Optimizing Equivalence Checking for Behavioral Synthesis

Kecheng Hao and Fei Xie
Department of Computer Science
Portland State University
Portland, OR 97207
{kecheng,xie}@cs.pdx.edu

Sandip Ray
Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712
sandip@cs.utexas.edu

Jin Yang
Strategic CAD Labs, DTS
Intel Corporation
Hillsboro, OR 97124
jin.yang@intel.com

***Abstract*—Behavioral synthesis is the compilation of an Electronic system-level (ESL) design into an RTL implementation. We present a suite of optimizations for equivalence checking of RTL generated through behavioral synthesis. The optimizations exploit the high-level structure of the ESL description to ameliorate verification complexity. Experiments on representative benchmarks indicate that the optimizations can handle equivalence checking of synthesized designs with tens of thousands of lines of RTL.**

I. Introduction

Recent years have seen a gradual migration away from hand-crafted RTL towards the Electronic System Level (ESL) designs specified at high level (*e.g.*, with SystemC). Consequently, there has been interest in behavioral synthesis, *i.e.*, compilation of an ESL specification to RTL.

In a previous paper [1], we proposed a framework for certifying behaviorally synthesized RTL. The key idea was to ameliorate equivalence checking complexity by comparing the RTL with the design representation after high-level compiler and scheduling transformations have already been applied to the ESL description. Consequently, a theorem proving approach was developed to pre-verify these high-level transformations. The framework introduced *Clocked Control/Data Flow Graph* (CCDFG), a formalization of design specification that augments the traditional Control/Data Flow Graph (CDFG) with a schedule. Each compiler and scheduling transformation is viewed as a CCDFG manipulation. Theorem proving cost is amortized by the reuse of the same high-level transformations over different designs, while the semantic closeness between the "post-scheduling" CCDFG and the RTL facilitates sequential equivalence checking (SEC) between the two by permitting effective mappings of internal operations.

In this paper, we present a suite of optimizations for the SEC step above, which exploit both the explicit control and data flow representations in the CCDFG and the module structures in the ESL description. We have applied these optimizations in verification of RTL synthesized by AutoPilot [2], a state-of-the-art behavioral synthesis tool. Our experiments show that they scale SEC to tens of thousands of lines of synthesized RTL from complex behavioral specifications (*e.g.*, unbounded loops, modules, etc.), making it viable for industrial designs. We know of no other SEC framework that can handle behaviorally synthesized RTL of such complexity.

II. Background

A. Behavioral Synthesis

A behavioral synthesis tool applies a sequence of transformations to an ESL specification to transform it into RTL. As a simple illustrative example, consider the synthesis of the Tiny Encryption Algorithm (TEA) [3]. Fig. 1 shows the C specification and the circuit synthesized by AutoPilot. The following transformation phases are involved in the synthesis.

- First, *compiler transformations* are applied to the ESL description. For instance, constant propagation is used in the example to remove unnecessary variables.
- The second phase is *scheduling*, which determines the clock step for each operation. Scheduling transformations include chaining operations across conditional blocks and decomposing one operation into a sequence of multi-cycle operations. In the example, the key transformation performed is *pipelining*, to enable overlapping execution of operations from different loop iterations.
- The third phase is *resource binding and control synthesis*, which binds operations to functional units, allocates and binds registers, and generates the control circuit to implement the schedule. For instance, the "+" operation is bound to a hardware adder, and a finite-state machine (FSM) module is generated to control circuit operations.

After these transformations, the design can be expressed as RTL. This design is subjected to further optimizations and manual tweaks to satisfy performance and power goals.

