Temperature-Aware Software-Based Self-Testing for Delay Faults

Ying Zhang[1] Zebo Peng[2] Jianhui Jiang[1*] Huawei Li[3] Masahiro Fujita[4]

[1] *School of Software Engineering, Tongji University, China*

[2] *Embedded Systems Lab, Linköping University, Sweden*

[3] *State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, China*

[4] *VLSI Design and Education Center, University of Tokyo, Japan*

yingzhang@tongji.edu.cn, zebo.peng@liu.se, jhjiang@tongji.edu.cn, lihuawei@ict.ac.cn, fujita@ee.t.u-tokyo.ac.jp

Abstract—**Delay defects under high temperature have been one of the most critical factors to affect the reliability of computer systems, and the current test methods don't address this problem properly. In this paper, a temperature-aware software-based self-testing (SBST) technique is proposed to self-heat the processors within a high temperature range and effectively test delay faults under high temperature. First, it automatically generates high-quality test programs through automatic test instruction generation (ATIG), and avoids over-testing caused by nonfunctional patterns. Second, it exploits two effective power-intensive program transformations to self-heat up the processors internally. Third, it applies a greedy algorithm to search the optimized schedule of the test templates in order to generate the test program while making sure that the temperature of the processor under test is within the specified range. Experimental results show that the generated program is successful to guarantee delay test within the given temperature range, and achieves high test performance with functional patterns.**

I. INTRODUCTION

High temperature has been one of the most critical factors to affect the reliability of computer systems [1]. It is common that at the normal temperature a computer system works correctly, but once the temperature rises up, many inexplicable problems would happen even though it has passed rigorous manufacturing tests. Many of these temperature-related problems are related to delay faults which happen under high temperature. With the increasing of temperature, the delays on circuits also increase, so that high temperature aggravates delay defects in the circuits. What's worse, the timing margin for high temperature or other problems shrinks significantly due to the increasing demand of the performance. The delay defect under high temperature is therefore a general problem for modern computer systems, and the problem has to be taken seriously, especially for reliability-critical systems. However, most current test methods don't address this problem properly, and it is thus necessary to propose a suitable test method for delay faults under high temperature.

The current manufacturing test methods are not suitable to test delay faults under high temperature. First, they often insert some DFT circuits into the circuit under test (CUT), and make CUT easy to be controlled and/or observed. As a consequence of this, the tests include many non-functional patterns that would never happen during real application. For delay faults, the manufacturing test has extremely high proportion of nonfunctional patterns, which may easily lead to the situation that a function-correct chip is classified as a faulty one by mistake. This is called over-testing, and it would cause unbearable economic loss [2]. Secondly, it may be dangerous to apply the manufacturing test for delay faults under high temperature. On one hand, the manufacturing test would have even worse over-test problem when the temperature rises up. That is because these nonfunctional patterns can excite untestable paths with longer delays easily, and their delay faults would be observed firstly when the temperature rises up. On the other hand, the manufacturing test is often developed to test a circuit in normal temperature [3] [4] instead of testing delay faults under high temperature. In manufacturing test, its signal transitions are usually several times of that of the normal application, and these extremely high signal transitions heat up the chip greatly, or even burn the chip [3]. Therefore, traditional manufacturing test method is not suitable for test delay faults under high temperature.

Software-based self-testing (SBST) has been a promising test method for processors that applies functional test patterns and can achieve comparable fault coverage as the full-scan method. SBST has already been proposed for delay faults under the transition delay fault model [5], and achieved more than 94% fault coverage. Meanwhile, some researchers propose SBST method for path-delay faults, and also achieve high test performance [6] [7]. Recently, automatic test instruction generation (ATIG) [8] [9] is presented to automatically generate SBST programs, and compact the test programs without fault coverage loss. Also, SBST is optimized to apply low-power or low-energy test for wireless sensor nodes [10]. However, SBST has not been applied for delay faults under high temperature.

In this paper, temperature-aware SBST is proposed that self-heats the processors to a high temperature range and effectively tests transition delay faults under high temperature. First, it automatically generates high-quality test templates through ATIG, and avoids the over-testing problem caused by nonfunctional patterns. Second, it exploits two effective power-intensive program transformations that guarantee to self-heat up the processors internally. Third, it applies a greedy algorithm to dynamically incorporate test template transformation with optimized test overhead and make sure that the temperature during test is within the specified range. To our knowledge, this is the first work on temperature-aware SBST targeting delay faults.

This paper is organized as follows. Section II covers the necessary background of software-based self-testing for delay fault under high temperature. Section III introduces delay test program generation through ATIG. Later, we exploit the power-intensive program transformations in Section IV. Section V discusses the greedy algorithm for temperature-aware test. Experimental results are analyzed in Section VI. Finally, we conclude this paper in Section VII.

