Compatibility Study of Compile-Time Optimizations for Power and Reliability

Ghazaleh Nazarian, Christos Strydis and Georgi Gaydadjiev
Computer Engineering Laboratory, Electrical Engineering Dept.
Delft University of Technology, Postbus 5031, 2600 GA, Delft, Netherlands
Email: {G.Nazarian, C.Strydis, G.N.Gaydadjiev} @ tudelft.nl

***Abstract*—Historically compiler optimizations have been used mainly for improving embedded systems performance. However, for a wide range of today's power restricted, battery operated embedded devices, power consumption becomes a crucial problem that is addressed by modern compilers. Biomedical implants are one good example of such embedded systems. In addition to power, such devices need to also satisfy high reliability levels. Therefore, performance, power and reliability optimizations should all be considered while designing and programming implantable systems. Various software optimizations, e.g., during compilation, can provide the necessary means to achieve this goal. Additionally the system can be configured to trade-off between the above three factors based on the specific application requirements. In this paper we categorize previous works on compiler optimizations for low power and fault tolerance. Our study considers differences in instruction count and memory overhead, fault coverage and hardware modifications. Finally, the compatibility of different methods from both optimization classes is assessed. Five compatible pairs that can be combined with few or no limitations have been identified.**

Keywords- **compiler optimization, low power, fault tolerance.**

I. Introduction

Microprocessors are used in an expanding range of applications, and the requirements for the hardware and software components are changing continuously. Biomedical implants are an interesting instance of such systems. Due to the importance of their application, reliability is the most significant design issue. Moreover, the limited battery capacities of such devices, make power consumption another important factor, while computational performance remains a valid concern.

Traditional hardware techniques for reliability optimization, performance improvements and power reduction usually conflict with each other: one factor may have negative impact on the remaining two. In fields with multiple design constraints such as biomedical implants aiming at reliability, low power and performance at the same time, the deployment of such techniques is not straightforward.

The advantage of software solutions over hardware ones is the portability to different hardware platforms without requiring significant (or any) hardware modifications. Furthermore, by using software optimizations, the instruction flow at runtime can be adjusted to achieve a desirable trade-off between reliability, power and performance based on dynamic application needs. However, at the software level, not all optimization methods are fully compatible. The compatibility between compiler-optimization techniques for low power and reliability depends on criteria such as performance and memory overheads. The goal of this paper is to highlight the compatibility of reliability- and power-optimization techniques. In a nutshell, the contributions of this work are:

- Categorising reliability-related optimizations based on the targeted errors, level of abstraction and checking method;
- Categorising optimization methods for power reduction based on power-consumption sources;
- Analysis and quantitative comparison of each technique in terms of performance (instruction count) overhead, memory overhead and hardware modifications;
- Proposing hybrid optimizations for power reduction and reliability based on the results of this analysis.

The rest of the paper is organized as follows: Section II introduces reliability-optimization schemes for hardening the program execution against transient hardware faults. The methods for optimizing power consumption are explained in Section III while Section IV discusses the compatible and contradictory methods between reliability and power optimizations and Section V concludes the paper and discusses promising directions for future research.

II. Reliability optimizations

Reliability of embedded processors is threatened by permanent and transient faults. End-of-production testing is used to detect permanent faults, while run-time testing is required to cope with transient faults. In this paper, transient faults are targeted and Single-Event Upset (SEU) used as fault model. The consequence of SEUs can be two error types in the executing program; **Data errors** and **Control-flow errors**. The consequence of data errors is storage of erroneous data in the memory or a register, while the consequence of control-flow errors is a change in the correct order of program-execution flow [1]. Related work on reliability optimization targeting one of the two or both types of errors hereafter are explained and are analyzed in Table II.

A. Control-flow checking

Software control-flow checking ensures the correct program-execution order. These methods use *signature monitoring (SM)* schemes.

Definitions: In SM methods, two notions are used; program *Basic Blocks* (BB) and *Control-Flow Graphs* (CFG).

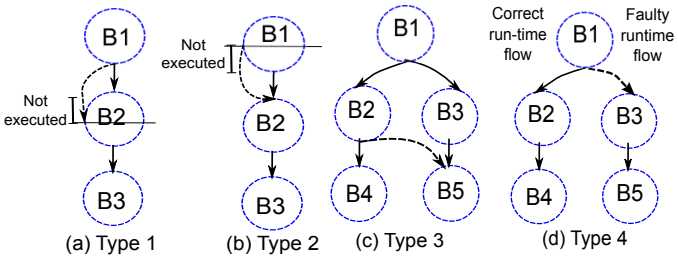

Fig. 1: Different targeted error types in signature monitoring.

