A Novel Test Time Reduction Algorithm for Test Architecture Design for Core-Based System Chips

Sandeep Kumar Goel  Erik Jan Marinissen
Philips Research Laboratories
IC Design – Digital Design & Test
Prof. Holstlaan 4, M/S WAY-41
5656 AA Eindhoven, The Netherlands

Abstract
This paper deals with the design of SOC test architectures which are efficient with respect to required ATE vector memory depth and test application time. We advocate the usage of a TestRail Architecture, as this architecture, unlike others, allows not only for efficient core-internal testing, but also for efficient testing of the circuitry external to the cores. We present a novel heuristic algorithm that effectively optimizes the TestRail Architecture for a given SOC by efficiently determining the number of TestRails and their widths, the assignment of cores to the TestRails, and the wrapper design per core. Experimental results for four benchmark SOCs show that, compared to previously published algorithms, we obtain comparable or better test times at negligible compute time.

1 Introduction
In order to reduce development time, designers increasingly embed pre-designed and pre-verified modules in their system chip (SOC) designs. These reusable modules are known as cores. Test development for such large, core-based SOCs poses major challenges [1]; test access is one of them. Typically, a core is deeply embedded in the SOC and due to its surrounding circuitry, direct access from the SOC pins to the core terminals is not possible. In order to be able to test an embedded core as a stand-alone unit, it should be isolated from its surrounding circuitry and electrical test access needs to be provided.

Zorian et al. [2] introduced a generic conceptual test access architecture for SOCs consisting of three elements: (1) test pattern source and sink, (2) a test access mechanism (TAM), and (3) a core test wrapper. The core test wrapper can isolate the core from its surroundings and provides switching functionality between regular functional access to the core and test access through the TAM. To design a test architecture for a given set of cores and a given number of test pins, an SOC integrator has to determine (1) the TAM type, (2) the number of TAMs, (3) the widths of these TAMs, (4) the assignment of cores to TAMs, and (5) the wrapper design. The test architecture has a large impact both on the required vector memory depth per ATE channel as well as on the test application time of the SOC; two key parameters in the overall SOC test costs. In the remainder of this paper, we loosely refer to these two parameters as “test time”.

In this paper, we address the problem of designing an efficient test architecture for a given SOC with respect to its test time. Our solution is built on the TestRail Architecture, which, compared to other known test architectures, allows not only for efficient core-internal testing, but also for efficient testing of the circuitry outside the cores. We present an effective and efficient heuristic algorithm that computes, for a given SOC, a TestRail Architecture instance optimized with respect to required ATE vector memory depth and test application time. For four benchmark SOCs [3], we show that the proposed algorithm yields good test application times in negligible compute time.

The sequel of this paper is organized as follows. Section 2 reviews prior work in this domain and compares the TestRail Architecture to other test architectures described in literature. Section 3 presents our novel algorithm, named TR-ARCHITECT, for generating an efficient TestRail Architecture instance for a given SOC. Section 4 contains experimental results for four benchmark SOCs and compares those with results obtained by other methods. Section 5 concludes this paper.

2 Prior Work
Various test architectures have been described in literature. Aerts and Marinissen [4] described the three scan-based test architectures depicted in Figure 1: (a) the Multiplexing Architecture, (b) the Daisy-chain Architecture, and (c) the Distribution Architecture.

In the Multiplexing and Daisy-chain Architectures, all cores get access to the full available TAM width. In the Multiplexing Architecture (a), only one core can be accessed at a time. This implies that the total test time is the sum of the individual core test times, but, more importantly, also that core-external testing (i.e., testing the circuitry and wiring in between the cores) is cumbersome or even impossible. This is due to the fact that only one core wrapper can be accessed at a time, while for core-external testing the wrappers of two or more cores need to be accessed simultaneously. Due to its bypass mechanism, the Daisy-chain Architecture (b) does not have this restriction. One of way to use this architecture, is to test all cores in parallel. As soon as a core runs out of test patterns, its bypass is turned on, while the testing of the remaining cores continues. In the Distribution Architecture (c), the total available TAM width is distributed over the cores. This allows cores to be tested concurrently, and hence the total SOC test time is the maximum of the individual core test times. In order to minimize the SOC test time, the width of an individual TAM should be proportional to the amount of test data that needs to be transported to/from a core.
The **Test Bus Architecture** presented by Varma and Bhatia [5] is a combination of the Multiplexing and Distribution Architectures. A single Test Bus is in essence the same as what is described by the Multiplexing Architecture; cores connected to the same Test Bus can only be tested sequentially. The Test Bus Architecture allows for multiple Test Buses on one SOC, which operate independently, as in the Distribution Architecture. Cores connected to a common Test Bus suffer from the same drawback as in the Multiplexing Architecture, viz. core-external testing is difficult or impossible.

