Minimizing Concurrent Test Time in SoC's by Balancing Resource Usage

Dan Zhao
CSE Department
SUNY at Buffalo
Buffalo, NY 14260-2000
danzhao@cse.buffalo.edu

Shambhu Upadhyaya
CSE Department
SUNY at Buffalo
Buffalo, NY 14260-2000
shambhu@cse.buffalo.edu

Martin Margala
ECE Department
University of Rochester
Rochester, NY 14627-0231
margala@ece.rochester.edu

ABSTRACT

We present a novel test scheduling algorithm for embedded core-based SoC's. Given a system integrated with a set of cores and a set of test resources, we select a test for each core from a set of alternative test sets, and schedule it in a way that evenly balances the resource usage, and ultimately reduce the test application time. Furthermore, we propose a novel approach that groups the cores and assigns higher priority to those with smaller number of alternate test sets. In addition, we also extend the algorithm to allow multiple test sets selection from a set of alternatives to facilitate testing for various fault models.

Categories and Subject Descriptors

B.7.1 [**Hardware**]: Types and Design Styles; B.8.1 [**Hardware**]: Reliability, Testing, and Fault-Tolerance

General Terms

Algorithm, Management and Reliability

Keywords

Resource balancing, system-on-a-chip test scheduling, test sets selection

1. INTRODUCTION

The system level integration is evolving as a new style of system design, where an entire system is built on a single chip using pre-designed, pre-verified complex logic blocks called embedded cores, which leverage the system by the intellectual property (IP) advantage. More specifically, the system designers (or integrators) may use the cores which cover a wide range of functions (e.g., from CPU to SRAM to DSP to analog, etc.), and integrate them into a system on a single chip (SoC) with their own user-defined-logics (UDLs). The SoC technology has shown great advantage in shortening the time-to-market of a new system and meeting various requirements (such as the performance, size and cost) of today's electronic products.

GLSVLSI'02, April 18-19, 2002, New York, New York, USA.

However, testing such core-based SoC's poses a major challenge for the system integrators, as they may have limited knowledge of the cores due to the so called IP protection, and on the other hand, various testing methods (e.g., BIST, scan, functional, structure, etc.) for many kinds of design environments are provided by different core vendors. Further, future SoCs will see several hundreds of embedded components in a single package [1]. In order to select an efficient test strategy for a SoC, several performance criteria listed below need to be considered.

1) *overall test time*. The overall test time of a testing scheme is defined as the period from the start time of the test activity to the end time when the last test task finishes. Note that, only when all test sets in parallel test queues finish their tasks, we say it's the end time of the test. In other words, the longest test queue dominates the overall test time. In addition, since the expensive testers are shared by many cores, the shorter the test time, the lower the cost is. The test time may be reduced by using shorter test vectors or better scheduling schemes.

2) *fault coverage*. In order to gain a high fault coverage, the individual embedded cores and UDLs should be tested thoroughly, which is also referred as core-level testing. This includes consideration of various fault models. In addition, the interconnections between different system blocks also need to be tested. Finally the system level testing should be processed to check the system functions.

3) *area overhead.* The area overhead is the extra silicon area needed in order to perform the SoC test. The area overhead should be limited within a certain area budget, and kept as small as possible.

4) *performance overhead.* As one undesirable side-effect of integrating test resources into the system, the power consumption of the SoC may increase while its speed may decrease. This performance overhead may vary when using different testing methods, and thus becomes a major performance criterion when evaluating various test strategies.

In this paper, we address the test scheduling problem for embedded core-based SoC's. We consider a system where one test set needs to be selected for each core from a group of test sets using various test resources, and propose a novel test scheduling scheme to reduce the overall test time. The basic idea of the proposed scheduling algorithm is to efficiently balance the resource usage.

The rest of this paper is organized as follows. In Sec. 2, we discuss the conflicts and constraints for the scheduling problem and the existing scheduling schemes. Sec. 3 describes a general SoC model, in which each core may have multiple test sets using different resources. In Sec. 4, we propose a novel scheduling algorithm

based on the effective balancing of resource usage. Sec. 5 extends the algorithm by selecting multiple test sets for each core. Finally, Sec. 6 concludes the paper and presents the future works.

