High-Level vs. RTL Combinational Equivalence: An Introduction

(Invited Paper)

Alan J. Hu
Department of Computer Science
University of British Columbia
ajh@cs.ubc.ca

Abstract— **With increasing use of higher-than-RTL specifications as the starting point of designs, a pressing need has emerged for equivalence verification between a high-level (e.g., non-synthesizable software) model and RTL. Other papers in this invited session discuss techniques for dealing with the sequential aspects of this problem. This paper presents an introduction to the main ideas for the combinational aspect: assuming we are given two combinational descriptions, one high-level and one RTL, how do we automatically and efficiently verify equivalence?**

I. Introduction

Increasing complexity is forcing design to move above RTL. Growing adoption of ESL, transaction-level models, MATLAB models, and C-based HDLs are all examples of this trend. The higher-level model is typically in software or a software-like language, so equivalence verification between the high-level software model and the RTL is needed, analogous to the current use of RTL-to-gate combinational equivalence verification.

The general problem of comparing a high-level model to RTL is arbitrarily complex and ill-defined. For example, consider the difficulty of comparing an instruction set architecture (ISA) specification to an out-of-order, superscalar processor implementation, or as an extreme example, comparing an ideal, mathematical signal processing algorithm to a lossy, imprecise implementation that is judged "good enough" based on user studies.

To clarify the general problem, and to have a realistic hope of creating automatic tools, it is helpful to break it down into specific sub-problems and make some simplifying assumptions. At the most abstract level, the questions are what it means for the two models to be equivalent, and when that equivalence is checked. For example, when comparing an ISA specification to a superscalar processor implementation, "equivalence" usually means that the programmer-visible state (e.g., architectural registers) is identical, and the timing correspondence is complicated by the fact that instructions can execute out-of-order and might not complete due to misspeculation or exceptions. Other examples of timing mismatches between a high-level and RTL model are untimed transaction-level models, pipelining, and retiming. Once the timing issues have been resolved, the next challenge is to find the correspondence between the states of the two models.

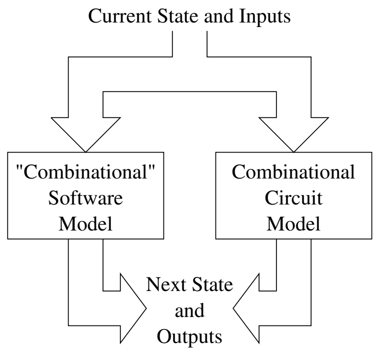

Fig. 1. High-Level-vs-RTL Combinational Equivalence. We assume that timing mismatches and equivalent-latch mappings between the high-level and RTL model have already been computed, reducing the problem to combinational equivalence. The software model might have complex control and data flow (e.g., branches, function calls), but is assumed to behave "combinationally", i.e., compute a result given the inputs. Do the two models compute the same result?

Often, the assumption is made that the two models are similar enough that it is possible to find a mapping between latches in the RTL and variables in the high-level model. With such a mapping, the remaining problem reduces to checking, on each clock cycle, the combinational equivalence of the two models. Koelbl et al. provide a broad tutorial overview of these issues and the general high-level-vs.-RTL equivalence checking problem [12].

The other papers in this invited session discuss techniques for dealing with the sequential aspects of the problem. This paper presents an introduction to the main ideas for the combinational aspect: assuming we are given two combinational descriptions, one high-level and one RTL, how do we automatically and efficiently verify equivalence? (Fig. 1.) This problem is the simplest and most fundamental, yet it is still practically relevant, as many leading companies have adopted a methodology that includes a cycle- and pin-accurate high-level model to facilitate verification. The combinational problem is also the foundation for more general equivalence-checking approaches, since they all aim to reduce the general problem to combinational equivalence.

1-4244-9707-X/06/$20.00 ©2006 IEEE

II. RTL-VS-GATE COMBINATIONAL EQUIVALENCE

We start by reviewing the problem of checking combinational equivalence between RTL and gate-level models (or equivalently, between two gate-level models, since RTL is easily converted to gate-level). The problems and solutions are analogous to high-level-vs.-RTL, and they can be more simply understood using gate-level examples.