B. Overall Framework

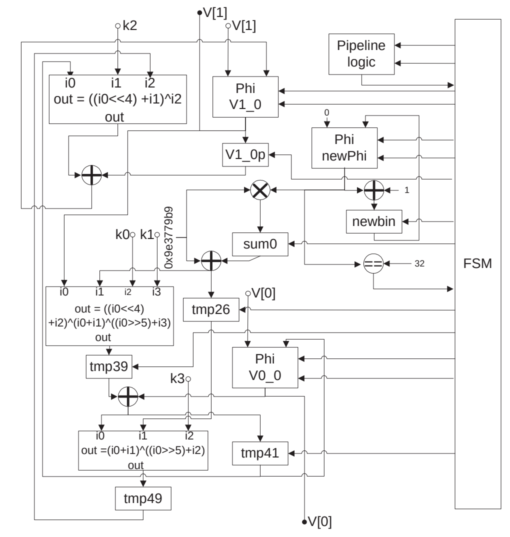
We implement SEC on top of a certified *reference flow*, consisting of pre-verified compiler and scheduling transformations. We refer to these pre-verified transformations as *primitive transformations*. Examples of primitive transformations include (1) refinement of operation schedules over multiple cycles, (2) operation balancing, and (3) pipelining. Each such transformation is *generic* and independent of the nuances of a specific design, *e.g.*, correctness of operation balancing only requires that the operation considered is associative and commutative. The transformations are culled from a production synthesis tool (*e.g.*, AutoPilot), and formalized as CCDFG manipulations. Given an ESL specification $\mathcal{E}$ and

```
void encrypt (uint32_t* v, uint32_t* k)
{
  /* set up */
  uint32_t v0=v[0], v1=v[1], sum=0, i;
  /* a key schedule constant */
  uint32_t delta=0x9e3779b9;
  /* cache key */
  uint32_t k0=k[0], k1=k[1],
           k2=k[2], k3=k[3];

  /* basic cycle start */
  for (i=0; i < 32; i++) {
    sum += delta;
    v0 += ((v1<<4)+k0)^(v1 + sum)
            ^((v1>>5)+k1);
    v1 += ((v0<<4)+k2)^(v0 + sum)
            ^((v0>>5)+k3);
  }

  /* end cycle */
  v[0]=v0; v[1]=v1;
}
```

(A)



(B)

Fig. 1. (A) C code for TEA encryption function. (B) Schema of RTL synthesized by AutoPilot.

the corresponding RTL $\mathcal{D}$, the overall verification framework involves the following automatic steps:

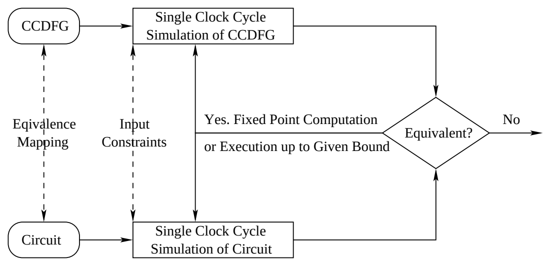
- Extract the initial CCDFG $\mathcal{C}$ from $\mathcal{E}$.
- Apply primitive transformations from the reference flow, following their application sequence in the synthesis tool. The result is a CCDFG $\mathcal{C}'$ that is close to $\mathcal{D}$ in abstraction.
- Apply SEC between $\mathcal{C}'$ and $\mathcal{D}$.

In addition to transformations, a production synthesis tool implements heuristics to control their application order. However, since such heuristics typically affect only performance, not correctness, the reference flow is oblivious to them.

C. CCDFG

A CCDFG is a CDFG augmented with a schedule. Fig. 2 shows three CCDFGs for TEA. In our framework, CCDFGs provide a uniform abstraction of internal design representations used in different behavioral synthesis tools (*e.g.*, AutoPilot, Spark [4], etc.). This enables us to view synthesis transformations uniformly as CCDFG manipulations.