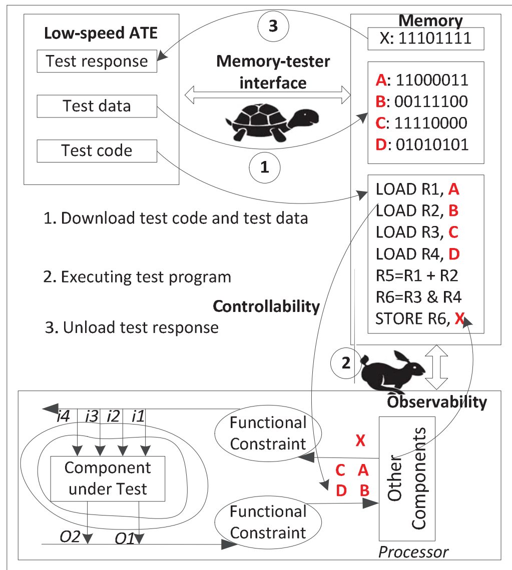
* To whom correspondence should be addressed.

This paper is supported in part by National Natural Science Foundation of China (NSFC) under grant No. (61432017, 61404092), in part by Jiangsu Prospective Research Project on Future Networks, and in part by the Fundamental Research Funds for the Central Universities (2013KJ036).

978-3-9815370-4-8/DATE15/©2015 EDAA

II. SBST FOR DELAY FAULT UNDER HIGH TEMPERATURE

Software-based self-testing has been a promising processor test method that applies normal programs to test both stuck-at and delay faults under the functional mode. When testing delay faults, SBST is executed according to Fig. 1(a) [11]. In this example, test data of a functional pattern pair of delay test are first stored in four registers via four *load* instructions. Second, it uses the first *add* instruction to initialize the CUT with *R1* and *R2*, and applies the second *add* instruction to test the delay faults at-speed with *R3* and *R4*. Third, it uses the *store* instruction and stores the result in *R6* for observation. In this way, SBST uses instruction pairs and tests delay faults successfully. The test programs generated with such instruction pairs are put together to form a test template, as shown in Fig. 1(b). Note that an extra *nop* instruction is inserted after the last *load* instruction because the fourth *load* instruction has data dependency with the second *sllv* instruction, a stall would happen before the *sllv* instruction, and the test pair would fail to detect the given defects.



(a) The Framework of SBST for Delay Faults

```
ADDIU $9  $0 0x0400
Label J0
LW  $10  $1  0x0000   Controlling Instructions
LW  $10  $2  0x0004   Bring in Extra Power
LW  $10  $3  0x0008
LW  $10  $4  0x000C
NOP
SLLV $1  $2  $5       Test Instructions
SLLV $3  $4  $6
SW  $10  $6  0x0004   Observing Instructions
ADDIU $8  $8  0x0010  Checker
BNE $8  $9  J0        Remove the Checker
```

(b) The Test Template for Instruction SLLV

Fig. 1. Software-Based Self-Testing for Delay Faults [11]

SBST would be one of the most suitable methods to test delay faults under high temperature, if it could create and maintain the high test temperature in the processor just through executing programs. This is because, first of all, SBST does not make any changes to the structure of the processors at all, and all of its patterns are functional. In this way, it avoids the over-testing problem, and it does not waste time to test many faults detected already by traditional manufacturing test methods. Second, SBST heats up the chip from the inside instead of the outside, so that it truly tests delay faults under high temperature conditions. In addition, it avoids the need to heat up the processor externally, which usually takes a long time, and it does not damage the insulating layer of the package, which would reduce the lifetime of the processor.

However, it is difficult to create and maintain high test temperature just through executing programs. On one hand, it is still an open problem to generate power-intensive SBST programs. In order to increase the power of an SBST program, let us analyze the test template in Fig. 1(b). It includes four parts, controlling instructions, test instructions, observing instructions, and the loop checker. First, the loop checker brings many stalls into the pipeline and degrades the density of signal transition greatly. Therefore, it is reasonable to remove these loop checker for higher power dissipation. Second, the controlling and observing instructions, consisting of *load* and *store,* cover a large proportion of the template. If these instructions have more power dissipation, the whole template would be more power-intensive. Of course, there are also other ways to increase the power consumption of an SBST program. On the other hand, because a power-intensive program may heat up the processor beyond the temperature threshold and damage the processor, the temperature must be maintained within the proper range.