2. RELATED WORK

The objective of test scheduling is to decide the start and end times for the test of each core in order to meet all of the constraints and avoid any conflicts of test application. The basic idea is to schedule the tests in parallel so that those nonconflicting tests can be executed concurrently, and thus the total testing time may be reduced.

There are several constraints that must be considered in scheduling of tests. First, in a core-based SoC, not all tests can be applied at the same time due to the resource conflicts. For example, several cores may share the same test generator or response evaluator, and thus cannot be tested in parallel. In addition, the power consumption must be taken into account in order to guarantee proper operating conditions. For instance, in a self-tested system, testing the cores in parallel may result in high power consumption and exceed the maximum power limit, which will result in system damage due to overheating, while the cores may not activate simultaneously in normal functional mode. Finally, certain fault coverage should be achieved when testing a SoC. There are usually a number of core-testing methods available and each of them detects different faults (e.g., BIST for detecting performance-related defects and nonmodeled faults, while external test for detecting modeled faults). One method or a combination of several methods may be needed to test a core in order to gain the required fault coverage.

Various approaches have been proposed for test scheduling. Chakrabarty has shown in [2] that the problem of test scheduling for SoC's is equivalent to the open shop scheduling, and thereby is NP-complete when the number of resources is larger than 2. A *"shortest-task-first"* algorithm has been proposed for scheduling large systems. However, it assumes that the cores have to use all of the pre-determined test sets and the corresponding resources. In other words, the scheduling algorithm only arranges the sequence of the test sets, which is different from (or actually a special case of) our proposed algorithm (to be discussed later in Sec. 4). Our algorithm selects the test sets from the candidates and determines the test schedule.

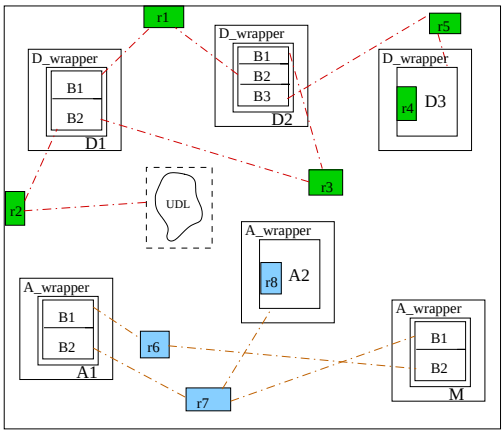
In [3], Larsson et. al. proposed a test parallelization (scan-chain subdivision) combining scheduling scheme to minimize test time under power limitation. However, their technique is based on a greedy algorithm which is not efficient for large SoC's with alternative test sets for each core.

The readers may also refer to [4, 5, 6, 7] for other scheduling algorithms.

3. SYSTEM MODELING

In this section, we describe a general SoC model, which includes not only digital cores (denoted as D_cores), but analog cores (denoted as A_cores), mixed-signal cores (denoted as M_cores) and UDLs as well (see Figure 1). Each core may have multiple test sets using different resources, and thus provides flexibility for test scheduling.

In SoC test, each core is surrounded by wrappers, which are used to isolate the cores and transport data under different modes (e.g., normal/core test/interconnect test mode). The Test Access Mechanism (TAM) serves as "test data highway", which transports test data and control signals and executes system chip test in a predetermined schedule. In addition, large UDLs can be treated as D_cores or A_cores according to their functionality. If A_cores and



r1 - r8 are different testing resources
D1 - D3 are digital cores
A1 - A2 are analog cores
M is a mixed-signal core
B_i is denoted as different functional blocks in a core

Figure 1: A General SoC Model.

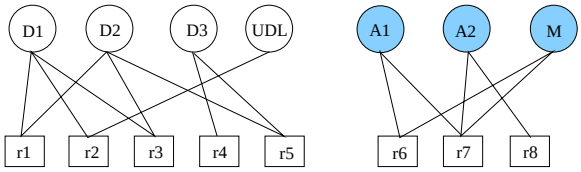


Figure 2: Graph Representation of Resource Sharing.

D_cores do not share resources, they can be grouped into two separate sets and tested in parallel using the same scheduling technique, as shown in Figure 2. In this paper, we do not take into consideration different core types and focus on D_cores only.

