Abstract— Rapid area-time estimation is an essential step for efficient design exploration of FPGA-based implementations. In this paper, we focus on area-time estimation of the controller for porting C-based functions onto commercial FPGA devices. We have adopted the one-hot encoding scheme for our FSM model, and devised techniques to estimate area-time of the next-state and control signal decoding logic. Experimental results for the Xilinx Spartan FPGA device show that the proposed model and techniques can lead to reliable area-time estimation. In particular, when compared to results from the commercial tool, the proposed area estimation strategy leads to an average absolute error of only about 10%. In addition, the maximum delay estimation error for the FSM is less than 1ns.

Keywords—Area-time estimation, controller, FPGA, FSM

I. INTRODUCTION

FPGAs (Field-Programmable Gate Arrays) have become an attractive solution to meet the technological and market uncertainties in embedded processing [1]. The advent of reconfigurable systems that incorporates microprocessor and FPGA on the same platform (e.g. Xilinx Spartan-3 [2] and Altera Stratix II [3]) necessitates efficient hardware-software partitioning strategies in emerging design flows. As such, there is a need for high-level estimation techniques that can rapidly and accurately estimate the hardware area-time information for porting high-level language (e.g. C, Matlab, etc.) to FPGA in order to facilitate efficient design exploration.

Figure 1 shows an overview of the design exploration framework that is being developed at the Centre for High Performance Embedded Systems [4]-[5]. The Trimaran compiler infrastructure [6] is relied upon to expose the hidden parallelism in the sequential C statements and to perform high-level optimization and scheduling. The output of Trimaran is an IR (Intermediate Representation) of the C-functions. Based on this IR, we perform a simple hardware binding process that attempts to reduce the complexity of the interconnectivity between registers and the selected CFUs (Custom Functional Units). CFUs are application specific functional units that have been pre-determined and stored in the CFU library. In order to perform area-time estimation, information pertaining to the controller and data-path are segregated from the IR after hardware binding. Area-time estimation is then performed on the controller and data-path separately, and the combined estimation results are used to evaluate the performance of the FPGA implementation. The process is then repeated for different sets of CFUs in order to populate the design exploration space. It is evident that rapid area-time estimation is instrumental to facilitate exploration of a large design space in reasonable time. The framework also incorporates a process to automatically generate RTL (Register Transfer Level) codes of the controller and data-path.

In this paper, we focus on area-time estimation for the controller only. Techniques for data-path estimation can be found in [4]-[5].

In the following section, we discuss related work in high level estimation of the controller. Section 3 describes techniques for constructing the FSM (Finite State Machine) model from the Trimaran IR. The proposed techniques for area-time estimation of the FSM are then presented in Section 4. Section 5 provides experimental results to evaluate the accuracy of the proposed methods. The paper concludes in Section 6.

II. RELATED WORK

High-level area-time estimation has received considerable interest in the research community for many years. Research efforts in this area are motivated by the
need to evaluate the hardware performance-cost indices of various design options early in the design phase, in order to reduce time-consuming implementation cycles. However, most of the previous works in high-level estimation do not take into account the controller even though it can contribute significantly to the total area-time of hardware accelerators.

Existing techniques for estimation of the controller can be categorized into two general approaches: 1) Estimation that is performed prior to high-level synthesis and 2) Estimation that is performed after the scheduling and hardware binding information of the controller is available.

Examples of estimation techniques based on the first approach include the work in [7], which target standard cell architectures. The area-time estimation strategy in [7] employs empirical formulas that incorporate parameters (e.g. number of operations, variables, resources, etc.) drawn from the behavioral specification. The work in [8] uses an empirical formula that takes into account the function generators and registers of the FSM for estimating the area of the FPGA controller. The number of function generators is determined from the number of conditional statements in the high-level language. The work in [9] uses a heuristic function that incorporates the number of control states, inputs-output functions and state-transitions to predict the expected schedule and binding of the FPGA controller. Only area estimation is considered in [9].

While the techniques discussed above can be undertaken earlier in the design cycle (as they do not require the scheduling and hardware binding information that are usually computed during high-level synthesis), they could lead to higher estimation inaccuracies. The work in [10] performs FPGA estimation only after a fully connected netlist of the controller has been obtained. Their estimation strategy aims to predict the effects of optimizations and technology mapping of the FPGA tool. The work in [11] uses the results of high-level synthesis for estimation and design exploration. Their method transforms the C-functions into Hierarchical Control Data Flow Graphs for area-time-power estimation of the controller.