Recently, several efficient test scheduling algorithms have been reported that control test temperature in a given range [12]. However, these algorithms work only in the context of core-based test with test access mechanisms, not for SBST. In the following section, we will describe our proposed technique, in details, to achieve temperature-aware SBST, and we will set the given temperature range at [105, 110] similarly as in [13], and use the testing of the ALU in the miniMIPS processor as an illustrated example, where the processor is the newest version without branch-prediction.

III. TEST PROGRAM GENERATION FOR DELAY FAULTS

We have developed an automatic test instruction generation (ATIG) approach to generate SBST programs with high coverage for delay faults. The developed method imposes functional constraint on the CUT, generates functional pattern pairs (instead of single patterns) for delay faults, and translates these pairs into instruction pairs, finally SBST programs. When applying SBST on an ALU, we assume first that the instruction pair consists of two same instructions, so that we can impose the functional constraint of the single type on the CUT, and then generate one test template to implement these test pairs. Later, we consider that the instruction pair contains different instructions.

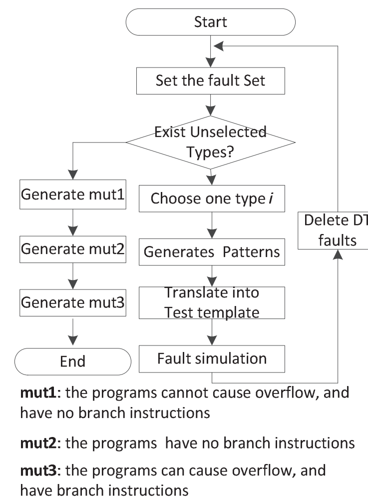


Fig. 2. Automatic Test Instruction Generation for Delay Faults on ALU

The flowchart of the ATIG algorithm is shown in Fig. 2. First, the algorithm sets all ALU faults as the initial fault set. Second, it checks if there is an unselected instruction type. If yes, it continues; or else it goes to the sixth step. Third, it

chooses an unselected instruction type, and generates functional test pairs under the functional constraint of the type. Forth, it applies a related test template for all these pairs, and stores the test data in the right address. Fifth, it starts sequential fault simulation with the test template, removes detected faults (DT) from the fault set, and then returns to the second step. In addition, the template that can't detect new faults is removed from the final programs. Sixth, the algorithm starts to generate programs for the test pair containing different instructions, denoted as mut1, mut2, and mut3. Finally, the algorithm obtains the test templates, including mut1, mut2, and mut3.

The proposed ATIG algorithm can achieve very good test quality on the ALU component, and will never cause any over-testing problem. As shown in Fig. 5(a), the fault coverage of the generated SBST program for the miniMIPS ALU rises up continuously with the increasing of the test templates, and finally arrives at 97.91%, which is more than that of the most advanced SBST programs [4]. Furthermore, it has higher test performance than traditional manufacturing test, even though the latter can achieve more than 99% fault coverage. That is because it does not test the faults that never happen under the functional mode, and it thus avoids the over-test problem. For example, the falling delay fault on the signal *overflow* would not happen under the functional mode, because the test program is interrupted once an overflow happens. Additionally, many delay faults that are excited when an instruction follows a branch instruction would not emerge either. Because stalls are inserted after every branch, and that delay faults cannot happen under the functional mode.

IV. Power-Intensive Program Transformation

To heat up the processor just through executing programs themselves, the original SBST programs have to be transformed so that they will be power-intensive. As the dynamic power of programs depend on the density of signal transitions, either the SBST programs are transformed to be more compacted with less stalls in their pipeline, or they bring in some extra power dissipation, in order to increase their power consumption.

A. Unrolling Loops for High Power

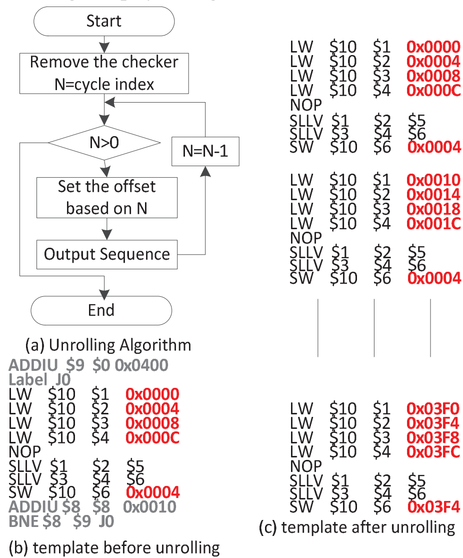


Fig. 3. Unrolling Loop in Test Template

Unrolling loops in programs is an effective transformation to enhance the power dissipation. Every loop contains a checker to decide whether or not the loop should continue, and the checker inevitably brings some stalls in the pipeline. If every loop is unrolled, the checker is no longer required, and no extra stalls will be encountered. As a result, unrolling loops can reduce the execution time of a program, and enhance the density of signal transitions. As loops often cover a large proportion of the executing period of programs, unrolling loops can increase power dissipation greatly.