Without loss of generality, we assume that a SoC includes n cores, and there are m resources available for testing. A core may need one or several tests to meet the required fault coverage. Each test set includes a set of test vectors and needs one resource, which can be used by one core at one time. There are different test sets to achieve the same fault coverage but with different test time by using different resources. In other words, a core vendor may have provided a set of alternative tests, and one test from each group needs to be performed to achieve the required fault coverage. A collision occurs when the tests sharing the same resource or the tests for the same core are performed in parallel. In addition, the total power consumption must not exceed the maximum power allowance at any time[1].

Given the test time and the required fault coverage, the goal of the scheduling technique is to efficiently determine the start time of the test sets to minimize the total test application time.

More formally, we define the SoC model as $TM = \{C, RSC, T, FC\}$, in which $C = \{c_1, c_2, ...c_n\}$ is a finite set of cores, $RSC = \{r_1, r_2, ...r_m\}$ is a finite set of resources, FC is the fault coverage required to test each core, and $T = \{T_{11}, T_{12}, ..., T_{1m}, ..., T_{n1}, T_{n2}, ..., T_{nm}\}$ is a finite set of tests, which are shown in a $n \times m$ matrix in Figure 3. In the matrix, each test set, defined as $T_{ij} = \{t_{ij}\}$ (where t_{ij} is the test time), consists of a set of test vectors. Test set T_{ij}

[1]Although we will not consider the power consumption limit in our scheduling algorithm, we would like to address this problem in our future work.

T_{ij}	r1	r2	r3	· · · · ·	rm
c1	T11	T12	0	· · · · ·	0
c2	0	T22	0	· · · · ·	T2m
c3	T31	0	0	· · · · ·	0
c4	0	0	T43	· · · · ·	0
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
cn	Tn1	Tn2	Tn3	· · · · ·	Tnm

Each vector T_{ij} in the matrix represents a test set for testing core c_i by using resource r_j.

Figure 3: Matrix Representation of Test Sets.

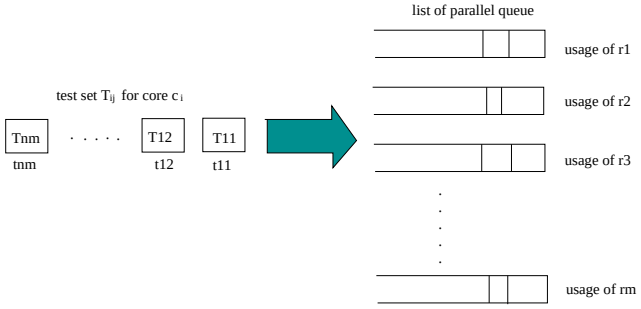


Figure 4: Parallel Usage of Test Resources.

represents a test set for testing core c_i by using resource r_j. The entries with zero indicate that such test sets are not available.

4. A NOVEL TEST SCHEDULING ALGORITHM: RESOURCE BALANCING

As the test scheduling problem is NP-complete, many heuristic algorithms have been proposed. However, as we have discussed in Sec. 2, most of them assume that all of the given test sets have to be used in testing. In this work, we propose an efficient heuristic scheduling algorithm for the case where one of a group of test sets may be selected for each core to perform testing, and take into consideration the test conflicts and the fault coverage requirements. A generalization of this problem is given in Sec. 5.

We consider a system discussed in Sec. 3 and define m queues in parallel corresponding to m resources that may be used at the same time independently (see Figure 4). The length of the queue denotes the total testing time of all test sets using the resource. We assume that there are P_i ($1 \leq P_i \leq m$) test sets, which use different resources, available for testing core c_i, and one core needs only one of these test sets to achieve the required fault coverage (However, this limit will be nullified later for the multiple test sets case.). In order to schedule the test sets for such a system, we propose a resource-balancing heuristic algorithm.