The proposed method in this paper for area-time estimation of the FPGA controller is based on the second approach, which requires the scheduling and binding information of the controller, as this generally leads to more accurate results. The time taken for compilation, hardware binding and estimation of each C-function in the proposed framework (see Figure 1) is less than a second and hence this will not impede the efficiency of the design exploration process.

III. CONSTRUCTING THE FSM MODEL

Figure 2 shows the controller and data-path model of the target architecture. The controller is a FSM that comprises of the following components: 1) Next State Decoding Logic (NSDL), which computes the next-state of the FSM based on the current state and inputs, 2) Control Signal Decoding Logic (CSDL), which decodes the control signals to the datapath (i.e. register enable signals, multiplexer selector signals, CFU enable signals), and 3) state registers, that hold the current state of the FSM.

The controller in the target architecture adopts the one-hot encoding scheme. Unlike other encoding schemes (e.g. binary and gray), the one-hot encoding scheme requires only 1 bit to be evaluated in order to determine the state of the FSM, which greatly simplifies the decoding logic. The complex decoding circuitry of the binary and gray encoding schemes could lead to high critical path delay for large FSMs.

As shown in Figure 1, the proposed technique for controller estimation consists of two steps: 1) Constructing the FSM model, and 2) area-time estimation of the FSM model. In this section, we discuss the proposed strategies for constructing the FSM model from the Trimaran IR. In particular, we discuss strategies for constructing the NSDL and CSDL components.

A. Next State Decoding Logic (NSDL)

The NSDL is responsible for computing the next-state of FSM based on the current state and inputs. Each execution cycle or control step in the IR is modeled as a state in the FSM. As most applications are data oriented (as opposed to control oriented), most of the states in the FSM are single-state transition, which means that there is only a single transition to the next state. Multi state transitions occur when certain branch-related operations are encountered in the IR. Since identifying single state transitions from the IR is a trivial process, we will only discuss the strategies employed for identifying multi-state transitions.

Trimaran branch-related operations can be categorized into un-conditional and conditional branches. We use the CFG example in Figure 3 with four basic blocks (BB) to describe these operations. Unconditional branch operations, e.g. BRU (Branch Unconditional) are used to branch to a new address without the need to satisfy any conditions. For example, the BRU operation in BB2 results in a branch to BB4. Conditional branch operations, e.g. BRCT (Branch on
Condition True) and BRCF (Branch on Condition False) are used to jump to an address when some conditions are met. For example, the BRCF operation in BB1 results in a branch to either BB2 or BB3 depending on whether the conditions are met or not. Similarly, the BRCT operation in BB3 results in a branch to either BB3 or BB4.

\[
\begin{align*}
\text{BB1} & \quad \{ (0) \text{CMPP; (1) PPBR; (2) BRCF; } \\
\text{BB2} & \quad \{ (0) \ldots (1) \text{PPBR; (2) BRU; } \\
\text{BB3} & \quad \{ (0) \text{CMPP; (1) PPBR; (2) BRCT; } \\
\text{BB4} & \quad \{ (0) \ldots \\
\end{align*}
\]

Figure 3: Constructing the NSDL state diagram

The conditional branch operations (i.e. BRCT, BRCF) are preceded by the CMPP (Compare-To-Predicate) and PPBR (Prepare-To-Branch) operations. The CMPP operation evaluates the conditions and sets up the value of the predicate register. The PPBR operation sets up the branch target by moving the destination’s address into the branch target register. After executing the BRCT operation, the machine jumps to the address specified in the branch target register if the predicate register value is 1 (conversely, if the BRCF operation is executed, the branch to the address in the branch target register takes place only when the predicate register value is 0). Otherwise, the machine branches to a new address by incrementing the program counter.