The **TestRail Architecture** presented by Marinissen et al. [6] is a combination of the Daisychain and Distribution Architectures. A single TestRail is in essence the same as what is described by the Daisychain Architecture: scan-testable cores connected to the same Test Rail can be tested simultaneously, as well as sequentially. A TestRail Architecture allows for multiple TestRails on one SOC, which operate independently, as in the Distribution Architecture. The advantage of the TestRail Architecture over the Test Bus Architecture is that it allows access to multiple or all wrappers simultaneously, which facilitates core-external testing.

Most SOC test architecture optimization algorithms published so far have concentrated on the Test Bus Architecture and assume cores with fixed-length scan chains. Chakrabarty described a TAM design approach that minimizes test time through Integer Linear Programming (ILP) [7]. Edabi and Ivanov replaced ILP by a genetic algorithm [8]. In [9], Chakrabarty extended the optimization criteria with place-and-route and power constraints, again using ILP as solution approach. In [10], Huang et al. mapped test architecture design to the well-known ILP problem. In [11], Iyengar et al. were the first to formulate the problem of two-dimensional bin packing and used a Best Fit Decreasing algorithm to solve it. Recently, test architectures based on other TAMs than the Test Bus have become the topic of optimization research as well. In [12], the authors presented a novel optimization algorithm for the Test-Rail Architecture. This algorithm optimizes the TestRail Architecture for a given SOC with respect to test time. It uses the following input parameters:

- The total available width for all TestRails \( w_{\text{max}} \).
- The set of cores \( C \) and for every core \( c \in C \):
  - The number of input terminals \( i_c \), output terminals \( o_c \), and bidirectional terminals \( b_c \);
  - The number of core-internal scan chains \( \kappa_c \), and for each scan chain \( k (1 \leq k \leq \kappa_c) \), the length of the scan chain \( l_{c,k} \) in flip flops;
  - The number of test patterns \( \kappa_c \).

A TestRail \( r \) is represented as a set of cores, which are connected to \( r \). Our algorithm determines the TestRail Architecture, which consists of:

- The set of TestRails \( R \), such that \( C = \bigcup_{r \in R} r \) and \( \forall_{r_1, r_2 \in R} (r_1 \cap r_2 = \emptyset) \), i.e., every core is assigned to exactly one TestRail;
- The width \( w(r) \) of every TestRail \( r \in R \), such that \( \sum_{r \in R} w(r) \leq w_{\text{max}} \), i.e., the summed widths of all TestRails does not exceed \( w_{\text{max}} \);
- The wrapper design for each core \( c \in C \);

such that SOC test time \( T \) is minimized.

The total test time \( T \) for a TestRail Architecture is the maximum of the test times of the individual Test Rails. The cores \( c \) connected to a TestRail \( r \) can be tested sequentially as well as concurrently. In case of sequential testing, while a core is being tested, all other cores connected to the same TestRail are bypassed. In case of concurrent testing, all cores \( c \in r \) are tested in parallel, until one of them runs out of test patterns. Without loss of generality, we assume that all cores in \( r \) are sorted in non-decreasing test pattern count order, i.e., \( p_1 \leq p_2 \leq \ldots \leq p_{|r|} \). This order implies that Core 1 runs out of test patterns first. We then turn on the bypass for this core, while the testing of the remaining cores continues. \( p_2 - p_1 \) test patterns later, Core 2 runs out of test patterns; its bypass is also turned on. This process is repeated until all cores have been completely tested.