The basic idea of this algorithm is to use the resources as evenly as possible, because the total testing time is dominated by the longest usage time among all resources used in a test. When scheduling core c_i, we temporarily insert all of its test sets into the corresponding queues. Then we choose the one resulting in shortest queue length, and delete other test sets of c_i from the other queues.

```
Read the data into the structure CORE, including the following members,
    CORE :  i        /* core id */
            P        /* the number of test sets */
            t[P]     /* the test time of each test set */
            r[P]     /* the corresponding test resource */

/* Balancing resource usage queues without grouping*/
begin   /* initialize the resource usage queues, rqs[i] is the queue of r i*/
for i := 1 to m do                          /* m is the number of resources */
    rqs[i] = ∅;
for i := 1 to n do                          /* n is the number of cores */
    short_length=CORE[i].t[1]+rqs[CORE[i].r[1]]_length;
    short_length_id = 1;
    for j := 2 to CORE[i].P do              /* for all test sets tij of ci */
        if (CORE[i].t[j] + rqs[CORE[i].r[j]]_length < short_length) then
            begin                           /* get the shortest queue length */
             short_length = CORE[i].t[j] + rqs[CORE[i].r[j]]_length;
             short_length_id =j;
            end
    Insert test(short_length_id) into rqs[CORE[i].r[short_length_id]];
    rqs[j]_length = short_length;           /* update the queue length */
end
```

Figure 5: The Scheduling Algo. Without Grouping.

For example, we execute the algorithm (refer to the pseudocode in Figure 5) for a core-based system with 6 cores and 4 resources as shown in Table 1, which shows the test time for each test set using a resource for each core. We first sequentially select the shortest test among their alternatives for cores c_0, c_1, c_2 and insert them into r_3, r_2, r_0 respectively. Note that, a shorter test set will not always be selected as it may use the resource corresponding to a longer queue. For example, when we schedule c_3, we may not select the one (T_{33}) with shortest test time (9) since it results in a longer queue length (for r_3). Instead, we select T_{32} and insert it into queue r_2. Similarly we select T_{41} for c_4, T_{50} for c_5, and finally T_{60} for c_6. The final schedule is shown in Figure 6(1), which results in a total test time of 14. It can be shown that, the worst-case time complexity of this algorithm is $O(T)$, where T is the number of the test sets. Moreover, we can see that one of the advantages of resource balancing approach is that there's no idle time between successive tests (namely, explicit dead time) in any of the queues.

PROPOSITION 1. *There is no "explicit dead time" in this resource balancing approach.*

PROOF. Explicit dead time arises due to resource conflicts. There are two types of resource conflicts defined in our system. 1) Several tests compete to use the same resource. This kind of conflict is totally overcome by resource balancing approach, since the tests competing for a resource sequentially enter the resource queue. 2) Different tests for the same core are executed at the same time. Although for each core a set of tests are provided, only one of them will be executed to test the core. So this type of conflict is eliminated. □

As there's no explicit dead time in resource balancing, our purpose is to efficiently and effectively reduce the implicit dead time at the end of the resource queues (Obviously, no implicit dead time appears at the beginning in this approach.). We find out that different ordering of ready-to-schedule cores (i.e., the cores before entering the resource queues), results in different schedule of tests, and accordingly the test time. More specifically, the ready-to-schedule cores are grouped based on the value of P_i such that in a group G_p, all cores have P alternate test sets, and the cores in the group with lower P value will be scheduled earlier, because these tests have

Table 1: The Matrix of Test Sets for An Example System

T_{ij}	r_0	r_1	r_2	r_3	P
c_0	12	7	0	6	3
c_1	0	4	1	0	2
c_2	3	0	8	12	3
c_3	0	0	13	9	2
c_4	5	7	6	2	4
c_5	5	0	1	0	2
c_6	4	6	0	0	2

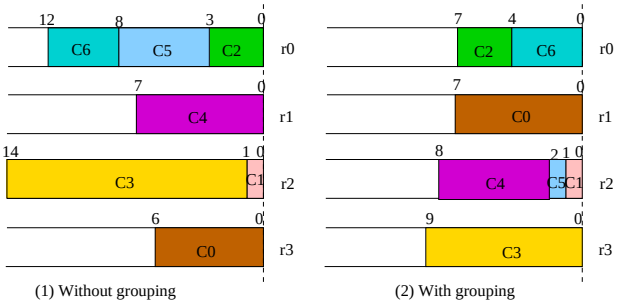


Figure 6: The Resource Balancing Approach.