The detail of the loop-unrolling transformation is shown in Fig. 3. At first, the algorithm removes the checker from the template, and it also obtains the iteration index of the loop as the variable *N*. Later, it resets the offset of the instruction *store* (or *load*) for the right memory address, and outputs the instruction sequence once. Then *N* is decremented, and the algorithm returns to the previous step. The algorithm repeats the steps until *N* is 0. For example, the algorithm replicates the template for instruction *sllv* 64 times to unroll the whole loop, and requires 704 words of storage space including 256 words for data. The additional storage space is the overhead that we have to pay for using this transformation.

B. Exciting Cache Miss for High Power

Exciting cache miss in programs is another effective transformation to enhance the power dissipation of modern processors. In a modern processor, as cache device often covers a large portion of the chip area, once its page is renewed, it would lead to large power dissipation. To excite cache miss, the program has to set the page *tag* of its *store* or *load* according to the following situations. Assume the cache has *n* pages (*C[0]…C[n-1]*), the whole memory has *m* pages (*M[0]… M[m-1]*), and that FIFO algorithm is used to replace cache pages. In addition, the set associate cache contains *s* sets, and each set has *k* pages($s*k=n$). First, in the direct mapped cache, if the *tag* of the page *M[t]* is not equal to the *tag* of *C[(t mod n)]* in equation 1 of Fig. 4, a cache miss happens. Second, in the fully associative cache, only if the *tag* of the page *M[t]* is different from the *tag* of each cache page shown in equation 2 of Fig. 4, a cache miss would happen. Third, in the set associate cache as the page *M[t]* is mapped into the p_{th} ($p=(t/k) \bmod s$) cache set, if the tag of the page *M[t]* is different from the tag of each cache page in the cache set $[p*k, p*k+k)$ shown in equation 3 of Fig. 4, a cache refresh would happen.

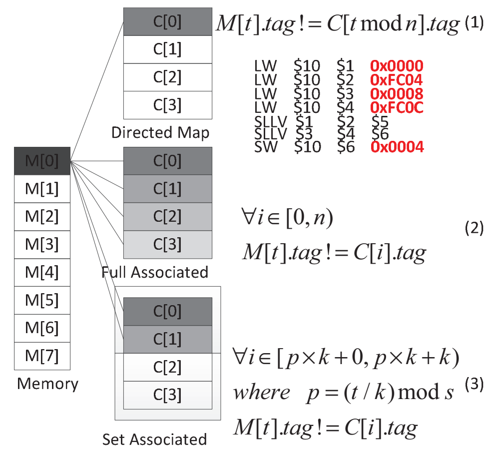


Fig. 4. Exciting Cache Miss in Test Template

Cache misses can be excited through modifying test programs slightly, and thus causing large power dissipation. Assume that a data cache based on the direct mapped style is inserted into the miniMIPS processor. To excite cache misses, the transformed program makes the odd load/store instruction access one memory page *M[i]*, and the even load/store instruction access another memory page *M[j]*, while these two pages are mapped to the same cache page. As shown in Fig. 4,