To determine the test time \( f(r) \) for a TestRail \( r \) with width \( w \), we assume the existence of a procedure \( \text{TESTTIME}(r; w) \). The procedure \( \text{TESTTIME}(r; w) \) uses a procedure \( \text{WRAPPERDESIGN}(c; w) \) for designing a wrapper around a Core \( c \in r \). In case of cores with fixed-length scan chains, our procedure \( \text{WRAPPERDESIGN} \) uses the \( \text{COMBINE} \) algorithm presented in [13]. For cores with flexible-length scan chains, the procedure assumes that all flip-flops in Core \( c \) can be equally distributed over the \( w \) wires.
The procedure OPTIMIZE-BOTTOM UP tries to optimize the test time for a TestRail Architecture, which will be further optimized by the procedures to follow. It consists of a short initialization, followed by three main steps.

In Step 1 (Lines 3–7), we assign cores to one-bit wide TestRails. If \( w_{\text{max}} \geq |C| \), each core gets assigned; if \( w_{\text{max}} < |C| \), only the largest \( w_{\text{max}} \) cores get assigned. ‘Large’ is here defined by the test data volume for each core, according to which the cores have been sorted in Line 1. In case \( w_{\text{max}} = |C| \), the procedure is now finished.

In case \( w_{\text{max}} < |C| \), we still have unassigned cores left. In Step 2 (Lines 8–13), these cores are added iteratively to the one-bit wide TestRail with the shortest test time. This procedure is in fact based on the Largest Processing Time (LPT) algorithm for Multi-Processor Scheduling [17].

In case \( w_{\text{max}} > |C| \), we still have TAM wires left. In Step 3 (Lines 14–19), these wires are added iteratively to the TestRail with the longest test time. This procedure is in fact based on the algorithm for the Scan Chain Distribution Problem (SCDP) [4].

**Algorithm 2 [CREATESTARTSOLUTION]**

1. sort \( C \) such that
   \[ \text{TestTime}(1, 1) \geq \text{TestTime}(2, 1) \geq \ldots \geq \text{TestTime}(|C|, 1); \]
2. \( R := \emptyset; \)
3. /* Step 1: Assign the (largest) cores to a one-bit TestRail each */
4. for \( i := 1 \) to \( w_{\text{max}} \) [\( |C| \)]
5. \( r_i := \{i\}; \)
6. \( R := R \cup \{r_i\}; \)
7. \( w(r_i) := 1; \)
8. \( t(r_i) := \text{TestTime}(r_i, w(r_i)); \)
9. /* Step 2: If there are cores left, add them to the least-occupied TestRails */
10. if \( w_{\text{max}} < |C| \)
11. for \( i := w_{\text{max}} + 1 \) to \( |C| \)
12. find \( r^* \) for which \( t(r^*) = \min_{r \in R} t(r); \)
13. \( r^* := r^* \cup \{i\}; \)
14. \( t(r^*) := \text{TestTime}(r^*, w(r^*)); \)
15. /* Step 3: If there are wires left, add them to the most-occupied TestRails */
16. if \( w_{\text{max}} > |C| \)
17. for \( i := |C| + 1 \) to \( w_{\text{max}} \)
18. find \( r^* \) for which \( t(r^*) = \max_{r \in R} t(r); \)
19. \( w(r^*) := w(r^*) + 1; \)
20. \( t(r^*) := \text{TestTime}(r^*, w(r^*)); \)

**3.2 Optimize Bottom Up**

The procedure OPTIMIZE-BOTTOM UP tries to optimize the test time of a given TestRail Architecture. It does so by trying to merge the TestRail with the shortest test time with another TestRail, such that the wires that are freed up in this process can be used for an overall test time reduction. Algorithm 3 lists the pseudo-code for procedure OPTIMIZE-BOTTOM UP. It is an iterative procedure, of which every iteration consists of two steps.