The formalization of CCDFG assumes that the underlying language provides the semantics for *primitive operations* (*e.g.*, arithmetic operations, comparison, etc.). The key components of the formalization are (1) control and data flow graphs, (2) microstep partition, and (3) schedule. Following standard conventions, the control flow is broken up into of basic blocks; correspondingly data dependencies follow the "read after write" paradigm: op_j is dependent on op_i if op_j occurs after op_i in a control path and computes an expression over some variable v that is assigned most recently by op_i in the path. A *microstep partition* is a partitioning of operations in a basic block such that if op_i and op_j are in the same partition then their execution order is irrelevant to control and data dependencies. Each component of a microstep partition is a *microstep*. A *schedule* is a *grouping* of microsteps; informally,



CCDFG
Single Clock Cycle Simulation of CCDFG
Eqivalence Mapping
Input Constraints
Yes. Fixed Point Computation
or Execution up to Given Bound
Equivalent?
No
Circuit
Single Clock Cycle Simulation of Circuit

Fig. 4. Dual-Rail simulation scheme for SEC between CCDFG and circuit.

if m_0 and m_1 belong to the same scheduling step then they are executed within the same clock cycle. A CCDFG execution is formalized through state-based semantics. A *CCDFG state* (resp., *CCDFG input*) is a valuation of the state (resp., input) variables. Given a sequence of inputs, an *execution* of a CCDFG G with microstep partition M and schedule T is a sequence of CCDFG states that corresponds to an evaluation of the microsteps of M respecting T.

III. Equivalence Checking

Since the CCDFG G represents the design after compiler and scheduling transformations, there is direct correspondence between operations in G and their implementations in the synthesized circuit M. To exploit this, we define a mapping $EMap$ from the operations in G to combinational nodes in M: each operation is mapped to the node that implements it. Fig. 3 shows the mapping for the synthesized circuit of TEA. Given $EMap$, we implement SEC as a dual-rail symbolic simulation (Fig. 4), with the two rails synchronized by clock cycle. The following steps are performed at clock cycle k.

1) For the current CCDFG state x_k and circuit state s_k we check whether for input i_k, the inputs to each operation

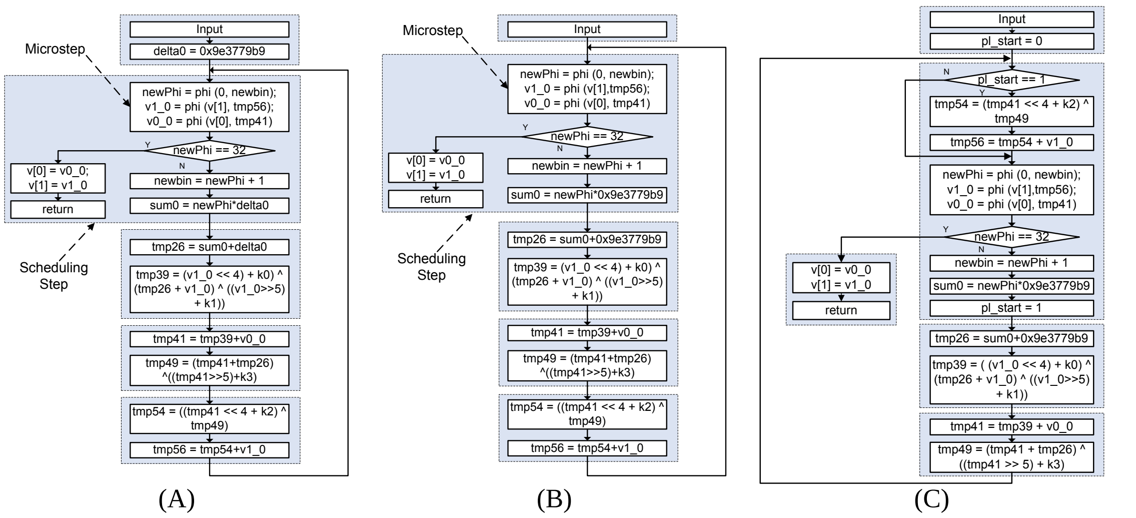


Fig. 2. (A) Initial CCDFG of TEA encryption function. (B) Transformed CCDFG after constant propagation. (C) Further transformed CCDFG after pipelining. The shaded regions represent scheduling steps, and white boxes represent microstep partitions. For brevity, only the control flow is shown.