By identifying the conditional branch instructions from the IR, a state diagram with multi-state transitions can be constructed for the FSM. The proposed method employs an internal data structure to record all the branch information of the IR. Figure 3 shows the state diagram that is constructed for BB3. Each execution cycle in the IR corresponds to one state in the FSM. For example, States 3.0 to 3.2 corresponds to the three operations in BB3 (i.e. State 3.0 corresponds to CMPP operation, State 3.1 corresponds to PPBR operation, and State 3.2 corresponds to BRCT operation). Since State 3.2 corresponds to a conditional branch, it has a transition to either State 0 (correspond to the branch target register) or to the subsequent state (State 4.0) depending on the value of the predicate register (I3).

\[
\begin{align*}
\text{State 3_0} & \quad \{ (0) \ldots (1) \text{CMPP; (2) PPBR; (3) BRCT; } \\
\text{State 3_1} & \quad \{ (0) \ldots (1) \text{CMPP; (2) PPBR; (3) BRCT; } \\
\text{State 3_2} & \quad \{ (0) \ldots (1) \text{CMPP; (2) PPBR; (3) BRCT; } \\
\end{align*}
\]

B. Control Signal Decoding Logic (CSDL)

CSDL is responsible for decoding the control signals for each state in the FSM. As mentioned earlier, the control signals to the data-path consist of register enable signals, multiplexer selector signals and CFU enable signals.

\[
\begin{align*}
\text{Reg 1} & \quad \{ (0) \ldots (1) \text{PPBR; (2) BRCT; } \\
\text{Reg 2} & \quad \{ (0) \ldots (1) \text{PPBR; (2) BRCT; } \\
\text{Reg 3} & \quad \{ (0) \ldots (1) \text{PPBR; (2) BRCT; } \\
\text{Reg 4} & \quad \{ (0) \ldots (1) \text{PPBR; (2) BRCT; } \\
\end{align*}
\]

From Controller

Control Information Table (CIT)

The proposed framework uses a CIT (Control Information Table) to store the control signal values of each state in the FSM. For example, Figure 4 shows the CIT that stores the control signal values of the multiplexers (CS0, CS1), which is used to select the inputs of the CFU from the four registers (Reg 1-4) in a time multiplexed manner. The CIT of the multiplexer indicates that Reg 1 is selected as the input to the CFU at State 0, 1 and 3; Reg 2 is selected as the input to the CFU at State 2 and 4, etc. Similarly, the CIT of a register specifies the register that is required to be written to in a particular state, and the CIT of the CFU specifies the CFUs that need to be activated in a particular state. Based on the CIT of each component, the CSDL of the controller can be constructed for area-time estimation.

IV. AREA-TIME ESTIMATION OF CONTROLLER

In this section, we present the proposed area-time estimation technique for the FSM that has been constructed using the methods discussed in the previous section. The estimation of the control path relies on a set of equations that model the way the FSM is mapped onto FPGA. In this paper, the target architecture is the Xilinx Spartan-3 device [2], which incorporates logic elements with 4-input LUT (Look-Up Table). Note that the proposed techniques are applicable for other FPGA families with LUTs that has different number of inputs.

We will discuss area-time estimation of the NSDL and CSDL components separately. The final estimation of the FSM is the sum of estimated values of all the components in the FSM, namely the state register, NSDL and CSDL.
A. Area-Time Estimation for NSDL

Using the one-hot encoding scheme, the states in Figure 3 are realized using a FF (state register) for each state as shown in Figure 5 (i.e. FF₁₂, FF₃₀, etc.). The inputs to the FFs are driven by a corresponding NSDL (i.e. NSDL₁₀ drives FF₁₀, NSDL₁₁ drives FF₁₁, etc.). Note that the succeeding NSDL (NSDL₁₀ and NSDL₄₀) of the multi-transition states (i.e. FF₁₂ and FF₃₂) has multiple inputs while succeeding NSDL of single transition states has single inputs. The combinational function for the NSDL is shown in Figure 5. Since a 4-input LUT can implement any combinational function with up to 4 inputs and one output, two LUTs is required to implement NSDL₁₀ and NSDL₄₀. The remaining NSDL with single inputs do not require an LUT implementation.

As mentioned earlier, the Trimaran IR has unconditional and conditional branches. Unconditional branches usually do not require any decoders (i.e. NSDL). An exception to this occurs when multiple unconditional branches have the same target address. This situation occurs when a “break” statement is encountered in C-codes. Since this occurrence is rare, we have assumed that all unconditional branches can be implemented without a decoder.

The estimated number of FFs \( A_{FF} \) is equal to the number of states as each state requires one state register. The critical path of NSDL \( T_{NSDL} \) can be estimated by summing the delay of the state registers and decoders.

