FUNCTIONAL DEBUGGING OF SYSTEMS-ON-CHIP

Darko Kirovski†, Miodrag Potkonjak‡, and Lisa M. Guerra§
†Computer Science Department, University of California, Los Angeles
‡Rockwell Semiconductor Systems, Newport Beach

ABSTRACT

Due to the exponential growth of both design complexity and the number of gates per pin, functional debugging has emerged as a critical step in the development of a system-on-chip. We introduce a novel debugging approach for programmable systems-on-chip. The new method leverages the advantages of the two complementary functional execution approaches, emulation and simulation. We have developed a set of tools, transparent to both the design and debugging process, which enables the user to run long test sequences in emulation, and upon error detection, roll-back to an arbitrary instance in execution time, and switch over to simulation-based debugging for full design visibility and controllability. The efficacy of the approach is dependent on the method for transferring the computation from one execution domain to another. To enable effective transfer of the computation state, we have identified a set of optimization tasks, established their computation complexity, and developed an efficient suite of optimization algorithms.

1. INTRODUCTION

With the increasing complexity of modern designs, functional verification emerges as a time and cost dominant step in the development process. For example, verification of the UltraSPARC-I took twice as long as its design [Yan95]. Traditional approaches, such as system emulation and simulation, are becoming increasingly inefficient to address debugging needs. Emulation is fast, but provides limited design controllability and observability. Simulation has the required controllability and observability, but is six to ten orders of magnitude slower than emulation [Mar97]. For simulation, state-of-the-art RT-level simulators are capable of performing error trace and timing analysis (Interra’s Picasso [Int98]) and backtracking (Synopsys’ Cyclone [Syn98]). For programmable processor simulation, instruction-set simulators provide full system visibility at various degrees of accuracy. The debugging circuitry in the emulator, usually implemented using a JTAG boundary scan methodology [Mau86], enables controllability and observability of particular internal states. The emulation testbeds have evolved into logic (functional) porting of the processor model into arrays of rapid prototyping modules (e.g. arrays of gates, FPGAs [Apt98, Quí98, Iko98]). Such emulation engines aim to provide both high execution speed and relatively high observability and controllability of all registers [Mar98]. These systems suffer from high cost and/or reduced controllability and observability (less than 10000 signals).

Recently, a technique which leverages the advantages of the two complementary functional execution approaches, emulation and simulation, has been presented [Kir97]. However, this technique targets only statically scheduled single-core ASIC designs. Trends in the semiconductor industry show that programmable systems-on-chip are becoming a dominant design paradigm. We have developed a generalized methodology for coordinated simulation and emulation of multi-core programmable systems-on-chip. The developed approach enables the user to migrate the functional execution of the design back and forth between the simulator and emulator. Long test sequences are run in emulation. Upon error detection, the computation is migrated to the simulation tool for full design visibility and controllability. To explain how execution is transferred from one domain to another, we use the notion of a complete cut. A complete cut is a set of variables which fully determines the design state at an arbitrary time instance [Kir98]. The running design (simulation or emulation) periodically outputs its cuts. The cuts are saved by a monitoring workstation. When a transition to the alternate domain is desired, any one of the previously saved cuts can be used to initialize, and then continue execution with preserved functional and timing accuracy.

The debugging paradigm introduces a number of optimization problems and a need for efficient implementation mechanisms. We propose a suite of algorithms which effectively identifies the minimal computation state and post-process the core design and system integration to enable I/O of variables of the identified computation state. We have conducted a set of experiments on standard multi-core benchmarks to quantify the overhead induced to enable the developed debugging methodology.

Figure 1: The targeted system: core architecture, embedded software, and core integration.

2. HARDWARE AND COMPUTATION MODEL

The architecture template used to evaluate the developed debugging method is depicted in Figure 1. The architecture is typical for most modern consumer electronics devices. It contains a set of application-specific (ASIC) and slave programmable cores (SPC) connected to a shared bus. The system is controlled by a single master programmable core (MPC). Each ASIC contains a datapath and/or memory hierarchy.
We target the following heterogeneous model of computation. The backbone of the model is the semi-infinite stream (SIS) random access machine (RAM) model. The standard RAM model [Aho83] is relaxed by removing a requirement for algorithm termination. The SISRAM model provides high flexibility with well tested and widely used semantics and syntax (C and Java). The second component of the heterogeneous model is synchronous data flow (SDF) [Lee87] which is used for specifying a potentially empty set of statically scheduled islands of computations. This model facilitates optimization-intensive implementation on both ASIC and programmable platforms.