to be put into certain queues (i.e., the corresponding cores have to be tested by using certain resources.). Next, we may choose proper test sets from the group with larger P value to balance the lengths of the queues. The pseudocode of this algorithm is shown in Figure 7.

```
Read the data into the structure CORE, including the following members,
    CORE :   i        /* core id */
             P        /* the number of test sets */
             t[P]     /* the test time of each test set */
             r[P]     /* the corresponding test resource */
/* Balancing resource usage queues with grouping */
begin  /* initialize the resource usage queues, rqs[i] is the queue of r i*/
for i := 1 to m do              /* m is the number of resources */
   rqs[i] = ∅ ;
for i := 1 to n do              /* n is the number of cores */
   CORE[i] --> G[CORE[i].P];    /* group the cores */
for each group
   for each core i in the group
      short_length=CORE[i].t[1]+rqs[CORE[i].r[1]]_length;
      short_length_id = 1;
      for j := 2 to CORE[i].P do
         if (CORE[i].t[j] + rqs[CORE[i].r[j]]_length < short_length) then
         begin
            short_length = CORE[i].t[j] + rqs[CORE[i].r[j]]_length;
            short_length_id =j;
         end
      Insert test(short_length_id) into rqs[CORE[i].r[short_length_id]];
      rqs[j]_length = short_length;
end
```

Figure 7: The Scheduling Algo. With Grouping.

Figure 6(2) illustrates the execution of the algorithm with grouping (refer to Figure 7) for the example system. By grouping, c_1, c_3, c_5 and c_6 are in the same group and are inserted into queue r_2, r_3, r_2 and r_0 respectively. Then we schedule c_0 and c_2. Note that, we will select the test with test time of 7 for core c_0 and insert it into r_1, although there are other test sets with shorter test times. Similarly we insert the test set with time of 3 for c_2 into r_0, and finally insert the test set with time of 6 for c_4 into r_2. The total test time is 9. Compared to the case (Figure 6(1)) where we don't use groping, the total test time is reduced by 5. As we can see, grouping the cores before scheduling can significantly reduce the total testing time and achieve better balancing of resource usage, while the worst case time complexity remains the same. Comparing the schedule with and without grouping (see Figures 6, 8 and 9), we get the following conclusion.

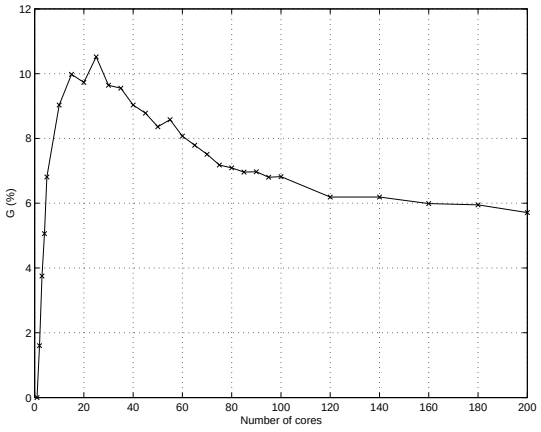


Figure 8: G Changing With the Number of Cores.

Result 1. Grouping always helps balance the resource usage queue lengths.

4.1 Simulation Study

We evaluate the proposed scheduling algorithms via simulations. In our simulation model, we use randomly generated test sets. We define the balance ratio as G, which can be expressed by the following equation:

$$G = \frac{L_{wg} - L_{wog}}{L_{wg}}$$

where L_{wg} is the total test time of a schedule with grouping while L_{wog} is the total test time of a schedule without grouping.

In simulation scenario 1, we study the effect of the number of cores on the test time, and compare the performance of the approach with grouping with that without grouping. We assume that there are 5 resources in the system and each core may be provided with 1 to 3 test sets using corresponding resources to meet the fault coverage requirement. The results are shown in Figure 8. As we can see, G is always larger than zero. In other words, grouping always helps reduce the total test time. In addition, G increases sharply when the number of cores increases. It reaches a peak of 10.52% when the number of cores is 25. Then it drops slowly when the number of cores increases further. This is reasonable because, when there are small number of cores, the total number of tests is also small and we couldn't balance the resource queues more evenly due to less flexibility. As the number of cores increases, the flexibility increases, and accordingly, G increases. On the other hand, when there are a large number of tests, the benefit of grouping will be dominated by the randomness, which in turn results in the dropping of the curve.