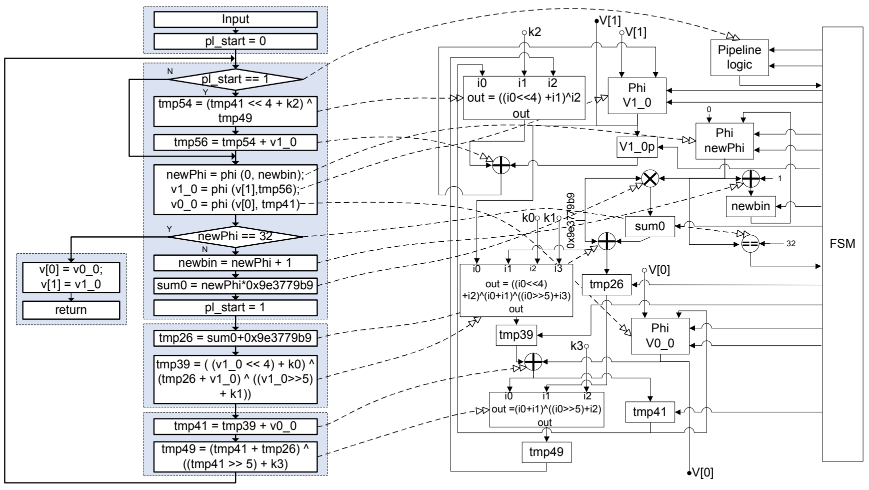


Fig. 3. Operation mapping from CCDFG to circuit for TEA; dashed lines represent mapping from CCDFG operations to circuit nodes.

op in the scheduling step t_k are equivalent to the inputs to $EMap(op)$. If yes, continue; else report failure.

2) We simulate G by executing t_k on x_k under i_k to obtain the state x_{k+1} and recording the outputs of each $op \in t_k$. We simulate M for one clock cycle from s_k with input $EMap(i_k)$ obtaining state s_{k+1}. The outputs of each op are checked for equivalence against the outputs of $EMap(op)$. If yes, continue; else, report failure.
3) The next scheduling step t_{k+1} is determined based on the control flow. If t_k has multiple outgoing control edges, the last microstep of t_k executed is identified. The outgoing control edge from this microstep whose condition evaluates to true leads to t_{k+1}.

The simulation proceeds until (i) the equivalence check fails, (ii) the end of the input sequence is reached (for bounded check), or (iii) a fixed point is reached (for unbounded check).

The dual-rail scheme is implemented at both bit-level and word-level. For bit-level, we use the Intel *Forte* environment [5] where symbolic states are represented using OBDDs. Our word-level implementation is based on Satisfiability Modulo Theory (SMT), using the CVC3 SMT engine [6]. Word-level simulation is viable since word-level mappings between operations and circuit nodes are explicit. Bit vectors are used to encode variables in the CCDFG and circuit; CVC3 checks input/output equivalence and determines the control paths.

The bit-level and word-level checkers are complimentary. The bit-level checker ensures decidability of SEC, while the word-level checker is crucial to scalability. Furthermore, we implement modular analysis, which permits the word-level checker to compositionally use results from bit-level checking as follows. Suppose M is a module of modest complexity but is awkward to check at word-level. Then the bit-level checker can compare the CCDFG of M with its circuit implementation; subsequently, while verifying a module that invokes M, the word-level checker can treat the CCDFG of M and its circuit implementation as equivalent black boxes.

IV. Optimizations

The scalability of the SEC algorithm critically depends on three key optimizations, which exploit the close correspondence between CCDFGs and their synthesized RTL designs.

A. Cutpoints

The cutpoint optimization involves pre-verifying comparison of specific CCDFG operations and their circuit implementations off-line. Subsequently, during SEC, these operations are replaced in the CCDFG and RTL by equivalent symbols. Note that only the equivalences (not computations) are relevant to SEC; if the inputs to a cutpoint are equivalent, their outputs can be replaced by equivalent symbols, causing only equivalences (not outputs themselves) to be propagated.