In Step 1 (Lines 3–12), the procedure finds a TestRail \( r_{\text{min}} \) with minimum test time, i.e., \( t(r_{\text{min}}) = \min_{r \in R} t(r) \). The cores in TestRail \( r_{\text{min}} \) and the cores in one of the other TestRals, say \( r \), are merged into a new TestRail, say \( r^* \), with width \( \max(w(r_{\text{min}}), w(r)) \). TestRail \( r \) is selected from \( R \setminus \{r_{\text{min}}\} \) such that \( t(r^*) \) is minimum and \( t(r^*) \) does not exceed the current overall test time \( T \).

In Step 2 (Lines 13–20), the merge is effected and \( R \) is updated. As the new TestRail \( r^* \) only uses \( \max(w(r_{\text{min}}), w(r)) \) wires, \( \min(w(r_{\text{min}}), w(r)) \) wires are now freed up. The freed-up wires are distributed over all TestRals, in order to reduce the overall test time \( T \) (similar to the algorithm for SCDP and Step 3 in Algorithm 2).

The procedure ends if all TestRals have been merged into one single TestRail, or when no TestRail \( r \) can be found such that \( t(r^*) \) does not exceed the current overall test time \( T \).

**Algorithm 3 [OPTIMIZE-BOTTOM UP]**

1. \( \text{improve} := \text{true}; \)
2. while \( \|R\| > 1 \) \&\& improve \{ 
3. /* Step 1: Find merge candidate */
4. \( r_{\text{min}} := \arg\min_{r \in R} t(r); \)
5. \( T := \max_{r \in R} t(r); t(r^*) := T; \)
6. for all \( r \in R \setminus \{r_{\text{min}}\} \)
7. \( r_{\text{temp}} := r_{\text{min}} \cup r; \)
8. \( w(r_{\text{temp}}) := \max(w(r_{\text{min}}), w(r)); \)
9. \( t(r_{\text{temp}}) := \text{TestTime}(r_{\text{temp}}, w(r_{\text{temp}})); \)
10. if \( t(r_{\text{temp}}) < T \) \&\& \( t(r_{\text{temp}}) < t(r^*) \)
11. \( r^* := r_{\text{temp}}; \)
12. \( r_{\text{del}} := r; \)
13. /* Step 2: Merge and Distribute freed-up wires */
14. if \( t(r^*) < T \)
15. \( w_{\text{free}} := \min(w(r_{\text{min}}), w(r_{\text{del}})); \)
16. \( R := R \setminus \{r_{\text{del}}\} \cup \{r^*\}; \)
17. /* Step 3: If there are cores left, add them to the least-occupied TestRals */
18. if \( w_{\text{max}} < |C| \)
19. find \( r^* \) for which \( t(r^*) = \max_{r \in R} t(r); \)
20. \( w(r^*) := w(r^*) + 1; \)
21. \( t(r^*) := \text{TestTime}(r^*, w(r^*)); \)

The operation of one iteration of procedure OPTIMIZE-BOTTOM UP is illustrated by means of an example, depicted in Figure 2.

**Figure 2:** Two subsequent steps of an iteration of procedure OPTIMIZE-BOTTOM UP.

Figure 2a shows a TestRail Architecture, in which TestRail 3, containing Cores A and D, has the shortest test time. Hence, \( r_{\text{min}} \) is TestRail 3. Subsequently, the procedure looks for another TestRail with which TestRail 3 can be merged. TestRail 1, containing Core C, does not qualify, as it already determines the overall SOC test time \( T \), and adding the cores of TestRail 3 to it, would increase that test time. How-
ever, TestRail 2 does qualify, and hence a new TestRail is created containing Cores A, B, and D. The w3 wires of TestRail 3 are now freed up, and in Step 2, they are distributed over the two remaining TestRails. This leads to a decrease in test time of both TestRails, and hence decreases the overall test time T.

3.3 Optimize Top Down

The procedure OPTIMIZE-TOPDOWN tries to optimize the test time of a given TestRail Architecture in two subsequent steps. In Step 1, the algorithm iteratively tries to merge the TestRail with the longest test time (the so-called bottleneck TestRail) with another TestRail, such that the overall test time is reduced. In case Step 1 does not yield test time improvement any more, we move to Step 2. In this step, the algorithm iteratively tries to free up wires by merging two non-bottleneck TestRails, under the condition that the test time of the resulting TestRail does not exceed the overall test time. The wires that are freed up can be used for an overall test time reduction. Algorithm 4 lists the pseudo-code for procedure OPTIMIZE-TOPDOWN.

