1.64	1.43	3.15	4.33	7.47	7.34	7.10	7.46
	Execution time		4.09	4.18	2.35	1.69	1.99	1.51	2.02	1.45

TABLE II
TVF VALUES AND EXECUTION TIME OF OUR BENCHMARK APPLICATIONS

codes, in that case the choice is clear for the performance. Since the *loop unrolling* gives the best performance among the other loop transformations, we evaluate the vulnerability and performance change if one prefers loop unrolling instead of other versions. Figure 3 presents the change in both RVF and execution time values for loop unrolling and other code versions (original, fused, and interchanged respectively). The vulnerability values do not differ much for each case. Specifically, the values are smaller than 5% even there is almost no change if we compare the original version with the unrolled one. On the other hand, the performance gain is obvious if the unrolled version is used. For instance, if one employs the unrolled version instead of the interchanged one, about 50% speedup is possible by sacrificing only 5% reliability.

Although the choice is clear for two applications in terms of performance and reliability, *water* simulation application reveals more interesting results which yield a tradeoff between performance and reliability concerns. Figure 4 demonstrates this tradeoff between two different water simulation algorithms. While *spatial* has larger performance advantage, *nsquared* is more reliable with similar rates. For the 8-core execution, one should trade 27% performance gain with 21% reliability loss if he selects *spatial* algorithm. We cannot easily conclude that one version satisfies both performance and reliability constraints. While the safety-critical systems with higher reliability needs may prefer the *nsquared* version by sacrificing some performance, the systems for which the performance is the most crucial factor may opt for the *spatial* version, which has higher performance but is much more vulnerable to hardware errors. However, it is difficult to trade the vulnerability with performance for the systems which do not have evident performance and reliability needs.

V. Conclusion

In this paper, we perform a performance-reliability analysis of different multithreaded applications running on multicore architectures. While the performance of the applications is measured by execution clock cycles, we use *Thread Vulnerability Factor* metric to evaluate the relative reliability of multithreaded applications. Using these values gathered from experimental evaluation, we conduct a performance-reliability tradeoff analysis which compares the different parallel versions of three applications in terms of reliability as well as performance. Our results indicate that the choice is clear for *FFT* and *Jacobi Kernel*. The *transpose* algorithm for *FFT* calculation results in less than 5% performance loss while the vulnerability increases by 20% instead of *binary-exchange* algorithm. One can prefer transpose algorithm for better reliability by sacrificing little performance. The *unrolled Jacobi* code reduces execution time up to 50% by not effecting vulnerability values. However, the tradeoff is more interesting for *Water Simulation* application. While *nsquared* version reduces the vulnerability values significantly, it increases execution time with similar rates compared to *spatial* version. One should trade performance with reliability to choose one version of the application.

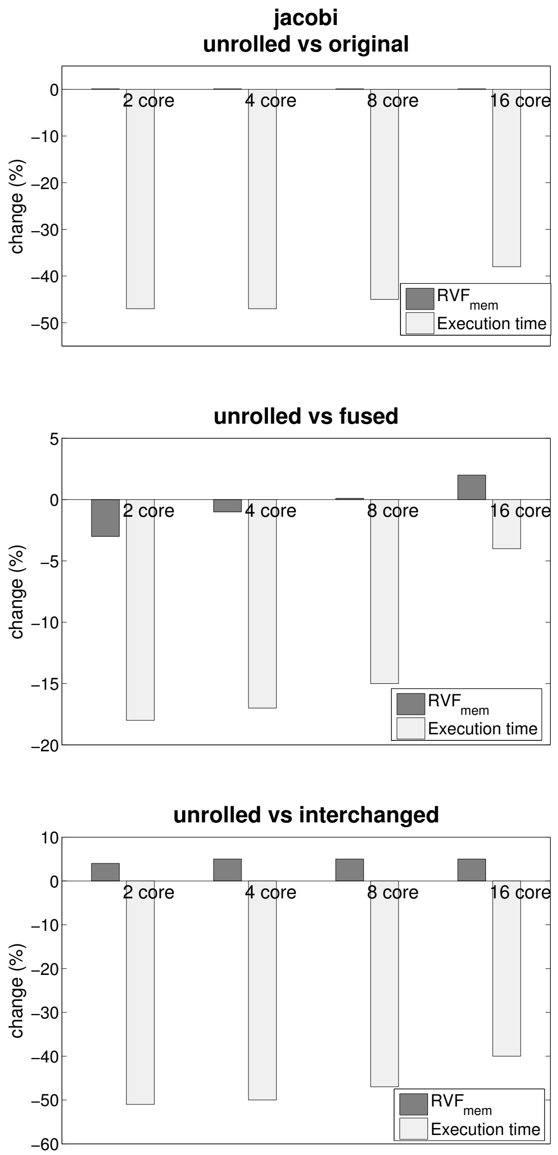

Fig. 3. The change in percentage of RVF values and execution time for loop transformations of Jacobi code