3. GLOBAL DESIGN-FOR-DEBUGGING FLOW

During the design of an application-specific core, debug functionality is added as a post-processing step. This functionality includes a set of register-to-output interconnections and a feature which enables the system integrator to select a specific cut. Since the system architecture is in general not known at the time the application-specific core is designed, this subset of interconnects should enable I/O of variables for a large number of candidate cuts. When numerous options exist for selecting a cut, the system integrator will more effectively coordinate the cut I/O.

The ASIC developer provides the system integrator with information about the set of cuts. For each cut, the variables and control steps at which each variable can be dispensed through the virtual pins of the ASIC is given. The system integrator faces three design problems. Firstly, for each ASIC a single cut has to be selected. Second, the selected cuts, jointly with the primary input and output, have to be scheduled for I/O over the available set of I/O pins. We integrated these two phases into a tight optimization loop which searches for a feasible solution. In the third step, if no feasible schedule is found, the system-on-chip cut I/O is spread in time by scheduling particular ASIC cuts sequentially.

Each PC has, in general, two components in its cut: instruction-accessible states (e.g. general-purpose registers), and states non-accessible to machine code (e.g. branch prediction hardware). The part of the cut accessible to instructions is I/O using code instrumentation. The portion of the core’s state that is not accessible by instructions must be either reset (e.g. cache flush) or I/O. The debug instructions are instrumented into the object code in a way similar to Purify [Has92]. An example of instrumented code is given in Figure 1. Instructions OUT[D[N]]; and if Debug[D[N] := IN; are sufficient to enable full controllability and visibility.

During the system software development, four subtasks are undertaken. In the first phase the minimal-size cut for each statically scheduled “computational island” is identified. In the second step, the programs are instrumented for cut I/O. In the last two phases, we identify cut variables outside the SDF islands and instrument the non-SDF code running on the MPC and SPCs with instructions which control the system cut I/O.

Figure 2 illustrates the technical details of the process of cut I/O. The code instrumentation which runs on the MPC starts the cut I/O. As shown in Figure 2, the MPC first sends a signal to the SPC which enables the system-on-chip cut I/O. The code instrumentation which runs on the MPC starts the non-SDF code running on the MPC and SPCs with instructions which control the system cut I/O. When the MPC has acknowledged the signal, the SPC initiates its cut I/O, which represents the end of the system cut I/O.

4. THE NEW APPROACH: DEBUGGING AUTOMATION

4.1. Programmable Core Cut-Selection

The I/O of cut variables on programmable engines is performed by executing debug instructions embedded into the original code by a compilation post-processing tool. The embedded instructions impact the system performance and impose storage overhead. The number of embedded instructions is directly proportional to the cardinality of the selected cut. The added instructions may also impose a timing overhead. These two problems are not identical, since different code segments can be executed with different dynamic frequencies in the SISRAM computational model. Since in Section 5 the timing overhead is shown to be minimal on a variety of applications, we focus our attention on the first problem.

PROBLEM: PC CUT SELECTION.

INSTANCE: Given a computation represented as directed graph and an integer M. QUESTION: Is there a subset of edges, corresponding to the node outputs Q,j, i = 1..N such that when deleted, leaves no directed cycles in the graph, and that N < M?

Figure 3: Pseudo-code for PC CUT SELECTION search.
This problem is NP-hard since there is one-to-one mapping between its special case, when all operations in the computation are executed exactly the same number of times, and the FEEDBACK ARC SET problem [Gar79]. For this problem, we have developed a heuristic summarized in Figure 3. Initially, the computation CDFG is partitioned into a set of strongly connected components (SCCs) using a breadth-search algorithm [Cor90]. All trivial SCCs, which contain exactly one vertex, are deleted from the resulting set because they do not form cycles. Then, for each SCC several processing steps are performed. Firstly, to reduce the solution search space, a graph compaction step is performed. Each path \( A \rightarrow B \) which contains only vertices \( V \in P, V \neq A \) with exactly one input variable is replaced with a new edge \( E_{A,B} \) which connects the source \( A \) and destination \( B \). Secondly, for each node \( V \) in the graph an objective is established. The objective evaluates the cardinalities of the newly created SCCs in the remaining graph when \( V \) is deleted. The node that has the smallest objective is deleted from the graph and added to the resulting cut. The described process is repeated until the set of nontrivial SCCs in the graph is empty.

4.2. ASIC Cut Selection

The debugging strategy comprises two modular phases, conducted by two parties in the system development process: the core provider and system integrator. The core developer selects a minimal number of register-to-output interconnects such that large number of complete cuts are available to the system integrator. An additional constraint is set on the timing occurrence of these cuts. Since the core developer does not know in advance the multi-core system configuration, its search for a set of register-to-output interconnects is targeted for large number of non-overlapping small cuts. Such subset of registers enables the system integrator to have flexibility in finding a solution to the cut scheduling problem. The definition of a debugging register subset forces selection of registers which define a large cardinality set of cuts with small cardinality, long life-times of containing cut variables, and non-overlapping life-times of variables in the set of cuts. The core developer faces an optimization problem to find a subset of registers with the smallest possible constants \( \text{Max} \) and \( \text{Min} \).