In scenario 2, we set the number of cores to 25 (where the peak is settled in simulation 1), and change the total number of resources in the system from 3 to 6. Figure 9 shows how G changes when the maximum number of resources provided for each core changes. As we can see, with the same total number of resources, G increases almost linearly with the maximum number of resources for each core,

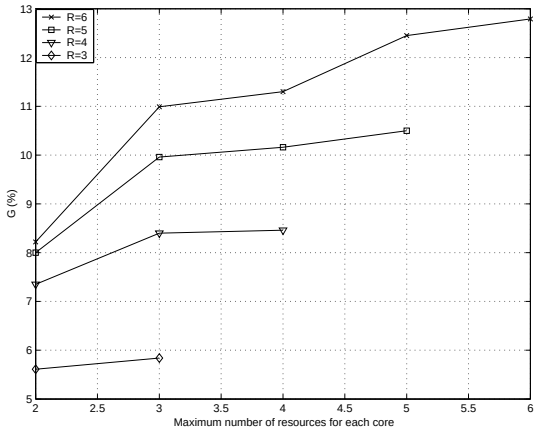


Figure 9: G Changing With the Max. Number of Resources for Each Core.

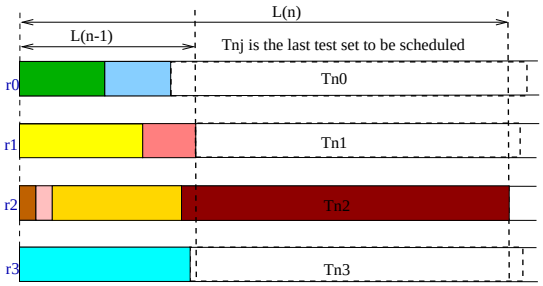


Figure 10: The Worst Case Scheduling.

while with the same maximum number of resources for each core, G increases when the total number of resources increases. This is again due to the change in flexibility of choosing test resources as described in scenario 1.

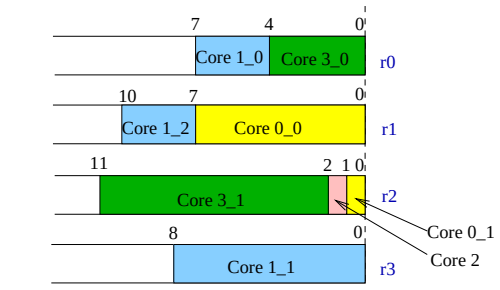
The worst case schedule occurs when the last test to be scheduled comes from a group of long tests (i.e., $t_{nj} \gg L(n-1)$, where $L(n-1)$ is defined as the total test time after $n-1$ cores have been scheduled, shown in Figure 10). In this case, t_{nj} dominates the total test time. We can reduce the test time in the worst case by rescheduling, in which t_{nj} is scheduled first.

5. MULTIPLE TEST SETS SELECTION & SCHEDULING

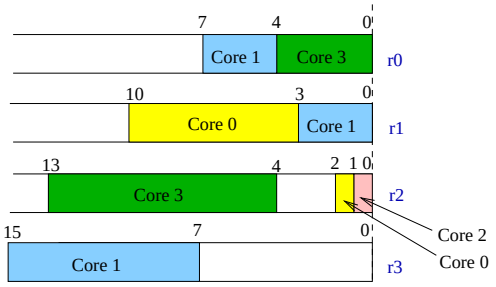
In the last section, we assumed that one core needs only one test set. More generally, a core may need multiple (say L) test sets to achieve a certain fault coverage. For example, in an embedded-core-based SoC, several test methods are used to test embedded memory. As we know, in addition to stuck-at, bridge, and open faults, memory faults include bit-pattern, transition, and cell-coupling faults. Parametric, timing faults, and sometimes, transistor stuck-on/off faults, address decoder faults, and sense-amp faults are also considered. [8] lists various test methods for embedded memory, i.e., Direct access, Local boundary scan or wrapper, BIST, ASIC functional test, Through on-chip microprocessor, etc. Different test methods may require different test resources, use different test time, and provide different fault coverages. In this case, we can simply make L virtual cores and convert the 1-L mapping to a 1-1 mapping. The only difference between this and the single test selection we discussed earlier is that, when choosing the shortest queue, one has to check if the selected test set conflicts with others which are for the same core and overlap the running time. Figure 12 illustrates the multiple test sets scheduling for a system shown in Figure 11, which can be performed in two steps. In Figure 11, the tests are to be performed using the corresponding resources, for instance test t_{10} to be applied using resource r_0, test t_{11} using resource r_2 etc. and in Figure 12, *Core i_j* for various values of j indicate the various instances of Core c_i. The pseudocode of this algorithm is given in Figure 13.