In Step 1 (Lines 1–14), the procedure iteratively does the following. It finds a bottleneck TestRail r_{max}. Subsequently, the procedure tries to find a TestRail r ∈ R \{r_{max}\}, which could be merged with TestRail r_{max} using + w(r_{max}) + w(r) wires into a new TestRail r*, such that t(r*) is minimum and t(r*) does not exceed the current overall test time T. If such a merge is found, then the merged TestRail is represented as w_{max}. For this success, the freed-up wires are distributed over the TestRails to minimize the test time of the architecture. If the search was not successful, then TestRail r_{max} is added to the set R_{skip}.

Step 2 (Lines 15–39) is quite similar to Step 1, apart from the following two differences: (1) As merge candidates, it only considers the TestRails in R \{R_{skip}\}, (2) the width of the merged TestRail + w(r) is determined by linear search between the lower limit w_L = max (w(r_{max}) + w(r)) and the upper limit w_U = w(r_{max}) + w(r), such that the width of the merged TestRail is less than the sum of the merged TestRails widths and is minimum. By this way the free width w_{max} is maximum and is represented as w_{max}. If this success is successful, the freed-up wires are distributed over the TestRails to minimize the test time of the architecture. If the search was not successful, then TestRail r_{max} is added to the set R_{skip}.

Algorithm 4 [OPTIMIZE-TOPDOWN]

The operation of Step 1 of procedure OPTIMIZE-TOPDOWN is illustrated in Figure 3. This figure shows a TestRail Architecture, in which TestRail 1, containing only Core C, is the bottleneck TestRail. TestRail 1 is merged with TestRail 3, which contains Cores A and D. The newly formed TestRail contains Cores A, C, and D, and gets assigned a width equal to w_1 + w_3. This leads to an overall test time reduction.

3.4 Core Reshuffle

The procedure CORERESHUFFLE tries to minimize the test time of a given TestRail architecture by placing one of the cores assigned to a bottleneck TestRail to another TestRail, provided that this reduces the overall test time. Algorithm 5 lists the pseudo-code for procedure CORERESHUFFLE.

Algorithm 5 [CORERESHUFFLE]

In Line 3, the procedure identifies a bottleneck TestRail r_{max}. If r_{max}
contains multiple cores, we identify the Core $j^*$ with the smallest test time. The procedure searches through the other TestRails, to see if there is one of them to which $j^*$ can be added without exceeding the overall test time $T$. If that is the case, Core $j^*$ is indeed moved from the bottleneck TestRail to this other TestRail (Lines 14–15). This procedure is repeated until the bottleneck TestRail contains only one core, or when no beneficial core re-assignment can be found.

### 4 Experimental Results

In this section, we present both sequential and concurrent test time results for our novel algorithm TR-ARCHITECT for TestRail Architecture optimization. As benchmarks, we used four SOCs from the new set of ITC’02 SOC Test Benchmarks [3]: d695, p22810, p34392, and p93791. We compare our results with four previously published approaches: (1) the Test Bus Architecture optimization method based on ILP and exhaustive enumeration in [11], (2) the faster heuristic Test Bus Architecture optimization method $Par_{eval}$ in [12], (3) the generalized rectangle-packing-based optimization (GRP) in [13], and (4) the cluster-based TestRail Architecture optimization in [14]. Note that the benchmark SOCs p22810 and p34392 are referred to as p21241 and p31108 in [12] respectively, but that as far as the data of the cores and their tests are concerned, these SOCs are identical. All approaches assume a flat design hierarchy. [11], [12], and [13] consider cores with fixed-length internal scan chains only, and hence we limit our discussion here to that case.

In terms of compute time, only the ILP/enumeration method is expensive. Even for a small number of distinct Test Buses, [11] reports compute times in the range of minutes to hours, depending on the complexity of the SOC and the total available TAM width. For all other methods, including our TR-ARCHITECT, compute time is in the range of one to few seconds for all SOCs and all TAM widths, and hence negligible.