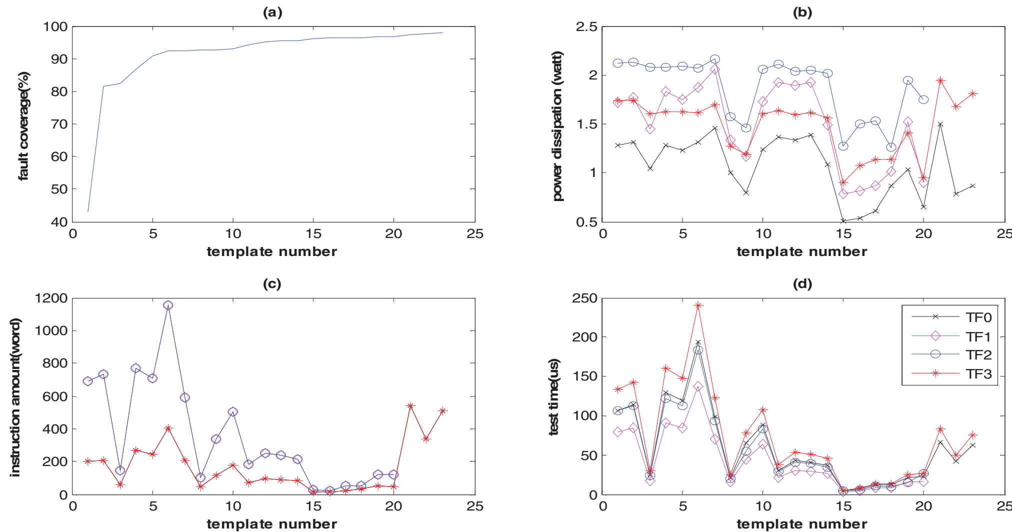


Fig. 5. The Fault Coverage, Power Dissipation, Test Time, and Instruction Amount of Self-Heating SBST Templates, where TF0 is the original program, TF1 is the program after unrolling its loops, TF2 is the program after both unrolling loops and exciting cache misses, TF3 is the program after exciting cache misses. In addition, the original template 21, 22, and 23 (mut1, mut2, and mut3) are unrolled templates which just have one transformation TF3.

the first and second *load* instructions in the program store the base address in the same register R10, while their offsets are 0x0000 and 0xFC04 respectively. In this case, these two instructions access different memory pages that are mapped to the same cache page, and thus a cache miss happens. With this transformation, whenever load or store happens, its related memory page is absent in the cache, and that page would replace the related cache page.

C. Impact on Power, Test Time, and Storage Space

To analyze the impact of the transformations discussed above, an experimental system is built up to evaluate the power dissipation, test time, store space of the self-heating program generated by such transformations as well as the original program. In the experiment, the miniMIPS processor with a direct mapped data cache unit is synthesized with a 90ns technical lib as the experimental bench, and its frequency is set as 10MHZ. The experiment reports the test time and storage space repaired to execute each test template. Later, the system extracts the power trace from the Synopsys tool Primepower, and then calculates the average power dissipation.

As shown in Fig. 5(b), the program transformations we have discussed increase the power dissipation obviously. First, TF1 that unrolls all loops increases the template power significantly by 40.2% on average. That is because TF1 removes the pipeline stalls resulted from the loops, makes the pipeline execution more efficient, and finally increases the density of signal transitions. Second, TF3 that excites cache misses of the *load* or *store* instruction also enhances the power dissipation by 40.0% on average. That is because cache access often take a large power proportion of modern processors, and frequent cache misses indeed bring in huge power dissipation. Third, these two methods can work together, and achieve the highest power dissipation, as shown by the TF2 curve. In Fig. 5(b), TF2 often doubles the power dissipation of the original template. Finally, note also that the power dissipation varies greatly among different templates, and some appropriate schedule is therefore required for temperature-aware test. In general, because the program transformations enlarge the power dissipation greatly, it is feasible to self-heat up processors internally.

The storage space and test time of these transformed templates are shown in Fig. 5(c) and 5(d), respectively. Unrolling loop is an expensive method to increase power, as it has to replicate the template N times where N is equal to the iteration index of the loop. In Fig. 5(c), the line of TF1 and TF2 (the same line), associated with loop unrolling, is much higher than the line of TF0 and TF3 (the same line), corresponding to the original program and the one with only exciting cache misses. Exciting cache miss is a much better transformation in this regard, since it does not cost any extra storage space. That is because this transformation just changes the offset of instruction *load* or *store*. Therefore, the curve for TF0 and that for TF3 become the same line as shown in Fig. 5(c), since TF3 only changes the offset of the *load* and *store* instructions of TF0. The same applies also to TF1 and TF2.

On the other hand, exciting cache miss brings in some test time overhead, while unrolling loop reduces test time greatly. In Fig. 5(d), the line indicating the test time of TF3 (exciting cache misses only) is the highest one, and the increasing ratio of time overhead varies greatly. That is because the cache misses cost extra clock cycles to refresh the cache, and the increasing ratio depends on the frequency of *load* and *store* in the template. For instance, the *sllv* template (template 3) that contains five of these memory access instructions has higher ratio than the *mfhi* template (template 15) with only three of such instructions. In Fig. 5(d), the line of TF1 (loop unrolling only) is the lowest one. That is because unrolling loop removes the checker, makes the pipeline more compacted, and thus reduces test time. The test time of TF1 is so low that even if loop unrolling and exciting cache misses work together, their test time is still lower than that of the original template. In conclusion, with reasonable overhead, an SBST program can be transformed to be power-intensive with the two simple transformations discussed in this section.