We utilize two types of cutpoints, *combinational* and *sequential*. Combinational cutpoints are applicable to combinational portions, and have been studied extensively [7]. RTL designs with complex combinational circuits are generated due to transformations such as loop unrolling: in the TEA example, AutoPilot can fully unroll the *for* loop, creating complex combinational circuits by aggregating operations from different iterations. Sequential cutpoints cut sequential circuits and keep complex expressions from propagating across clock cycles.

In the TEA example (Fig. 3), the scheduling step starting with the conditional *pl_start==1* and ending with the assignment *pl_start=1* is implemented as a combinational block that can be cut at all operations, *e.g.*, the one computing *tmp54*; the equivalence of this operation with the corresponding RTL is certified separately (*e.g.*, by theorem proving). On the other hand, the operation that computes *tmp49* can be used as a sequential cutpoint since it connects two scheduling steps.

To explain the role of post-scheduling CCDFGs in cutpoint optimization, note that the ESL specification is unclocked while the RTL is clocked. Furthermore, after application of high-level transformations, the RTL has little correspondence in internal operations with the behavioral description, making it difficult to identify cutpoints. However, this problem is eliminated in our framework since there is a readily available correspondence with the post-scheduling CCDFG, *e.g.*, the operation-to-resource mapping, which provides natural candidates for cutpoints.

B. Cut-loop optimization

A major challenge in SEC is termination, which typically requires expensive fixed-point computation. Termination becomes a problem when the input description contains unbounded loops. Consider the CCDFG of the Greatest Common Divisor (GCD) algorithm shown in Fig. 5. The bit-level symbolic simulation for GCD, even for 8-bit integers, involves more than 6850 seconds and 1197606 BDD nodes. A naive fixed-point computation at word-level is also expensive. Even for designs with deep bounded loops (*e.g.*, TEA), full unrolling is too expensive for both bit-level and word-level simulations.

Our solution is the *cut-loop optimization*, which "cuts" the loop, reducing the fixed-point computation to three checks, *i.e.*, at the entry, body, and exit. At entry, we check equivalence

```
int gcd (int a, int b)
{
  int t;
  do {
    if (a >= b) a=a–b;
    else {t=a;a=b;b=t;}
  } while (b != 0);
  return a;
}
```

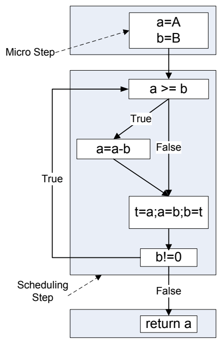


Fig. 5. C source code and CCDFG for GCD

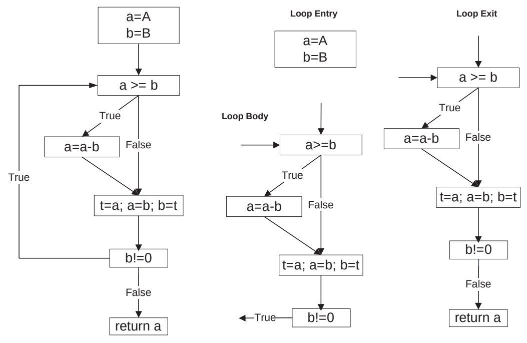


Fig. 6. Cut-loop optimization for GCD example

between the CCDFG and the RTL for the path to the initial loop entry. For the body, we check that if (1) equivalence is maintained at the loop join point, and (2) the loop does not exit, then equivalence is maintained after one iteration. For the exit, we check that if (1) equivalence is maintained at the loop join point, and (2) the loop exits, then equivalence is maintained at the loop exit. The loop structure and entry point information are available from the synthesis tool. The checks above are inspired by inductive assertions in software verification [8], [9]: the three checks are essentially the proof obligations discharged by a verification condition generator, if we think of equivalence with RTL as the invariant maintained by the loop. Using ACL2, we proved that the checks guarantee word-level equivalence over the entire loop execution.