Core ID	Fault model	Test set selection group	Corresponding resource
c0	f00	t00 = 12 t01 = 7 t02 = 6	r0 r1 r3
	f01	t03 = 4 t04 = 1	r1 r2
c1	f10	t10 = 3 t11 = 8 t12 = 12	r0 r2 r3
	f11	t13 = 13 t14 = 8	r2 r3
	f12	t15 = 5 t16 = 3 t17 = 6 t18 = 11	r0 r1 r2 r3
c2	f20	t20 = 5 t21 = 1	r0 r2
c3	f30	t30 = 4 t31 = 6	r0 r1
	f31	t32 = 18 t33 = 11 t34 = 9	r1 r3 r2

Figure 11: A Fault Model Based System.



(a) test set selection for each fault model of cores



(b) rescheduling to avoid conflicting

Figure 12: Multiple Test Sets Scheduling.

First, we create L virtual cores for each core corresponding to L fault models. For each fault model, a group of test sets with various test time is provided for the required fault coverage. This means, each virtual core has a group of test sets available and we select one of them to perform testing. Thus we map the multiple

tests selection model to the single test selection case. We select the tests in a way that we balance the queues in order to avoid the situation where all the test sets will only use some of the resources and thus result in long length in these queues. In the second step, we need to reschedule the tests for the same core which overlap the running time. The shortest-task-first procedure is adopted here for rescheduling. The worst case complexity is $O(r^3)$, where r is the number of the virtual cores.

```
Read the data into the structure V_CORE, including the following members,
        V_CORE :  i        /* virture core id */
                  id       /* real core id */
                  P        /* the number of test sets */
                  t[P]     /* the test time of each test set */
                  r[P]     /* the corresponding test resource */
begin
/* Step 1 : balancing resource usage queues with grouping for virture cores*/
for i := 1 to m do              /* m is the number of resources */
   tmp_rqs[i] = ∅;              /* initialize, tmp_rqs[i] is the temp queue of r i*/
for i := 1 to N do              /* N is the number of virture cores */
   V_CORE[i] --> G[V_CORE[i].P];          /* group the virture cores */
for each group
   for each virture core i in the group
      short_length=V_CORE[i].t[1]+tmp_rqs[V_CORE[i].r[1]]_length;
      short_length_id = 1;
      for j := 2 to V_CORE[i].P do
          if (V_CORE[i].t[j] + tmp_rqs[V_CORE[i].r[j]]_length < short_length) then
          begin
             short_length = V_CORE[i].t[j] + tmp_rqs[V_CORE[i].r[j]]_length;
             short_length_id =j;
          end if
      Insert test(short_length_id) into tmp_rqs[V_CORE[i].r[short_length_id]];
      tmp_rqs[j]_length = short_length;
      tmp_rqs[j]_Numtest ++;
/* Step 2 : rescheduling tests by shortest-task-first */
for i := 1 to m do                  /* initialize structure SJF[l] */
   for j := 1 to tmp_rqs[i]_Numtest do
      for k := 1 to V_CORE[tmp_rqs[i]_Numtest[j]].P do
          if (V_CORE[tmp_rqs[i]_Numtest[j]].r[k]==i) then
          begin
             SJF[l].vid=V_CORE[tmp_rqs[i]_Numtest[j]].i;
             SJF[l].coreid=V_CORE[tmp_rqs[i]_Numtest[j]].id;
             SJF[l].starttime=0;
             SJF[l].stoptime=V_CORE[tmp_rqs[i]_Numtest[j]].t[k];
             SJF[l].testtime=V_CORE[tmp_rqs[i]_Numtest[j]].t[k];
             SJF[l].res=V_CORE[tmp_rqs[i]_Numtest[j]].r[k];
             l ++;
          end if
while(FLAG==1)                      /* rescheduling */
begin
   FLAG=0;
    for i := 1 to N do
        for j := i+1 to N do
            /* check conflicting tests */
            if ((SJF[i].coreid==SJF[j].coreid)      /* if for the same core */
               ||(SJF[i].res==SJF[j].res)) then     /* if share the same resource */
                /* check overlapping tests */
                if (((SJF[i].stoptime>SJF[j].starttime)&&(SJF[i].stoptime<=SJF[j].stoptime))
                  ||((SJF[j].stoptime>SJF[i].starttime)&&(SJF[j].stoptime<=SJF[i].stoptime))) then
                begin           /* test j complete first */
                   if (SJF[i].stoptime>=SJF[j].stoptime) then
                         SJF[i].starttime=SJF[j].stoptime;
                         SJF[i].stoptime=SJF[i].starttime+SJF[i].testtime;
                   else         /* test i complete first */
                         SJF[j].starttime=SJF[i].stoptime;
                         SJF[j].stoptime=SJF[j].starttime+SJF[j].testtime;
                    FLAG=1;
                end if
end while
end
```