V. TEMPERATURE-AWARE SBST

The last problem to address before we will have a complete temperature-aware SBST so that an processor will be tested for the worst-case delay defects is to schedule all test templates so that they will be executed when the processor temperature is always within the specified range. This is a difficult problem, since *N* templates have *N* different power dissipations, and the different order of them will lead to different power curves, and consequently different temperature traces. The total amount of possible schedules that result in different temperature traces is

equal to the factorial of N. Furthermore, since each test templates can be transformed with the two transformations discussed in Section IV, there are four different implementations for each template. It is, in general, an NP problem to find out a feasible schedule that satisfies the temperature specification. On top of this, the feasible schedule is required to be optimized for as low test overhead as possible, and there are several different trade-offs to be made. The self-heating transformations can be used to enhance the power dissipation, but they usually require more cost of storage space or test time. For instance, unrolling loops multiplies the instruction amount of the test template, so it would make the SBST program much bigger.

A greedy algorithm is therefore proposed to dynamically incorporate test template transformation with optimized test overhead, and guarantee the test temperature range. The basic idea is “low-overhead transformation first, and high temperature first”. Specifically, the algorithm first searches the feasible templates from the low-overhead transformations, where the overhead of storage space has priority over the time overhead because an SBST program is executed at speed. It selects then the template that can heat the processor to the highest temperature (HT), because it provides more margins for the low-power templates, and this helps to find out a feasible schedule quickly.

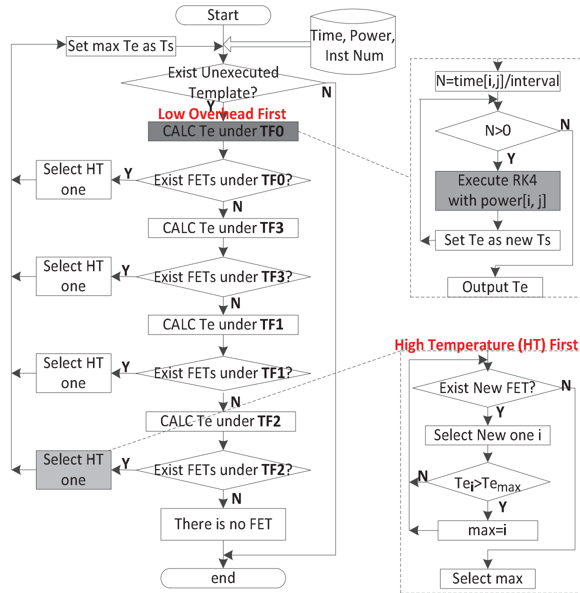


Fig. 6. The Greedy Algorithm for Temperature-Aware SBST, where HT, FETs, CALC, and interval stand for “Highest-Temperature”, “feasible template”, “calculate”, and “sample interval in RK4”, respectively, while Time[i, j] and Power[i, j] stand for the executing time and power of the ith template in the jth transformation.

The flowchart of the greedy algorithm is shown in Fig. 6. First, it applies extra power-intensive templates to heat the chip up to the lower limit of the given temperature range, which is then used as the starting temperature *Ts*. Second, it checks if there are still some unexecuted templates left to be scheduled. If there are no such templates, the algorithm has obtained an optimized schedule and it will terminate; or else, the algorithm continues. Third, according to the “low-overhead transformation first” policy, it searches, from the original template TF0, the feasible templates that keep the temperature within the specified range. Specifically, it applies the 4th order runge-kutta method *(*RK4*)* in Hotspot [14], and calculates the ending temperature *Te* of each unexecuted template in TF0. As shown in the upper subfigure of Fig. 6, the algorithm gets the test time *time[i, j]* of the i_{th} template in TF0 (*j=0*), and sets the cycle index *N* as *time[i, j]/interval*, where *interval* is the sampling interval of RK4. Later, the algorithm uses the power *power[i,j]* of the i_{th} template in TF0 (*j=0*), and calculates *Te* after the first interval. Then, it sets *Te* as *Ts* for the next interval. After repeating these steps *N* times, it obtains *Te* of the template. In addition, the template that makes *Te* within that range is called feasible template (FET). If there is only one feasible template, it is chosen and inserted into the schedule immediately. If there are several feasible templates, the rule “high temperature first” is then used. In the lower subfigure of Fig. 6, the algorithm checks these feasible templates one by one. If *Te* of one template is higher than the current max *Te* (Te_{max}), it labels the template and its *Te*. Finally, it finds out the template with the highest *Te*, chooses that template, and inserts it into the schedule. In the above two situations, the chosen template is marked as “executed”, its *Te* is used as the new *Ts* for the next scheduling step, and the algorithm returns to the second step. Or else, if there are no feasible templates, the algorithm continues. Because storage space has higher priority to test time, the algorithm searches first feasible templates from TF3. After that, the algorithm will search feasible templates from TF1 to TF2. Finally, if none of the template is feasible, there would be no schedule satisfying the requirement of the given temperature-aware test.