We illustrate cut-loop optimization on the GCD example in Fig. 6. At the loop entry, the check that a and b are equivalent to their RTL counterparts is trivially true since they are inputs. For the body check the condition $b! = 0$ is applied to ensure the iteration does not exit, and for the exit check the condition $b = 0$ is applied to ensure the loop exits. For both body and exit checks, the condition being checked is that if a and b are equivalent before executing $a >= b$ then they are equivalent after one iteration. With this optimization, word-level SEC on GCD finishes within two seconds. The cut-loop optimization is also useful for deep bounded loops, *e.g.*, we achieved major speed-up for word-level SEC on TEA (cf. Section V).

Note that loop detection is greatly simplified since CCDFGs are derived from ESL designs by applying primitive transformations. Nested loops are handled by recursive loop reduction.

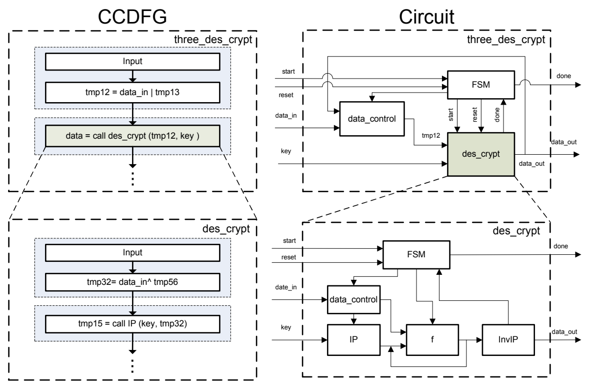


Fig. 7. Modular SEC for 3DES

C. *Modular analysis*

Synthesized RTL is often large and complex, *e.g.*, for 3DES design, AutoPilot generates 18053 lines of Verilog. Behavioral synthesis reduces RTL size via modular reuse: without modules, the RTL for 3DES would be 128K lines.

Modules may be present in input description or introduced by behavioral synthesis. To support modules, CCDFGs are extended with function calls. An example function invocation in the 3DES CCDFG is shown in Fig. 7. With modules, a given behavioral description corresponds to several CCDFGs (each corresponding to a module). A module can be either combinational or sequential. A combinational module returns in the same clock cycle in which it is invoked, while a sequential module takes several cycles. Note that the top-level CCDFG may not capture all the scheduling steps since some are in other sequential modules. In the synthesized RTL, there is a module for each CCDFG. In addition to RTL code implementing functionality, there is additional code for interfaces, *e.g.*, a module commonly needs `reset`, `start`, and `ready` signals besides input/output data signals.

One naive approach to handle modules is to unfold them, causing each module to be analyzed at each invocation. We prefer compositional analysis of each module separately. Our scheme works as follows.

- For each module M, the CCDFG and RTL for M are checked for equivalence separately.
- When verifying a module M' that invokes M, the invocation of M in the CCDFG and RTL of M' are replaced by equivalent uninterpreted functions.

The equivalence between function invocation in CCDFG and module interfacing mechanism in RTL is pre-certified. Modular analysis is possible because of explicit correspondence between the CCDFG and the RTL of a module: since we use the same module structure used in the synthesis, the decomposition does not introduce over-approximations. Currently, we do not handle recursive modules since recursions in ESL descriptions are typically removed by compiler transformations; however, modular analysis can be extended to recursion by replacing the callee with a "module summary", analogous to procedure summaries in software verification [10].

TABLE I
DESIGNS, FEATURES, AND OPTIMIZATIONS

Designs	Features	Optimizations
GCD	Unbounded Loop	Cut-Loop
DCT	Sequential without Loop	Cutpoint
TEA	Bounded Loop Unrolled Loop	Cut-Loop Cutpoint
DES	Bounded Loop Unrolled Loop High Sequential Complexity	Cut-Loop Cutpoint
3DES	Bounded Loop Unrolled Loop High Sequential Complexity	Cut-Loop Cutpoint Modular Analysis
3DES_key	Bounded Loop Unrolled Loop High Sequential Complexity High Combinational Complexity	Cut-Loop Cutpoint Modular Analysis