B. Area-Time Estimation of CSDL

The CSDL decodes the output control signals for each state using an OR-function. For example, based on the CIT in Figure 4, the Boolean function of the selector signals for the multiplexers can be expressed as follows: CS₀ = State 2 or State 4; CS₁ = State 0 or State 2 or State 3.

Since a single bit is used to represent each state in the one-hot encoding scheme, the Boolean functions for CS₀ and CS₁ can be implemented using a 2-input and 3-input OR gate respectively. Hence, each of these signals can be implemented using a 4-input LUT that is configured to implement a 2-input and 3-input OR gate respectively.

When the control signals are decoded from more than four states, an OR-tree is required. Xilinx utilizes cascaded LUTs to implement OR-tree with low depth. For OR-tree with large depth, Xilinx utilizes the fast carry-chain, which results in a faster implementation. In order to estimate the area-time of the OR-tree, the proposed technique evaluates both implementations (i.e. cascaded LUT and carry chain), and selects the estimated values for the faster implementation. We will discuss techniques for estimating the area-time of the OR-tree using: 1) cascaded LUT and 2) carry-chain.

Implementation of OR-Tree using cascaded LUT

In the cascaded LUT implementation, the OR-tree is partitioned to 4-input OR groups so that each group can be mapped onto a 4-input LUT. The 4-input OR groups are then cascaded in the form of a tree as shown in the example in Figure 6.

The estimated number of FFs \( A_{FF} \) is equal to the number of states as each state requires one state register. The critical path of NSDL \( T_{NSDL} \) can be estimated by summing the delay of the state registers and decoders.

In order to estimate the area-time of the OR-tree implementation using cascaded LUT, we have developed a recursive algorithm for computing the number of LUTs
LUTs of CSDL (A_{CSDL-1}^{LUT}) and number of stages (logic depth d) that is required to map the OR-tree onto LUTs. Each iteration of the algorithm computes for a single stage of the OR-tree, and can be briefly described as follows:

1. Calculate the number of OR groups in the current stage. For example, the number of OR groups in Stage 2 is m + n, where m = \lfloor x/4 \rfloor and n = 1 if m + (x \% 4) > 4 else n = 0.

2. If m + n ≤ 4, then the subsequent stage only requires a single LUT and the algorithm terminates.

3. Otherwise, the m + n signals serve as inputs to the next stage of the OR-tree. The algorithm repeats to construct a new stage for the OR-tree.

The delay of the OR-tree implementation using cascaded LUT (T_{CSDL-1}^{LUT}) can be computed using equation (2), where \( T_{LUT} \) is the propagation delay of the LUT that is obtained from FPGA data-sheet, and \( T_{int} \) is the pre-characterized interconnection delay between the LUTs that is obtained using the method described in [5].

\[
T_{CSDL-1} = (d \times T_{LUT}) + (d - 1) \times T_{int}
\]  

(2)

**Implementation of OR-Tree using carry chain**

The OR-tree implementation in Figure 6 may lead to high critical path delay when the number of stages is large. Xilinx utilizes the fast carry-chain in the logic elements to implement OR-trees with large logic depth.

\[\text{Carry Chain} \]

As shown in Figure 7, the outputs of the OR groups in the first stage of the OR-tree (implemented using LUTs) are connected to the carry-chain which has been configured to implement an OR-function. This is achieved by tying one of the multiplexer inputs of the carry chain to logic ‘1’. This implementation is faster than the conventional OR-tree structure as all the OR groups in the first stage of the OR-tree can be computed in parallel, and their results are fed to the fast carry chain to produce the final result.

It is evident that the number of LUTs required for this implementation is the number of OR groups in the first stage of the OR-tree (see Figure 7). Hence the number of LUTs of CSDL (A_{CSDL-2}^{LUT}) that is implemented using carry chain is estimated as shown in (3).

\[
A_{CSDL-2}^{LUT} = \left\lfloor x/4 \right\rfloor
\]  

(3)

The delay of the OR-tree that is implemented using the carry-chain (T_{CSDL-2}^{LUT}) can be estimated as shown in (4), where \( T_{Max} \) is the propagation delay of the multiplexer in the carry chain, and \( T_{Carry} \) is the carry chain delay between two LUTs. The values of \( T_{Max} \) can be approximated to \( T_{LUT} \) (i.e. \( T_{Max} = T_{LUT} \)) and \( T_{Carry} \) is obtained from the FPGA data-sheet.