Basic blocks: are branch-free sections of the program [2]; a set of consequent instructions or statements where only the last instruction (statement) can be a branch, and only the first instruction (statement) can be a branch destination.

CFG: of a program represents the correct order of basic-block execution. In the CFG, each node corresponds to a BB and an edge between two nodes denotes a jump from one BB to another. SM methods check BBs execution order using the program's CFG.

Targeted errors: There are four error types affecting the program control flow as depicted in Figure 1. Type (1) includes errors which cause skipping the initial part of a block. In type (2) faults, the epilogue section of a BB is omitted. Faults of type (3) cause erroneous jumps from one BB to an illegal BB. Faults affecting the conditional-branch argument are of type (4). These type of faults cause faulty branches[1].

Signature monitoring methods: The basic idea of SM techniques is to have a static signature for each BB and only one global dynamic or Run-time Signature (RS). The content of the static signatures is defined before run-time (at compile time), while the dynamic signature is calculated at run-time whenever the program execution reaches to a new BB in the CFG. By comparing the two signatures after each control-flow transfer, the correct run-time execution order of BBs is checked. RS can be calculated by a dedicated hardware (e.g., a watchdog processor) or by software (e.g., compiler or a watchdog task). Compiler-assisted version embeds dedicated functions for RS calculation in predefined points of the BB, the so-called *Signature-Generating Function (SGF)*. After updating the RS, a function called *Signature-Monitoring Function (SMF)* checks the consistency between the current BB signature and the RS. Depending on the SGF and SMF, some methods require storage of static parameters for each BB at compile time holding information about successors or predecessors of each BB. These parameters affect the memory and performance overheads.

Well-known, previously proposed SM methods with the selected SGFs, SMFs and the corresponding static parameters are presented in Table I. The static signature of the current BB is denoted by "*i*" (S_i). Predecessors signatures are denoted by "*pre1*" or "*pre2*" (e.g., S_{pre1}); and successors signatures are denoted by "*nxt1*" or "*nxt2*" subscripts (e.g., S_{nxt1}). Static parameters stored at compile time are indicated with "*P*". Static parameters may belong locally to a BB. In this case, they are also denoted by an "*i*" subscript (P_i). Static parameters that are global do not have an "*i*" subscript (e.g., P_2 in CFCSS [2]).

[1]They exist in the CFG, but the wrong edge is taken due to a faulty conditional argument.

B. Data-value-checking

Two types of methods are proposed for checking data errors; *data-duplication-with-comparison* and *executable assertions*.

Targeted errors: Data-value errors affect the system in three ways; (a) Erroneous data stored in memory or registers; (b) Erroneous instruction execution due to change of opcode; (c) Errors in conditional branches due to faulty data value of the condition argument. This data-value error causes type(4) control-flow errors (Figure 1(d)).

1) Data-duplication-with-comparison: The basic idea of these methods is to save a duplicate version of the original data at compile time, called **shadow** [8], and to compare the original and shadow versions at run-time in critical points. Critical points are places where the program output is written to the memory or the execution flow of the program is determined. Thus, three points in the program to insert comparisons are: Before "*store*" instructions; before "*branch*" instructions and before system calls (external libraries). Well-known methods proposed in this category are EDDI [8] at instruction level and ([9] and [6]) at source-code level.

2) Executable assertions: By adding extra statements to the program, the validity of specific constraints is tested. The added statements for testing are called executable assertions. An example is to check *outputs* and *state variables* with executable assertions [10]. Compile time assertions with AND/OR operatots proposed in [11] mask errors and protect variables with statically known values.

C. Data and control-flow checking

The methods targeting both data and control-flow errors are either using Error-Capturing Instructions (ECI) or combine data-value and control-flow checking as a hybrid technique.

1) Error-capturing instructions (ECI): ECIs are special instructions residing in memory locations which are not reachable during normal program execution. Targeted control-flow errors by ECIs are errors causing execution to diverge temporarily/permanently from the correct execution and targeted data errors, are the ones causing a read/write to a wrong location of the memory. ECIs proposed in [7] are of two types: A software-interrupt instruction or a branch instruction forming an infinite-loop along with a watchdog timer.