Figure 13: The Multiple Test Sets Scheduling Algo.

6. CONCLUSION AND FUTURE WORK

In this paper, we have presented an efficient test scheduling algorithm for embedded core-based SoC's. With the flexibility of selecting a test set from a set of alternatives, we have proposed to schedule the tests for a given system in a way that balances the resource usage queue as evenly as possible, thus reducing the overall test time. Furthermore, we have presented a grouping scheme to optimize the schedule and evaluated the approaches via simulation. Our simulations showed that there is no explicit dead time in our approach and we can further reduce the implicit dead time by proper grouping. We have also extended the algorithm to allow multiple test sets selection from a set of fault model based alternatives.

Our initial results lead to further study in the following research directions.

- Development of efficient test scheduling algorithm to reflect the various constraints, not only resource sharing and fault coverage, but also power dissipation.
- Experiments with benchmarks for performance verification of the proposed scheduling algorithms.
- Extension of our work to mixed-signal SoC's. We will discuss the modeling of mixed-signal SOC for developing testability analysis, scheduling and diagnosis and present efficient test scheduling algorithms to minimize the test cost.

7. REFERENCES

[1] A. Allan, D. Edenfeld, J. William H. Joyner, A. B. Kahng, M. Rodgers, and Y. Zorian, "2001 technology roadmap for semiconductors," *IEEE Computer*, vol. 35, pp. 42–53, January 2002.

[2] K. Chakrabarty, "Test scheduling for core-based systems using mixed-integer linear programming," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, pp. 1163–1174, October 2000.

[3] E. Larsson and Z. Peng, "System-on-chip test parallelization under power constraints," in *Proc. of IEEE European Test Workshop*, May 2001.

[4] M. Sugihara, H. Data, and H. Yasuura, "Analysis and minimization of test time in a combined BIST and external test approach," in *Design, Automation and Test in Europe Conference 2000*, pp. 134 – 140, March 2000.

[5] R. Chou, K. Saluja, and V. Agrawal, "Scheduling tests for VLSI systems under power constraints," *IEEE Trans. on VLSI Systems*, vol. 5, pp. 175–185, June 1997.

[6] V. Muresan, X. Wang, V. Muresan, and M. Vladutiu, "A comparison of classical scheduling approaches in power-constrained block-test scheduling," in *Proceedings IEEE International Test Conference 2000*, pp. 882–891, October 2000.

[7] Y. Zorian, "A distributed BIST control scheme for complex VLSI devices," in *Proceedings IEEE VLSI Test Symposium (VTS)*, pp. 4–9, April 1993.

[8] R. Rajsuman, "Design and test of large embedded memories: An overview," *IEEE Design and Test of Computers*, vol. 18, pp. 16–27, May-June 2001.