\[
T_{CSDL-2}^{LUT} = T_{LUT} + T_{Max} + (A_{CSDL-2}^{LUT} - 1) \times T_{Carry}
\]

(4)

\[
= (2 \times T_{LUT}) + (A_{CSDL-2}^{LUT} - 1) \times T_{Carry}
\]

\[
C. \text{ Combined Area-Time Estimation}
\]

As mentioned earlier, the proposed technique evaluates both implementations of the CSDL (i.e. cascaded LUT and carry chain), and selects the estimated values for the faster implementation. Hence, let’s first redefine the estimated area-time of CSDL as:

\[
A_{CSDL}^{LUT} = \begin{cases} A_{CSDL-1}^{LUT} & \text{if } T_{CSDL-1} < T_{CSDL-2} \\ A_{CSDL-2}^{LUT} & \text{otherwise} \end{cases}
\]

\[
T_{CSDL} = \begin{cases} T_{CSDL-1} & \text{if } T_{CSDL-1} < T_{CSDL-2} \\ T_{CSDL-2} & \text{otherwise} \end{cases}
\]

(5)

In the previous sections, we have presented the area estimation in terms of number of LUTs. However, the number of slices is a more realistic indication of the FPGA area utilization. The Xilinx slice refers to a basic configurable logic cell with 2 LUTs and 2 FFs. For simplicity, we first assume that each slice consist of 1 LUT and 1 FFs.

If all the LUTs and FFs that are used for implementation resides on different slices (i.e. on a slice, either the LUT or FF is used), the total number of slices would be sum of total number of LUTs and total number of FFs. In order to maximize the utilization of the slices, the synthesis tool will try to pack LUTs and FFs that are used for implementation onto the same slices. Hence, there is a need to model the way in which LUTs and FFs are packed for estimating the number of slices. Our observation reveals that a LUT-FF pair is usually packed onto the same slice if the LUT drives the input of the FF (e.g. NSDL3_0 and FF3_0 in Figure 5).

Based on this observation, we can now derive a mathematical model to estimate the total number of slices for the controller as shown in (6), where \( n_{LUT,FF} \) denote the estimated number of LUT-FF pairs that are packed onto the same slice. The division by 2 in (6) takes into account the fact that the Xilinx slice incorporates 2 LUTs and 2 FFs.

\[
A_{Slice}^{Controller} = \frac{A_{NSDL}^{LUT} + A_{FF} + A_{CSDL}^{LUT} - n_{LUT,FF}}{2}
\]

(6)
As mentioned earlier, each conditional branch produces two LUTs that drive the inputs of two FFs. As the synthesis tool will try to pack each of this LUT-FF pair onto a single slice, the total number of LUT-FF pairs can be estimated as $n_{\text{cond}} \times 2$, which is equal to $A_{\text{LUT}}^{\text{NSDL}}$ (see (1)). Hence, the equation in (6) can be simplified to (7)

$$A_{\text{Slice Controller}} = \frac{A_{\text{FF}} + A_{\text{LUT}}^{\text{CSDL}}}{2}$$

(7)

The critical path of the controller is estimated as the maximum register to register delay between the NSDL and CSDL components. Hence, the critical path delay of the controller can be estimated as shown in (8).

$$T_{\text{Controller}} = \max(T_{\text{NSDL}}, T_{\text{CSDL}})$$

(8)

V. EVALUATION OF THE PROPOSED TECHNIQUE

In order to evaluate the accuracy of the proposed area-time estimation for controller, we have used ten C-functions that are found in common embedded applications in EEMBC [11] and Trimaran benchmark. These ten functions were first compiled using Trimaran to produce the IR. The techniques discussed in this paper are then used to estimate the area-time of the controller. In order to compare the estimation results with actual implementation results, we have auto-generated the RTL codes for the controller, which were then implemented using Xilinx ISE 8.1.

Table 1 shows the comparison between the area-time estimation results and the actual implementation results of the auto-generated RTL codes. The target device is Xilinx xc3s5000-4. The first column lists the ten C-functions that have been used in the comparison. Columns 2 to 5 list the implementation results obtained from the Xilinx tool, while columns 6 to 9 list the corresponding estimation results that are obtained using the proposed methods in this paper. Finally, the last two columns reports the absolute error of the proposed area-time estimation (in terms of slices and ns) when compared to the Xilinx implementation.