Table 1 lists the absolute test times in clock cycles achieved by the five optimization methods for the four SOCs for a range of $w_{max}$ values. Per line of the table, the bold-font entries denote the lowest test time over all five methods. The last column in the table lists the rank of our algorithm over the total of five algorithms.

From Table 1, we can see that the sequential test times obtained from TR-ARCHITECT are generally better than the test times for concurrent test. This is due to the fact that in case of concurrent test, all cores which are being tested form a long scan chain in order to transfer data from one core to another. This results in a long scan-in/scan-out time per test session. While in case of sequential test, all cores except the core-under-test are bypassed and this results in less scan-in/scan-out time per test session.

For SOC d695, TR-ARCHITECT yields minimum test times for TAM widths $w_{max} > 40$. For $w_{max} \leq 40$, the test time achieved by TR-ARCHITECT is 0.4%–4.6% to larger than the test time obtained by the computationally intensive ILP/enumeration method. For the other TAM widths, the gain in test time as compared to ILP/enumeration method, is in the range of 4% to 15%.

For SOC p22810, TR-ARCHITECT outperforms the other methods. For $w_{max} \geq 24$, it is ranked at the first place. The test time improvement obtained by TR-ARCHITECT varies from 14% to 49% compared to the ILP/enumeration method. Figure 4 shows the TestRail Architecture (a), and the sequential test time schedule (b), obtained by TR-ARCHITECT for SOC p22810 with $w_{max} = 64$. In Figure 4, the total TAM width is partitioned over nine TestRails as $1 + 6 + 6 + 10 + 5 + 3 + 4 + 6 + 23 = 64$.

### Table 1: Experimental results for test time of TR-ARCHITECT and four others.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$T$</td>
<td>$T^*$</td>
<td>$T^*$</td>
<td>$T$</td>
<td>$T^*$</td>
<td>Rank</td>
</tr>
<tr>
<td>d695</td>
<td>16</td>
<td>42508</td>
<td>43864</td>
<td>44585</td>
<td>44530</td>
<td>44522</td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>28292</td>
<td>30032</td>
<td>31569</td>
<td>30021</td>
<td>29702</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>21566</td>
<td>22268</td>
<td>23306</td>
<td>23488</td>
<td>21690</td>
</tr>
<tr>
<td></td>
<td>40</td>
<td>17901</td>
<td>18448</td>
<td>18837</td>
<td>19034</td>
<td>17979</td>
</tr>
<tr>
<td></td>
<td>48</td>
<td>16975</td>
<td>15300</td>
<td>16984</td>
<td>16194</td>
<td>15100</td>
</tr>
<tr>
<td></td>
<td>56</td>
<td>13207</td>
<td>12941</td>
<td>14974</td>
<td>13479</td>
<td>12664</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>12941</td>
<td>12941</td>
<td>11984</td>
<td>11033</td>
<td>11033</td>
</tr>
<tr>
<td>p22810</td>
<td>16</td>
<td>462310</td>
<td>46081</td>
<td>489192</td>
<td>-</td>
<td>476301</td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>361571</td>
<td>313607</td>
<td>330016</td>
<td>-</td>
<td>310249</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>312639</td>
<td>246332</td>
<td>245718</td>
<td>259975</td>
<td>226640</td>
</tr>
<tr>
<td></td>
<td>40</td>
<td>278359</td>
<td>232049</td>
<td>190558</td>
<td>206205</td>
<td>199995</td>
</tr>
<tr>
<td></td>
<td>48</td>
<td>278359</td>
<td>232049</td>
<td>173705</td>
<td>173705</td>
<td>160221</td>
</tr>
<tr>
<td></td>
<td>48</td>
<td>288472</td>
<td>153990</td>
<td>157159</td>
<td>146390</td>
<td>145417</td>
</tr>
<tr>
<td></td>
<td>56</td>
<td>266938</td>
<td>153990</td>
<td>142342</td>
<td>133587</td>
<td>133405</td>
</tr>
<tr>
<td>p34392</td>
<td>16</td>
<td>998733</td>
<td>1033210</td>
<td>1034911</td>
<td>-</td>
<td>1023490</td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>720858</td>
<td>882182</td>
<td>759427</td>
<td>876529</td>
<td>721244</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>591027</td>
<td>663193</td>
<td>544579</td>
<td>585309</td>
<td>552746</td>
</tr>
<tr>
<td></td>
<td>40</td>
<td>544579</td>
<td>544579</td>
<td>544579</td>
<td>544579</td>
<td>544579</td>
</tr>
<tr>
<td></td>
<td>48</td>
<td>544579</td>
<td>544579</td>
<td>544579</td>
<td>544579</td>
<td>544579</td>
</tr>
<tr>
<td></td>
<td>56</td>
<td>544579</td>
<td>544579</td>
<td>544579</td>
<td>544579</td>
<td>544579</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>544579</td>
<td>544579</td>
<td>544579</td>
<td>544579</td>
<td>544579</td>
</tr>
<tr>
<td>p93791</td>
<td>16</td>
<td>19717120</td>
<td>1986200</td>
<td>1932311</td>
<td>-</td>
<td>1838402</td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>1187990</td>
<td>1209420</td>
<td>1310841</td>
<td>-</td>
<td>1240305</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>887751</td>
<td>894342</td>
<td>988039</td>
<td>947111</td>
<td>940745</td>
</tr>
<tr>
<td></td>
<td>40</td>
<td>698583</td>
<td>741965</td>
<td>794027</td>
<td>816972</td>
<td>786608</td>
</tr>
<tr>
<td></td>
<td>48</td>
<td>593733</td>
<td>599373</td>
<td>669156</td>
<td>777707</td>
<td>620877</td>
</tr>
<tr>
<td></td>
<td>56</td>
<td>514688</td>
<td>514688</td>
<td>569436</td>
<td>542445</td>
<td>530059</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>460328</td>
<td>473997</td>
<td>517958</td>
<td>467680</td>
<td>461128</td>
</tr>
</tbody>
</table>