2) Hybrid fault-detection: detects both data and control-flow errors, [12], [13] and [6]. Hardware support can also be used to reduce the performance overhead and code size by reducing the amount of data duplication ([12] and [13]). SWIFT [12], combines a modified version of EDDI [8] for data-error checking and a modified version of CFCSS [2] for control-flow checking. This method protects memory elements by ECC and parity bits, thus the need for duplicating "*store*"

[2]P_i contains the signatures product of all legal successors of the BB.
[3]P_{3i} contains the signatures product of all legal predecessors of the BB.

SM METHODS	SGF	SMF	ADDITIONAL STATIC PARAMETERS
CFCSS [2]	$RS = RS \oplus P_{1i}$ $RS = (RS \oplus P_{1i}) \oplus P_2$	$ifRS! = S_i \ br \ error$	$P_{1i} = S_i \oplus S_{pre1}$ $P_2 = \begin{cases} 0000 & \text{in predecessor 1} \\ S_{pre1} \oplus S_{pre2} & \text{in predecessor 2} \end{cases}$
ECCA [3]	$RS = P_i + (RS - S_i)$	$RS = \frac{S_i}{\overline{RS\%S_i} \cdot (RS\%2)}$	$P_i = \prod S_{nxt}$ [2]
YACCA [4]	$RS = (RS \ \& \ P_{1i}) \oplus P_{2i}$	$If\,(P_{3i}\%RS) \quad error\,()$	$P_{1i} = S_{pre1} \bar{\oplus} S_{pre2}$ $P_{2i} = S_{pre1} \ \& \ (S_{pre1} \bar{\oplus} S_{pre2}) \oplus S_i$ $P_{3i} = \prod S_{pre}$ [3]
CCA [5]	$setRS_1 = S_{1i}$ $enqueueS_{2i}$	$dequeueRS_2$ $ifRS_1! = S_{1i} \quad error()$ $ifRS_2! = S_{2i} \quad error()$	not required
[6]	$setRS = S_i$	$ifRS! = S_i \quad error()$	not required
ACFC [1]	$RS = RS \oplus MASK$	$ifRS! = CONSTANT \ \ error()$	not required
BSSC [7](SM part)	$call\ entry\ routine$	$call\ exit\ routine$	not required

TABLE I: SGFs, SMFs and additional static parameters of SM methods for RS generation

METHODS	OVERHEADS		DETECTED FAULTS
	memory (bytes)	inst. count	
CFCSS [2]	2.5	22	CF(1,3)
ECCA [3]	3.5	76	CF(1,2,3)
YACCA [4]	4.5	48	CF(2,3)
CCA [5]	2	216	CF(1,2,3)
Rebaudengo [6]	28	51	data & CF(1,4)
ACFC [1]	0	7	CF(1,3)
EDDI [8]	28	33	data & CF(4)
SWIFT [12]	2	76	data & CF(1,2,3)
Rebaudengo [9]	20	35	data & CF(4)
ECI&BSSC [7]	-	-	CF(1,2,3,4) & data

TABLE II: Analysis of reliability optimization methods

```
int main(int x)
{ int c=3;
    if (x== 5)
        x= c+2;
    else
        x= c-2;
    c= x;}
```

(a)

```
int main(int x0, int x1)
{ int c0=3;     int c1=3;
    if(x0==5)
      if(x0 != x1)    error();
      x0= c0+2;       x1= c1+2;
      if(c0 != c1)    error();
    else
      x0= c0 - 2;     x1= c1 - 2;
      if(c0 != c1)    error();
      c0= x0;         c1= x1;
      if(x0 != x1)    error();
return 0;}
```

(b)

Fig. 2: (a) Source code, (b) Modified code based on [9]

instructions is eliminated. By having CFCSS for control-flow checks, the duplicated branch instructions are also not necessary. Moreover, SMFs for control-flow checking are applied only in BBs with "*stores*".

D. Analysis of reliability optimizations

Table II depicts the overheads and fault coverage of the related works presented in Section II. Without loss of generality, we have used a PowerPC (PPC405) as our target architecture when calculating the overheads.

1) Signature-monitoring overhead analysis: In Table II the memory and instruction-count overhead of applying different SM methods are estimated for the code presented in Figure 2(a). The equivalent assembly code of this snippet has 4 BBs, thus the calculated instruction count and memory overhead for each BB is multiplied by 4. The memory overhead of signature-monitoring schemes is estimated based on the number of bits required to store the static signatures and parameters[4]. Instruction-count overhead is calculated by counting the corresponding instructions to SGF/SMF.