VI. Experimental Results

In this section, the experiment on the miniMIPS ALU and its results of the proposed temperature-aware SBST technique are described in detail. In the experiment, the processor at the room temperature (25 degree) is heated up to 105.02 degree by executing a 2-watt program. The proposed algorithm generates a feasible schedule that makes the temperature within the specified temperature range of [105, 110]. For comparison purpose, three references are used: the first executes the original templates starting directly from the room temperature; the second execute the templates in TF2 after the temperature arrives at 105.02 degree; and the third applies Hotspot to simulate the temperature of the generated schedule by the proposed algorithm.

The proposed greedy algorithm is successful in finding out a feasible schedule for the given temperature-aware test for the miniMIPS ALU which consists of 23 test templates. It takes advantage of the RK4 function to calculate *Te* of the unexecuted templates in TF0 first. For this, its executing time is 107 times of the RK4 sampling interval, so the algorithm loads in its average power, repeats RK4 107 times, and finally gets the maximal value of *Te*. Unfortunately, after calculating each template in TF0, the algorithm finds no feasible template that keep the temperature *Te* within the given temperature range [105, 110]. According to “low overhead transformation first”, it searches then feasible templates in TF3, and finds out 6 feasible templates that satisfy the temperature requirement. According to “high temperature first”, template 20 that raise up the temperature with the maximal value of 1.67 degree is chosen to the schedule, and its *Te* (105.02+1.67=106.69 degree) would be set as *Ts* for the next scheduling step. For the new starting temperature of 106.69 degree, the algorithm finds out several feasible templates in TF0. The same to the previous steps, it selects template 18 in TF0 that does not bring in any extra overhead, but the temperature falls to 106.19 degree, since executing this temperate leads to a temperature reduction of 0.5 degree. For this example, the algorithm quickly find out an optimized feasible schedule, as shown in Fig. 7(a).

The experimental results also show that the proposed algorithm guarantees the temperature of the processor to be within the given range during the whole execution of the test templates, and implements therefore temperature-aware SBST

successfully. Fig. 7(b) shows the temperature traces of the temperature-aware SBST program generated by the proposed algorithm and the three references. First of all, the temperature trace of the proposed algorithm (red) is within [105, 110] completely, which satisfies therefore the temperature specification. Additionally, this trace is only slightly above the 105 degree line. This means the algorithm guarantees the lower temperature limit, and, at the same time, avoids using the high-overhead transformations that will lead to large temperature increase. It thus finds an optimized schedule without too much overhead. Second, the temperature trace of the first reference rises up, but it is far away from the given temperature range. This means that, without temperature-aware SBST, the original test templates, will be applied at a very low temperature level, and therefore some delay defects will be not detected. Third, the temperature trace of the second reference is mostly above the specified upper temperature limit. Therefore it is harmful to use always the power-intensive test templates directly, because this will lead to too high temperature for the processor and it may even burn it. It is thus necessary to control the temperature of the SBST program. Finally, the temperature trace of the third reference, generated directly by the Hotspot simulator is coincident with the temperature trace of the proposed algorithm. It means that the temperature prediction of the proposed algorithm is accurate and therefore the proposed temperature-aware SBST technique will be able to deliver the temperature guarantee with the same accuracy as Hotspot can provide.

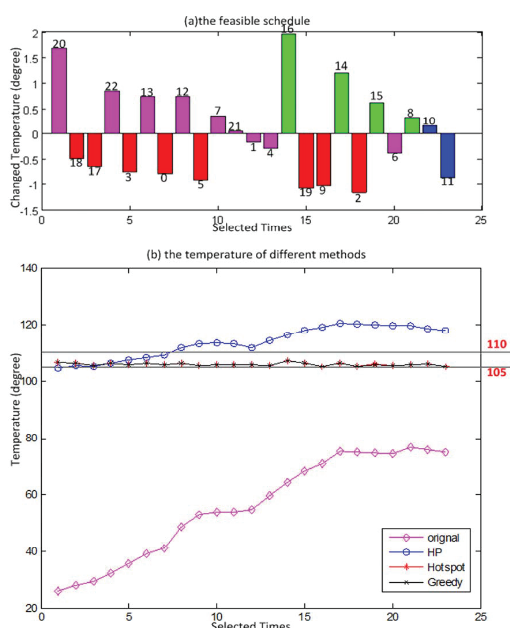


Fig. 7. Experimental Results of Temperature-Aware SBST, while in (a) the red, green, blue, and pink bar stands for TF0, TF1, TF2, and TF3, and the number is the template number; in (b) the pink, blue, red, and black line stands for the temperature trace of the original program, power-intensive program, feasible schedule on hotspot and on the algorithm, respectively.