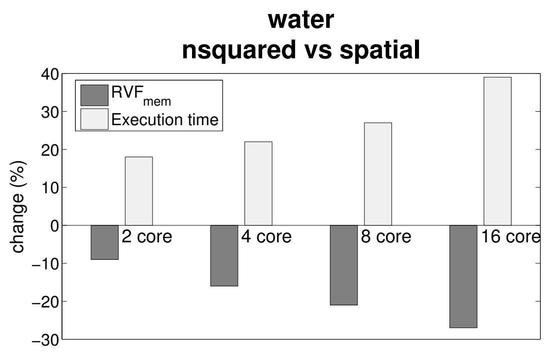

Fig. 4. The change in percentage of RVF values and execution time for versions of water simulation

ACKNOWLEDGMENT

This research was supported by The Scientific and Technological Research Council of Turkey (TUBITAK) with a research grant (Project Number: 108E035).

REFERENCES

[1] P. Shivakumar, M. Kistler, S. Keckler, D. Burger, and L. Alvisi, "Modeling the effect of technology trends on the soft error rate of combinational logic," in *Proc. International Conference on Dependable Systems and Networks*, 2002.

[2] T. M. Austin, "Diva: A reliable substrate for deep submicron microarchitecture design," in *Proc. 32nd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-32)*, 1999.

[3] P. M. Wells, "Mixed-mode multicore reliability," in *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[4] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "Swift: Software implemented fault tolerance," in *Proc. International Symposium on Code Generation and Optimization*, 2005.

[5] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang, "The case for a single-chip multiprocessor," in *Seventh International Symposium Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, 1996.

[6] D. Gizopoulos, M. Psarakis, S. V. Adve, P. Ramachandran, S. Kumar, and S. Hari, "Architectures for online error detection and recovery in multicore processors," in *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2011.

[7] M. L. S. Gal-On, "Measuring multicore performance," *IEEE Computer*, vol. 41, no. 11, 2008.

[8] M. Kulkarni, V. Pai, and D. Schuff, "Towards architecture independent metrics for multicore performance analysis," in *Proc. SIGMETRICS*, 2010.

[9] V. Sridharan and D. R. Kaeli, "Eliminating microarchitectural dependency from architectural vulnerability," in *Proc. IEEE 15th International Symposium on High Performance Computer Architecture*, 2009.

[10] I. Oz, H. R. Topcuoglu, M. Kandemir, and O. Tosun, "Quantifying thread vulnerability for multicore architectures," in *Proc. 19th Euromicro International Conference on Parallel, Distributed and Network-Based Computing (PDP)*, 2011.

[11] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," in *Proc. Annual IEEE/ACM International Symposium on Microarchitecture*, 2003.

[12] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex fourier series," *Mathematics of Computation*, vol. 19, no. 90, 1965.

[13] A. Gupta and V. Kumar, "The scalability of fft on parallel computers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 8, 1993.

[14] H. Dursun, K.-I. Nomura, L. Peng, R. Seymour, W. Wang, R. K. Kalia, A. Nakano, and P. Vashishta, "A multilevel parallelization framework for high-order stencil computations," in *Proc. International Euro-Par Conference on Parallel Processing*, 2009.

[15] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan, "Effective automatic parallelization of stencil computations," in *Proc. Programming language design and implementation (PLDI)*, 2007.

[16] S. Kamil, K. Datta, S. Williams, L. Oliker, J. Shalf, and K. Yelick, "Implicit and explicit optimizations for stencil computations," in *Proc. Workshop on Memory system performance and correctness*, 2006.

[17] L. Renganarayana, M. Harthikote-Matha, R. Dewri, and S. Rajopadhye, "Towards optimal multi-level tiling for stencil computations," in *Proc. Parallel and Distributed Processing Symposium*, 2007.

[18] D. Frenkel and B. Smit, *Understanding Molecular Simulation: From Algorithms to Applications*. Academic Press, 2001.

[19] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: Characterization and methodological considerations," in *Proc. 22nd Annual International Symposium on Computer Architecture*, 1995.

[20] J. P. Singh, W. Weber, and A. Gupta, "Splash: Stanford parallel applications for shared-memory," in *Technical Report, Stanford University*, 1991.

[21] S.Magnusson, M.Christensson, J.Eskilson, D.Forsgren, G.Hallberg, J.Hogberg, F. Larrson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *IEEE Computer*, vol. 35, no. 2, 2002.