2) Data-value check overhead analysis: For estimating instruction-count overhead of data-duplication-with-comparison methods at instruction level, the code in the Figure is compiled to PPC405 instructions and the method is applied to the obtained instructions, and finally, added instructions are counted. Figure 2(b) is the optimized code at source-code level using the method in [9]. This code is also compiled for PPC405 and the total number of extra instructions are calculated. Memory overhead is estimated by counting the total number of "*store*" instructions. The reason for counting "*store*" is that with each shadow "*store*" instruction an extra memory location is required.

[4]The details over all calculations can be obtained from [14].

III. Power consumption optimizations

Dynamic power consumption in CMOS circuits is the major source of power dissipation (given by: $P_{dyn} = \alpha \cdot C \cdot V^2 \cdot f$) [15]. The total energy consumption in a system is the product of the consumed power and the execution time (*t*) [16]; ($E_{dyn} = P_{dyn} \cdot t = P_{dyn} \cdot N \cdot T$). *N* is the number of clock cycles that the device is operating and *T* is the clock period. Thus the dynamic energy consumption is calculated as:

$$E_{dyn} = \alpha \cdot C \cdot V^2 \cdot N \quad , (T = 1/f) \tag{1}$$

This formula shows the total energy can be reduced in three ways; reduction of the transition density (α); reduction of the operating voltage (*V*) and reduction of the number of cycles the device is operating (*N*). A figure of merit describing the density of bit transitions in a processor executing a consecutive set of instructions is the *Hamming distance*.

Based on measurements performed in [16], three subsystems are identified as the main sources of power consumption in the system: (1) Bus lines driving the off-chip storage elements; (2) Processing units and (3) Pipeline latches.

In the rest of this section, related work is organized in two main categories; the ones requiring special hardware support (or modification) and the ones that are independent of the underlying hardware. At the end of the section, Table III shows the targeted source of power consumption and the targeted factor to minimize (in formula 1) in each related work.

A. Hardware-dependent

1) Coding: Transmitted data on the buses can be coded in order to decrease bus-lines transitions. Gray coding is a well-known coding for which the Hamming distance between two consequent codes is constant and always equals one. It can be used to address the instructions in the memory since instructions in the memory are most of the time contiguous and the execution of programs is sequential. To use Gray-coding for addressing, memory should have Decimal to Gray-code decoder and encoder.

2) Sub-system shut-down: In order to decrease the number of active clock cycles of the processor or memory (*N* factor in equation 1), some compiler methods partially shut down idle parts of the module ([17] and [18]). However, this scheme is useful only when the corresponding parts are idle for a long period. There is a need for executing look-ahead algorithms at compile time to predict the periods in which the processing units or memory are active.

3) Dynamic Voltage Scheduling: schedules instruction execution while the operating voltage (*V*) is changed in Variable-Voltage Processors (VVP). In VVPs the voltage level can be tuned to high for high performance and to low for idle periods. The main challenge is to find a scheduling algorithm without significant performance overhead. In order to schedule the tasks, the Worst-Case-Execution-Time is considered. But in real execution, the tasks are executed faster and the slack time should be divided between other tasks, [19] and [20].

B. Hardware independent

1) Increasing locality: Proper mapping of data into the memory reduces the accesses to the external memory and bus transitions. An example of re-mapping data in the memory is interleaving array elements of multiple arrays into a single array, [21]. Mapping two arrays A and B to D is shown below:

$$\begin{array}{l} For(i=1, i \leq N, i++) \\ C[i] = A[i] + B[i]; \end{array} \Longrightarrow \begin{array}{l} For(i=1, i \leq N, i++) \\ C[i] = D[2i-1] + D[2i]; \end{array}$$

The result is clustered data storage in the memory which is also useful in partial shut-down of inactive parts of the memory. Increasing locality is also possible by making the loops linear. Techniques for making the loops linear are: (1) loop unrolling; (2) loop fusion and (3) loop fission [22].

2) Scheduling: Instruction re-scheduling performed by the compiler changes the order of instruction execution. Data- (DDG) and control-dependency graphs (CDG) of the application source code have data and instructions as nodes, respectively. By using the information in the graphs, the scheduler reorders the instructions to reduce the Hamming distance. The scheduling introduced in [23] is dealing with VLIW instructions.