<table>
<thead>
<tr>
<th>Function</th>
<th>Implementation</th>
<th>Estimation</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Average</td>
<td>Average</td>
<td>Error</td>
</tr>
<tr>
<td></td>
<td>Slices (ns)</td>
<td>Slices (ns)</td>
<td>Slices (ns)</td>
</tr>
<tr>
<td></td>
<td>Actual</td>
<td>Estimated</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5%</td>
<td>3.8%</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Comparison of area-time estimation results and actual implementation results.

Xilinx’s logic synthesis provides an option for us to specify the encoding scheme to be used for synthesizing the controller. However, we have not imposed on the tool to employ a specific encoding scheme as we want to provide the tool with the flexibility of choosing the best option during its optimization process. Since the number of states in the FSMs for the experiments considered ranged from medium size (i.e. 27 states) to very large size (i.e. 272 states), we expect the tool to employ the one-hot encoding scheme as this leads to better performance. Analysis of the implementation reports shows that our expectation is correct and the one-hot encoding scheme was selected by the tool for all the cases considered. Hence, this justifies the choice of employing the one-hot encoding scheme for the proposed FSM model.

The average percentage error in the number of LUTs and FFs is only about 7.5% and 3.8% respectively. These results demonstrate the accuracy of the analytical models used for estimating the LUTs and FFs of the FSM. The average area estimation error in terms of slices is about 10.3%. In addition, the estimation error (for number of slices) for the largest function used in the experiments (i.e. ViterbiDecoderIS136) is only about 4.5%. The high accuracy of the proposed area estimation technique is due to the approach adopted for predicting the packing of LUT-FF pairs onto the same slices. The absolute error of the critical path estimation is less than 1ns for all cases considered.

VI. CONCLUSION

We have proposed a high-level area-time estimation technique for the FPGA controller. The NSDL is constructed by analyzing the IR to identify conditional branch operations that can lead to multi state transitions. Other operations lead to single state transitions. The CSDL is constructed using an OR-tree that decodes the control signal outputs based on the CIT of the data-path components. We have identified two OR-tree structures for implementing the decoder logic in the Xilinx tool, which are based on 1) cascaded LUT, and 2) carry-chain implementation. The proposed strategy performs area-time estimation of the NSDL and CSDL components using a set of equations that model how the components are mapped onto the FPGA architecture. We have also formulated a model that estimates the number of slices by predicting how the LUTs and FFs are packed onto the FPGA architecture. Experimental results demonstrate the effectiveness of the proposed technique for area-time estimation of the controller.

REFERENCES

### TABLE 1: EVALUATION OF PROPOSED ESTIMATION TECHNIQUE WITH ACTUAL IMPLEMENTATION RESULTS

<table>
<thead>
<tr>
<th>Benchmark (Function)</th>
<th>Actual</th>
<th>Estimated</th>
<th>Absolute Error</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LUTs</td>
<td>FFs</td>
<td>Slices</td>
</tr>
<tr>
<td>Autocor00_bxpAutoCorrelation</td>
<td>21</td>
<td>27</td>
<td>16</td>
</tr>
<tr>
<td>Adpcm_coder</td>
<td>64</td>
<td>79</td>
<td>51</td>
</tr>
<tr>
<td>Adpcm_decoder</td>
<td>49</td>
<td>66</td>
<td>41</td>
</tr>
<tr>
<td>Comb_sort</td>
<td>21</td>
<td>34</td>
<td>20</td>
</tr>
<tr>
<td>Sha_transform</td>
<td>27</td>
<td>30</td>
<td>23</td>
</tr>
<tr>
<td>Cjpeg2_rgb_yuv_convert_jccolor</td>
<td>46</td>
<td>43</td>
<td>33</td>
</tr>
<tr>
<td>Conver00_ConvolutionalEncode</td>
<td>33</td>
<td>60</td>
<td>39</td>
</tr>
<tr>
<td>Mpeg2_dist1_motion</td>
<td>177</td>
<td>170</td>
<td>144</td>
</tr>
<tr>
<td>Mpeg2_dist2_motion</td>
<td>71</td>
<td>71</td>
<td>57</td>
</tr>
<tr>
<td>Viterbi00_ViterbiDecoder1S136</td>
<td>385</td>
<td>272</td>
<td>287</td>
</tr>
</tbody>
</table>