PROBLEM: DEBUGGING REGISTER SELECTION.

INSTANCE: Given a scheduled CDFG, and integers \( \text{MinOF} \) and \( \text{MinCard} \). QUESTION: Is there a subset of registers \( R_i, i = 1.R_m \), that determines a set of \( m \) distinct CDFG cuts \( C_i, i = 1,m \), and has \( \text{OF}(C_i, i = 1,m) \leq \text{Min} \) and \( \text{MinCard} \) ?

Heuristic definition 1. A “debugging register subset” is a subset of registers \( R_i, i = 1,R_m \), which defines a set of \( m \) distinct cuts \( C_i, i = 1,m \), that satisfies the following two statements:

\[
R_m < \text{Max} \text{ and OF}(C_i, i = 1,m) = \sum_{i=1}^{m} \sum_{v \in C_i} \text{LiveVar}(v) = K,
\]

where function \( \text{LiveVar}(v) \) returns the number of variables alive at control step \( C \), \( K \) is a given real number, and \( m, \text{Max}, \text{Min} \) are given integers.

A special case of this problem, with no register sharing among CDFG computation variables and no additional heuristic requirements, is NP-hard since it is equivalent to the FEEDBACK ARC SET problem [Gar79]. We have developed a heuristic to search for a debugging register subset in a scheduled and assigned CDFG. The algorithm is formally explained using the pseudo-code in Figure 4. The algorithm first partitions the CDFG into a set \( S \) of SCCs. Then, for each register \( R_i \), an objective function evaluates the set of SCCs \( S_{new} \in S_{new} \) which are result of deletion of all variables held by register \( R_i \). The objective function used to quantify the register selection is:

\[
\text{OF}(R, SR, cdg) = \frac{\text{Scar}(\text{LiveVar}(C,SR))}{\sum_{v \in V} f_{\text{Term}}(v)} = \text{Scar} \text{sum of squares of cardinalities of the set } S_{new} \text{ of SCCs created when all variables held by } R \text{ are deleted from the CDFG. LiveVar returns the number of variables alive at } C \text{ and held by the currently selected subset of registers } SR \text{. The register with the highest objective function is selected, added to the currently selected subset of registers } SR \text{, and all its variables are deleted from the original CDFG. The process of register selection is recursively repeated while the set of nontrivial SCCs is not empty.}

4.3. Multi-ASIC Cut Selection and Scheduling

We introduce an algorithmic solution which enables I/O of cut variables from multiple statically scheduled ASICs. We use the common multiple (CM) of all as the system debugging period. Within this period the algorithm tries to find a feasible schedule of variables of all ASIC cuts such that the range of control steps is minimal between the moments when the first and last variable in the ASIC subsystem cut is output.

PROBLEM: CUT SELECTION AND SCHEDULING.

INSTANCE: Given a set of cores \( ASIC \), a set of cuts \( CUT_{Core} \) for each core \( Core \in ASIC \), a set of variables \( V \) for each cut \( C \in CUT_{Core} \), a set of control steps \( CS \), for which each variable \( v \in V \) is alive, a set of \( CS \) control steps at which chip ports are idle, and integer \( \text{MaxRange} \). QUESTION: Is there a selection \( f \) of a cut \( CUT_{Core} \) for each core, such that for each variable \( v \in CUT_{Core} \) exists a distinct control step \( CS_{v} \in CS \) at which the chip port is idle, no two variables \( v \in CUT_{Core} \) are scheduled for transfer \( CS_{v} \neq CS_{v'} \) through the chip port at the same idle control step, and that the \( \text{Max}(CS_{v} - CS_{v'}) < \text{MaxRange} \) ?

The NP-complete problem of scheduling a subset of variables in a CDFG [Kir97] is a special case of this problem. We developed a most-constrained least-constraining heuristic described using the pseudo-code in Figure 5. Initially, for each ASIC, the available cuts are sorted in decreasing order with respect to the average lifetime of contained variables. The selection and scheduling search loop selects one cut for each ASIC from its list of available cuts. Cuts that contain variables with longer average lifetime are given priority. Next, within \( CM \) consecutive control steps, the subset of \( M \) consecutive control steps in which the variables of the selected cuts can be scheduled. The search is initiated by determining the lower bound on the range of control steps \( M = M_{max} \), for which all cuts can be dispensed. This bound is equal to the sum of the cardinalities of all ASIC cuts. Then, within \( CM \) consecutive control steps, a set \( T \) is found where each element \( T_{p} \in T \) represents a subset of \( N_{p} \) consecutive control steps which contains at least
We have conducted a set of experiments to evaluate the effectiveness of our system debugging paradigm. Table 1 shows experimental results for the ASIC design-for-debugging technique. All designs were synthesized using the HYPER system [Rab91]. Columns 2 - 6 of Table 1 present the iteration period in control steps, the number of variables in the computation, and total area. Column 8 presents the number of variables in the smallest cut, broken down into I/O and non-I/O cut variables. Finally, the last column shows the area overhead (OH).