TABLE II
BIT-LEVEL EQUIVALENCE CHECKING STATISTICS

Bit Width	# of Circuit Nodes	Time (Sec.)	BDD Nodes
2	96	0.02	503
3	164	0.05	4772
4	246	0.11	42831
5	342	0.59	16244
6	452	12.50	39968
7	576	369.31	220891
8	714	6850.56	1197604

V. EXPERIMENTAL RESULTS

Table I illustrates the designs used to evaluate our approach. Each design is synthesized by AutoPilot. The designs are selected carefully to exercise different facets of our framework. GCD, albeit simple, is included to illustrate unbounded loops that demand the cut-loop optimization. DCT (Discrete Cosine Transform) requires handling a long sequential computation without loop. We additionally use a number of increasingly elaborated cryptographic encryption algorithms, *e.g.*, TEA, DES, 3DES, and 3DES_key (3DES with key generation). DES, 3DES and 3DES_key contains bounded loops and benefit from cut-loop; their sequential and combinational complexities also illustrate the role of cutpoints. 3DES and 3DES_key have modular structures and modular analysis is vital to discharge their SEC. DES was deliberately synthesized without modules to further investigate the role of modular analysis. All experiments were conducted on a workstation with 3GHz Intel Xeon processor with 2GB memory.

To establish a baseline, we use the bit-level checker on the GCD implementation (cf. Table II). Since all operations are bit-blasted, the running time grows exponentially with bit width. For 8-bit GCD, SEC takes about 2 hours. Pure bit-level SEC is thus not feasible for more complex designs.

Table III shows the results on word-level SEC for all the designs from Table I. Here, "-" signifies "out of time or memory", "CP" for cutpoints, "CL" for cut-loop, and "MA" for modular analysis. The "NO" column represents "no optimizations": it is clear that without the optimizations, SEC cannot handle long computation sequences or loops. Since DCT contains only sequential computations and no modules,

TABLE III
WORD-LEVEL EQUIVALENCE CHECKING STATISTICS

Design	GCD			DCT		TEA			DES			3DES			3DES_key		
RTL Size (# Lines)	364			688		1001			11520			18053			79976		
Optimizations	NO	CP	CP CL	NO	CP	NO	CP	CP CL	NO	CP	CP CL	NO	CP MA	CP MA CL	NO	CP MA	CP MA CL
Time (Secs)	-	-	2	71	30.1	-	116	15.6	-	5896	1482	-	872.5	355.7	-	2868.5	2351.7
Memory (MB)	-	-	4.1	92.16	49.2	-	141.3	24.6	-	614.4	426.4	-	114.7	59.4	-	307.2	307.2

cut-loop and modular analysis are not applicable; however, cutpoint optimization reduces the symbolic simulation cost to about half, in both time and memory usage. Cutpoints, together with modular analysis, can handle long computation sequences and bounded loops, (*e.g.*, TEA, 3DES, and 3DES_key), but blows up on fixed-point computation for unbounded loops (*e.g.*, GCD), underlining the need for cut-loop. The cut-loop optimization handles unbounded loops, while also reducing the time and memory usage for designs with bounded loops. The savings from cut-loop are relatively less for 3DES_key since the design contains large combinational computations (for generating the key) which overshadow loop unrolling cost. The results on DES highlight the importance of modular analysis when possible: although the RTL is smaller than 3DES and 3DES_key, the time and memory usage is higher due to lack of modules (and hence, modular analysis); for 3DES and 3DES_key, even synthesis by AutoPilot fails without modules.

The results indicate that word-level SEC with our optimizations can scale to realistic designs. Note that each of DES, 3DES, and 3DES_key is over $10,000$ lines of RTL, and 3DES_key (even with modules) involves about $80,000$ lines. We know of no other framework that can apply SEC on behaviorally synthesized RTL at this scale.

VI. Related Work

Koelbl *et al.* [11] provide a tutorial introduction on methods of comparing high-level designs with RTL. There has been recent work on combinational equivalence checking between designs in SystemC and RTL [12]. Note that the abstraction gap between the two models often necessitates cost-prohibitive symbolic co-simulation for input-output equivalence.