Consider the simple example in Fig. 2. We are given two combinational circuits, with equivalent inputs, and want to compute whether the outputs are always equivalent. The basic approach to this problem is to use *symbolic simulation* [5] to automatically compute the input/output relationship for each circuit, and then compare these relations. Symbolic simulation works like normal simulation — it applies inputs to the circuit, and then simulates the behavior of each gate to compute its output. However, the inputs are given as variables, so the computed output is a symbolic expression in terms of the inputs. Returning to Fig. 2, for the left-hand circuit, we might compute $b \wedge c$ as the "value" of the output of the first AND gate, and then $(b \wedge c) \wedge d$ as the value of the wire labeled `x`, and finally $a \oplus ((b \wedge c) \wedge d)$ for the output wire `f` (where $\oplus$ denotes exclusive-OR). Similarly, we would compute $a \oplus (b \wedge (c \wedge d))$ as the output of the right-hand circuit. Verification then consists of proving (e.g., via Boolean algebra) that these two expressions are equivalent.

Note that we can use any representation we choose for the symbolic expressions, as long as we can construct the symbolic expression for the output of any gate based on the symbolic expressions for its inputs. For example, BDDs [4] showed considerable promise for this purpose and are still one of the most common representations for Boolean functions: they are empirically compact and efficient for common functions, and they are a canonical representation, so testing for logical equivalence is trivial. Unfortunately, for real, industrial verification problems, the BDDs grow too big to be computed. As an alternative that avoids any space blow-up, we could introduce a fresh variable name for each wire and have symbolic simulation build up a set of constraints on the values of these variables. Returning once again to Fig. 2, if we create new variable names v_i for all the internal wires, then we can derive the set of constraints: $(b \wedge c = v_1) \wedge (v_1 \wedge d = v_2) \wedge (a \oplus v_2 = f)$ for the left-hand circuit; and $(c \wedge d = v_3) \wedge (b \wedge v_3 = v_4) \wedge (a \oplus v_4 = g)$ for the right-hand circuit. We can conjoin all these constraints together, along with the constraint $(f \neq g)$, and throw the whole formula at a SAT solver;[1] the formula is unsatisfiable iff the two circuits are equivalent. This approach has no space blow-up, but blows up in run time on industrial verification problems instead.

The major practical breakthrough for combinational equivalence checking was the idea of cutpoints [1], [3]. Given two combinational circuits whose functional equivalence needs to be verified, the cutpoint approach assumes that they are structurally similar, so the two circuits likely contain subcircuits that are also functionally equivalent. Accordingly, the idea is to look for points in the two circuits that can be proven to be equivalent. The equivalent logic is cut out of the circuits and is replaced by a new primary input. (Fig. 2.) If we can repeat this process all the way to the primary outputs, we have proven the two circuits equivalent, thereby reducing the original verification problem into a sequence of simpler verification problems. Note that the method is conservative: if we fail to prove the circuits equivalent, we cannot conclude that they are inequivalent without further computation. (Fig. 3.) Minimizing the cases where the method is unable to prove the equivalence of equivalent circuits (called "false negatives" or "false inequivalence") has been an active research area. In general, the solutions to this problem are ways to re-introduce constraints on the cutpoints, either in advance [3] or as needed [11], [14].

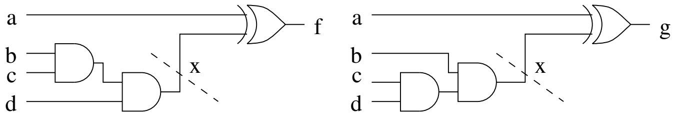

Fig. 2. Simple Cutpoint Example. To introduce cutpoint x, we first verify that $(b \wedge c) \wedge d$ is equivalent to $b \wedge (c \wedge d)$. Then, we can verify that f is equivalent to g because both are equal to $a \oplus x$.

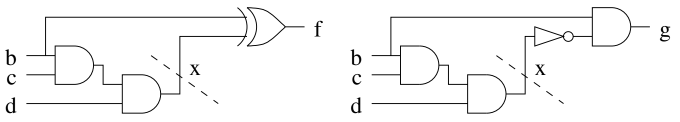

Fig. 3. False Inequivalence. Cutpoint verification fails because $f \neq g$ when $b = 0$ and $x = 1$. However, this is a false inequivalence, because if $x = 1$, then b must be 1.