The limitation of the proposed algorithm is that it may not obtain the most optimized schedule that requires the lowest overhead. The greedy algorithm can find out the best solution only when the previous scheduling step does not affect the latter one. However, *Te* is based on *Ts*, so that the current scheduling result relies on the previous one, and the greedy algorithm may fail to obtain the best schedule.

However, the algorithm is very efficient. On one hand, the algorithm only takes 8.484s to obtain a feasible schedule for generating the temperature-aware test for the miniMIPS ALU, even though the RK4 function is very time-consuming (0.01s on average). This is because the algorithm is based on two simple rules. Furthermore, the rule "high temperature first" provides a larger temperature margin for, and simplifies, the next scheduling step, thus speeding up also the whole scheduling process. On the other hand, the algorithm has optimized the storage space of the schedule greatly. Due to the rule "low overhead first", the algorithm mostly chooses the templates of TF0 and TF3, as shown in Fig. 7 (a). In this way, it avoids to bring in large overhead of storage space. As the result, the algorithm brings 60.6% extra storage space in total for this example, while the power-intensive only SBST solution would requires 118%storage overhead.

VII. CONCLUSIONS

In this paper, a temperature-aware SBST technique is proposed for testing delay faults under high temperature by self-heating the processor within the given temperature range. First, it automatically generates SBST program templates by ATIG, and avoids the serious over-testing program associated with traditional manufacturing test. Second, we propose some transformations of the SBST program templates to enhance their power intensity, which makes it possible to self-heat up the processor internally by just executing programs. Third, the technique applies a greedy algorithm to search the optimized schedule of the SBST templates, while maintaining the high temperature with reduced overhead. Experimental results on the ALU of the miniMIPS processor show the generated SBST test program is successful to guarantee delay test within the given temperature range, and achieves very good test performance with functional patterns.

REFERENCES

[1] N. Aghaee, Z. Peng, P. Eles, "An Efficient Temperature-Gradient Based Burn-in Technique for 3D Stacked ICs", *DATE'2014*, pp.1-4.

[2] A. Krstic, et al., "Embedded Software-Based Self-Test for Programable Core-Based Designs", IEEE Design & Test of Computers, Vol.19, 2002, pp.18-27.

[3] C. P. Ravikumar, M. Hirech, X. Wen, "Test Strategies for Low-Power Devices", Jounal of Low Power Electronics, Vol. 4, 2008, pp.127-138.

[4] X. Liu, Q. Xu, "On X-Variable Filling and Flipping for Capture-Power Reduction in Linear Decompressor-Based Test Compression Environment", IEEE. Trans. on CAD, Vol.11, pp.1743-1753.

[5] D. Gizopoulos, et al., "Systematic Software-Based Self-Test for Pipelined Processors", IEEE Trans on VLSI Systems, Vol. 16, 2008, pp.1441-1452.

[6] V. Singh, M. Inoue, K.K. Saluja, H. Fujiwara, "Instruction-Based Self-Testing of Delay Faults in Pipeline Processors", IEEE Trans. on VLSI Systems, Vol.11, 2006, pp.1203-1215.

[7] K. Christou, et al., "A Noval SBST Generation Technique for Path-Delay Faults in Microprocessors Exploiting Gate- and RT- Level Description", *VTS'2008*, pp.389-394.

[8] Y. Zhang, H. Li, X. Li, "Automatic Test Program Generation using Executing-Trace Based Constraint Extraction for Embedded Processors", IEEE Trans. on VLSI Systems, Vol.21, 2013, pp.1220-1233.

[9] Y. Zhang, A. Rezine, P. Eles, Z. Peng, "Automatic Test Program Generation for Out-of-Order Superscalar Processors", *ATS'2012*, pp.338-343.

[10] A. Merentitis, N. Kranitis, A. M. Paschalis, D. Gizopoulos, "Low Energy Online Self-Test of Embedded Processsors in Dependable WSN Nodes", IEEE Trans. on DSC, Vol.8, 2011, pp.207-217.

[11] M. Psarakis, et al., "Microprocessor Software-Base Self-Testing", IEEE Design & Test of Computers, Vol.27, 2010, pp.4-19.

[12] Z. He, Z. Peng, P. Eles, "A Heuristic for Thermal-Safe SoC Test Scheduling", *ITC'2007*, pp.1-10.

[13] Z. He, Z. Peng, P. Eles, "Multi-Temperature Testing for Core-based System-on-Chip", *DATE'2010*, pp. 208-213.

[14] http://lava.cs.virginia.edu/HotSpot.