Kundu *et al.* [13] propose the use of bisimulation correspondence to validate designs generated by behavioral synthesis. Their approach is implemented for the Spark synthesis tool [4]. However, they do not provide a uniform design representation across different synthesis tools or implement optimizations necessary for scaling to design sizes that we handle.

There has also been significant work on SEC between RTL and gate-level hardware designs [14], [15]. Furthermore, there have been effort for SEC between software specifications and hardware implementations [16]: GSTE assertion graphs were extended so that an assertion graph edge has associated assignments that update state variables. These extended assertion graphs motivated our formulation of CCDFGs, which preserve both control/data flows and the schedule. Finally, there has been work on equivalence checking with other graph representations, *e.g.*, Signal Flow Graph [17].

VII. Conclusion and Future Work

We have presented optimizations enabling a scalable equivalence checking framework for RTL designs generated through behavioral synthesis. These optimizations target different design features, which have enabled the use of word-level SEC for verifying designs with tens of thousands of lines of RTL synthesized by a state-of-the-art behavioral synthesis tool (*e.g.*, AutoPilot). One direction for future research is to check synthesized RTL that has undergone significant manual tweaks (for power, performance, etc.), thereby distorting the mapping with the corresponding CCDFG operations.

Acknowledgment

This research was partially supported by National Science Foundation Grants #CCF-0916772 and #CCF-0917188 and by a research grant from Intel Corporation.

References

[1] S. Ray, K. Hao, F. Xie, and J. Yang, “Formal verification for high-assurance behavioral synthesis,” in *Proc. of ATVA*, 2009.
[2] AutoESL, *AutoPilot Reference Manual*, 2008.
[3] D. J. Wheeler and R. M. Needham, “Tea, a tiny encryption algorithm,” in *Fast Software Encryption*, 1994.
[4] D. Gajski, N. D. Dutt, A. Wu, and S. Lin, *High Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers, 1993.
[5] C.-J. Seger, R. Jones, J. O’Leary, T. Melham, M. Aagaard, C. Barrett, and D. Syme, “An industrially effective environment for formal hardware verification,” *TCAD*, vol. 24, no. 9, 2005.
[6] C. Barrett and C. Tinelli, “CVC3,” in *CAV*, 2007.
[7] A. Kuehlman and K. F, “Equivalence Checking Using Cuts and Heaps,” in *DAC*, 1997.
[8] R. Floyd, “Assigning Meanings to Programs,” in *Mathematical Aspects of Computer Science, Proc. of Symposia in Applied Mathematics*, 1967.
[9] C. A. R. Hoare, “An Axiomatic Basis for Computer Programming,” *Communications of the ACM*, vol. 12, no. 10, 1969.
[10] T. Ball and S. K. Rajamani, “Automatically Validating Temporal Safety Properties of Interfaces,” in *SPIN*, 2001.
[11] A. Koelbl, Y. Lu, and A. Mathur, “Formal Equivalence Checking between System-level Models and RTL,” in *ICCAD*, 2005.
[12] A. J. Hu, “High-level vs. RTL combinational equivalence: An introduction,” in *ICCD*, 2006.
[13] S. Kundu, S. Lerner, and R. Gupta, “Validating High-Level Synthesis,” in *CAV*, ser. LNCS, A. Gupta and S. Malik, Eds., vol. 5123. Springer, 2008, pp. 459–472.
[14] J. Baumgartner, H. Mony, V. Paruthi, R. Kanzelman, and G. Janssen, “Scalable sequential equivalence checking across arbitrary design transformations,” in *ICCD*, 2006.
[15] D. Kaiss, S. Goldenberg, Z. Hanna, and Z. Khasidashvili, “Seqver: A sequential equivalence verifier for hardware designs,” in *ICCD*, 2006.
[16] X. Feng, A. J. Hu, and J. Yang, “Partitioned model checking from software specifications,” in *ASP-DAC*, 2005.
[17] L. Claesen, M. Genoe, and E. Verlind, “Implementation/specification verification by means of SFG-Tracing,” in *CHARME*, 1993.