C. Analysis of Power-Optimizations

Table III compares the presented techniques for power optimization. Please note that precise results on overheads as in the earlier case for reliability optimizations, cannot be easily obtained without implementation of all proposed techniques on the same platform. This is out of the scope of this paper. However all methods should at least satisfy the constraint of $P(Compiler\_Optimizations) \gg P(HW\_Overheads)$.

METHODS	OVERHEAD	Power Source	Minimized Factor
[24] coding	compile time	Bus	α
[21] locality increase	none	Bus	α
[23] scheduling	performance	Processing units and latches	α
[19] DVS	performance	Overall	V
[18] part./compl. shutdown	performance	Overall	N

TABLE III: Analysis of power optimization methods

IV. Discussion

Based on the methods characteristics, techniques for power reduction and fault tolerance can be combined to form new hybrid optimization methods, some under certain limitations and some without any limitation.

A. Fully compatible methods:

- **Instruction re-scheduling and duplication** Added data-check instructions (*shadow* and *compare* instructions) for reliability optimization can be re-scheduled to reduce the Hamming distance. However, *compare* instructions should be scheduled before critical points of the program, such as "*store*" and "*branch*" instructions;
- **Loop flattening (unrolling or fusing) and signature monitoring:** SM schemes add extra SGF and SMF, which has an overhead in terms of power and performance. Loop-unrolling and -fusing, reduces the number of BBs. Thus, the overhead of added SGF and SMF will decrease.

B. Compatible methods with limitations:

- **Loop-fissing and signature monitoring:** In processors with small cache sizes, loop-fissing divides bigger loops not fitting in the cache, into smaller loops. However, by breaking a loop into several smaller loops, additional SGFs and SMFs are needed in each new loop. If SGFs and SMFs have memory references, extra SGFs and SMFs causes extra memory accesses. This issue plus the execution of SGF and SMF instructions cause an extra overhead in terms of power consumption;
- **Instruction re-scheduling and signature-monitoring (or executable assertions):** Re-scheduling the instructions for power reduction reorders instruction execution based on the minimal Hamming distance between the binary forms of consecutive instructions. Thus, it should be done when the binary format of the instructions are known (after the assembler/linker). In signature-monitoring and also executable-assertion techniques, special instructions for run-time execution are added in specific points of the code. But if these methods and instruction re-scheduling are implemented on the same code, the special added instructions may be reordered by instruction re-scheduling. A possible solution to this

limitation is to apply signature-monitoring (or executable-assertions) at later phases of compilation (after re-scheduling). Another solution is to restrict signature-monitoring instructions, and prohibit the scheduler to reorder them;

- **Partial shut down of memory and ECI (Error-Capturing-Instruction):** In order to detect control-flow errors, ECIs are placed in un-used memory locations. However, by shutting-down un-used locations of memory, [21], ECIs will not be executed. Possible solution for this issue is to remember memory locations that ECIs are added and mark them as active memory locations.

V. Conclusions

In this paper we have surveyed different compiler optimization methodologies for increasing reliability and reducing system power consumption. The techniques for improving the reliability were categorized into: signature-monitoring schemes, error-capturing-instructions, data-redundant methods and executable assertions. Power optimization methods were divided into scheduling, coding, increasing locality, subsystem shut-down and dynamic voltage scaling. Each of the optimization techniques from both classes was analyzed in terms of overheads and implementation issues. Furthermore, we investigated the compatibility of different methods from both classes. Based on our analysis, two promising combinations for embedded systems requiring both power and reliability optimizations are: instruction re-scheduling with instruction duplication and loop flattening (unrolling or fusing) with signature monitoring. These methods can be implemented in a compiler and do not require additional hardware support. Three additional pairs have also been identified with certain limitations: loop-fissing and signature monitoring, instruction re-scheduling and signature monitoring (or executable-assertions) and partial shut down of memory and ECI. Future work involves the implementation a new compiler optimization method considering both power and reliability. Our new method will exploit the above mentioned combinations.

References

[1] R. Venkatasubramanian, J. P. Hayes, and B. T. Murray, "Low-cost on-line fault detection using control flow assertions," in *9th IEEE On-Line Testing Symposium*. IEEE, July 2003, pp. 137–143.

[2] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Control flow checking by software signatures," *IEEE Trans. on Reliability*, vol. 51, no. 1, pp. 111–122, March 2000.

[3] V. N. Z. Alkhalifa, N. Krishnamurthy, and J. Abraham, "Design and evaluation of system-level checks for on-line control flow error detection," *IEEE Trans. on Parallel and Distributed Systems*, vol. 10, no. 6, pp. 627–641, June 1999.