Table 1: Experimental results for test time of TR-ARCHITECT and four others.
The number at the end of each TestRail represents its test completion time and the numbers inside a TestRail identify the cores which are connected to that TestRail.

For SOC p34392, ILP/enumeration performs better than other methods for $w_{\text{max}} = 24$. For large $w_{\text{max}}$, the test time for this SOC does not decrease beyond 544579 clock cycles, because one of the cores, Core 18 reaches its minimum test time value when assigned to a TestRail of width 10. This core then becomes the bottleneck core and determines the overall test time. All five methods reach this optimum test time. For our method, we reach this at $w_{\text{max}} = 35$ (not shown in the table), while GRP [13] already reaches this for $w_{\text{max}} = 32$

For SOC p93791, ILP/enumeration performs better than any other method. However, the TR-ARCHITECT obtains a test time comparable to ILP/enumeration method for all TAM widths. If we compare the increase in test time versus the decrease in compute time, then we see that TR-ARCHITECT performs good. The percentage increase in test time is 0.2% to 12%, while the percentage decrease in compute time is 98% to 99%. For this SOC, the Parwval method [12] also proves to be quite a strong contestant.

5 Conclusion

In this paper, we presented a novel algorithm named TR-ARCHITECT for test architecture design for core-based SOCs. TR-ARCHITECT optimizes TestRail Architectures with respect to required ATE vector memory depth and test application time. A major advantage of the TestRail Architecture over other architectures is that it allows for accessing multiple core wrappers in parallel, hence enabling the testing of the circuitry in between the cores. TR-ARCHITECT optimizes both wrapper and TAM design, works both for cores with fixed/flexible-length scan chains and all TAM widths. The algorithm requires a negligible amount of compute time and therefore is also suitable for wide TAMs. This is specifically an improvement over the CPU-intensive ILP/enumeration-based method in [11].

We presented test time results for our new algorithm TR-ARCHITECT for four benchmark SOCs taken from the set of ITC'02 SOC Test Benchmarks and compared those to previously published approaches. Our test time results are comparable or better than those previously published. As future work, we are investigating the possibilities of including the core-external tests and design hierarchy constraints in the optimization approach.

Acknowledgements

We thank our colleagues Bart Vermeulen and Harald Vranken for their useful comments on an earlier draft version of this paper.

References