<table>
<thead>
<tr>
<th>ASIC</th>
<th>T</th>
<th>Var-size</th>
<th>Area (mm²)</th>
<th>Cut size</th>
<th>Area OH (mm²)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cascade</td>
<td>4</td>
<td>51</td>
<td>3.88</td>
<td>2+4+0</td>
<td>0</td>
</tr>
<tr>
<td>Dir. Form II</td>
<td>10</td>
<td>55</td>
<td>12.08</td>
<td>2+1</td>
<td>0</td>
</tr>
<tr>
<td>Modem</td>
<td>30</td>
<td>57</td>
<td>3.67</td>
<td>2+4</td>
<td>0</td>
</tr>
<tr>
<td>Lin3</td>
<td>20</td>
<td>27.9</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Ellip</td>
<td>10</td>
<td>86</td>
<td>14.40</td>
<td>2+0</td>
<td>0</td>
</tr>
<tr>
<td>Voterra</td>
<td>15</td>
<td>40</td>
<td>2.88</td>
<td>2+4</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 1: Debug information for the ASIC implementation.

Table 2 presents the experimental results which demonstrate the efficiency of our multi-ASIC cut-selection and scheduling technique. Columns 2 - 4 present the resulting system period, the number of variables in the system cut, and the range of control steps in which the cut transfer is accomplished. By comparing these two columns, it is clear that the available system I/O-idle control steps are efficiently utilized to I/O interleaved cuts.

<table>
<thead>
<tr>
<th>Application-specific core mix</th>
<th>System period</th>
<th>Cut size</th>
<th>Steps to I/O I/O cut</th>
</tr>
</thead>
<tbody>
<tr>
<td>Link1(T=10), Link3(T=15), Cont. Fraction</td>
<td>22</td>
<td>22</td>
<td>22</td>
</tr>
<tr>
<td>Cascade, Dir. Form II, Parallel(T=10), Mat, Ellip</td>
<td>31</td>
<td>24</td>
<td>27</td>
</tr>
<tr>
<td>Cont. Fraction, Mat, Dir. Form II, Modern</td>
<td>18</td>
<td>20</td>
<td>23</td>
</tr>
<tr>
<td>Parallel(T=10), Modern, Dir. Form II, Cascade</td>
<td>19</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>Parallel(T=95), Ellip, Link3(T=10), Voterra</td>
<td>21</td>
<td>19</td>
<td>20</td>
</tr>
</tbody>
</table>

Table 2: Efficiency of the cut selection and scheduling.

5. EXPERIMENTAL RESULTS

We have presented the first debugging approach for programmable systems-on-chip that coordinates emulation and simulation. We have developed design-for-debugging algorithms for code instrumenting with cut I/O instructions and an optimization methodology for efficient cut scheduling of a set of ASICs on a shared bus. The effectiveness of the approach is demonstrated on a set of programmable and ASIC multi-core designs where full system observability and controllability have been enabled with low hardware and performance overhead.

6. CONCLUSION

Table 3 presents results that evaluate the feasibility and overhead of programmable core design-for-debugging. Applications were taken from the MediaBench benchmark suite [Lee97]. Columns 2 - 4 quantify the total number of variables in the program, the cardinality of the program cut, and the ratio of cut size to total variables. The last column determines the upper bound on the overhead on the program execution performance. Note that this overhead is negligible during emulation since system emulation is orders of magnitude faster than system simulation.

<table>
<thead>
<tr>
<th>Application</th>
<th>Variables</th>
<th>Cut size</th>
<th>% ofVars for cut I/O</th>
</tr>
</thead>
<tbody>
<tr>
<td>PGP</td>
<td>33</td>
<td>0.06</td>
<td>0.06</td>
</tr>
<tr>
<td>GSM.enc</td>
<td>140</td>
<td>12</td>
<td>8.64</td>
</tr>
<tr>
<td>JPEG.enc</td>
<td>515</td>
<td>17</td>
<td>3%</td>
</tr>
<tr>
<td>MP3.dec</td>
<td>954</td>
<td>24</td>
<td>2.5%</td>
</tr>
</tbody>
</table>

Table 3: Cut selection for programmable machines.