[4] O. G. ad M. Rebaudengo, M. S. Reorda, and M. Violante, "Soft-error detection using control flow assertions," in *18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*. IEEE, November 2003, pp. 581–588.

[5] G. A. Kanawati, V. S. S. Nair, N. Krishnamurthy, and J. A. Abraham, "Evaluation of integrated system-level checks for on-line error detection," in *Proceedings of IEEE International Computer Performance and Dependability Symposium*. IEEE, September 1996, pp. 292–301.

[6] M. Rebaudengo, M. S. Reorda, M. Torchiano, and M. Violante, "Soft-error detection through software fault-tolerance techniques," in *Proceedings of the 14th International Symposium on Defect and Fault-Tolerance in VLSI Systems*. IEEE, 1999, pp. 210–218.

[7] G. Miremadi, J. Karlsson, U. Gunnejlo, and J. Torin, "Two software techniques for on-line error detection," in *22nd International Symposium on Fault-Tolerant Computing*. FTCS, July 1992, pp. 328–335.

[8] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Error detection by duplicated instructions in super-scalar processors," *IEEE Trans. on Reliability*, vol. 51, no. 1, pp. 63–75, March 2002.

[9] M. Rebaudengo, M. Reorda, and M. Violante, "A new software-based technique for low-cost fault-tolerant application," in *Annual Reliability and Maintainability Symposium*, 2003, pp. 25–28.

[10] J. Vinter, J. Aidemark, P. Folkesson, and J. Karlsson, "Reducing critical failures for control algorithms using executable assertions and best effort recovery," in *Proceedings of the International Conference on Dependable Systems and Networks*, 2001, pp. 347–356.

[11] J. Chang, G. A. Reis, and D. I. August, "Automatic instruction-level software-only recovery," in *Proceedings of the International Conference on Dependable Systems and Networks*. DSN, 2006, pp. 83–92.

[12] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "Swift: Software implemented fault tolerance," in *Int. Symposium on Code Generation and Optimization*, March 2005, pp. 243–254.

[13] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, D. I. August, and S. S. Mukherjee, "Design and evaluation of hybrid fault-detection systems," in *International Symposium on Computer Architecture*. ISCA, June 2005, pp. 148–159.

[14] G. Nazarian, "Technical report on overhead analysis of compiler optimizations for fault tolerance and low power," CE-TR-2011-01.

[15] V. Venkatachalam and M. Franz, "Power reduction techniques for microprocessor systems," vol. 37, no. 3, pp. 195–237, September 2005.

[16] V. Tiwari, S. Malik, A. Wolfe, and M. T.-C. Lee, "Instruction level power analysis and optimization of software," *J. VLSI Signal Process. Syst.*, vol. 13, no. 2-3, pp. 223–238, August 1996.

[17] M. B. Srivastava, A. P. Chandrakasan, and R. W. Brodersen, "Predictive system shutdown and other architectural techniques for energy efficient programmable computation," vol. 4, no. 1, pp. 42–55, March 1996.

[18] C.-H. Hwang and A. C.-H. Wu, "A predictive system shutdown method for energy saving of event-driven computation," vol. 5, no. 2, pp. 226–241, April 2000.

[19] D. Mosse, H. Aydin, B. Childers, and R. Melhem, "Compiler-assisted dynamic power-aware scheduling for real-time applications," in *In Workshop on Compilers and Operating Systems for Low Power*, 2000, pp. 1–9.

[20] F. Xie, M. Martonosi, and S. Malik, "Compile-time dynamic voltage scaling settings: opportunities and limits," in *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*. ACM, 2003, pp. 49–62.

[21] V. Delaluz, M.Kandemir, N.Vijaykrishnan, and I. Kolcu, "Compiler directed array interleaving for reducing energy in multi-bank memories," in *Proc. of ASP-DAC*, 2002, pp. 288–296.

[22] M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and W. Ye, "Influence of compiler optimizations on system power," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 9, no. 6, pp. 801–804, December 2001.

[23] C. Lee, J. K. Lee, T. Hwang, and S.-C. Tsai, "Compiler optimization on vliw instruction scheduling for low power," vol. 8, no. 2, pp. 252–268, April 2003.

[24] C.-L. Su, C.-Y. Tsui, and A. Despain, "Low power architecture design and compilation techniques for high-performance processors," in *Compcon Spring '94, Digest of Papers.*, February 1994, pp. 489–498.