An important problem is heuristics to find good candidate cutpoints, since a brute-force search of all possible cutpoints is too expensive. A common approach starts with a quick structural comparison to isolate differences between the two circuits being compared (akin to the Unix utility `diff`). Next, for each wire, a signature is computed that is its value during (normal, non-symbolic) simulation of a few hundred or thousand random inputs. Wires that have the same signature are good candidate to attempt to prove equivalent and use as cutpoints.

Overall, combinational equivalence checking is one of the biggest success stories for formal verification, having completely supplanted the formerly time-consuming task of RTL-vs.-gate-level simulation. To read further on this topic, some good surveys include [11] and [10]. Here, we consider how we can use these ideas for high-level-vs.-RTL combinational equivalence checking.

III. ANALYZING A HIGH-LEVEL MODEL

When moving from RTL-vs.-gate to high-level-vs.-RTL, the immediate question is what is different about the problem, and the obvious answer is, "the high-level model." In particular, the key question is how to analyze a software model and derive

[1]The state-of-the-art for SAT solvers changes rapidly. The website `http://www.satcompetition.org` has results from periodic competitions, as well as links to the major SAT reference sites, like `http://www.satlib.org` and `http://www.satlive.org`.

the input/output relationship, just as we did for a gate-level or RTL model.

Symbolic simulation is again the solution. For straight-line code, symbolic simulation is easy, since an assignment statement can be thought of as a gate, with the left-hand side being the output. At each point in the program, we keep track of the current symbolic expression for the value of each variable, and we use these to compute symbolic expressions for any computation that is done. For example, given the sequence of statements:

```
a = b + c;
d = d + a;
```

we could simulate the first statement and compute $b_0 + c_0$ as the new value for variable a, where b_0 and c_0 are the initial values of b and c. Then, for the second assignment statement, we compute the new value of d to be the sum of the current values of d and a, namely $d_0 + (b_0 + c_0)$. This simple approach can be easily extended to handle arrays, structs, logical and arithmetic operators, and even pointers (e.g., [6]).

Control flow is what distinguishes symbolic simulation of software from hardware. The fundamental difference is that in hardware, for any input, every gate output is driven to some value, whereas in software, a given statement might execute once, many times, or not at all, depending on branch and loop conditions that affect the control flow. Seen another way, each possible execution path through the software model produces a different symbolic simulation. Accordingly, we modify the symbolic simulation algorithm to track the conditions under which the current execution path will execute. This can be done by replacing the symbolic expressions with pairs of symbolic expressions: the first is the same symbolic expression as we have described already, the second is a Boolean-valued expression that indicates under what conditions the first expression is valid. Whenever the symbolic simulation reaches a branch, it must continue to explore along both paths, recording for each path the branch condition that was assumed.

For example, consider the simple software model and corresponding gate-level model in Figs. 4 and 5. Let us consider symbolically simulating the path where the if condition always evaluates to true. At first, the variables i and count are both 0. After one pass through the loop body, count gets incremented, but only under the assumption that key = data[0], so it would have the symbolic value:

$$0 + 1 \quad \textit{if} \quad \texttt{key} = \texttt{data}[0].$$

After another iteration, it would have the symbolic value:

$$0 + 1 + 1 \quad \textit{if} \quad (\texttt{key} = \texttt{data}[0]) \wedge (\texttt{key} = \texttt{data}[1]),$$

and so forth. If we symbolically simulate all possible paths through the software model, and prove equivalence to the hardware model for each path, then we have proven equivalence. Unfortunately, even this tiny example has 128 paths to analyze, ignoring the loop tests. With a more complicated loop structure, we would also have to consider at each iteration both the path when the loop continues and the path when it exits.

```
int count_matches(int key, int data[7])
{
  int i, count = 0;
  for (i=0; i<7; i++) {
    if (key==data[i]) count++;
  }
  return count;
}
```

Fig. 4. High-Level Software Model Example. The model is “combinational”: given inputs, it computes a result. It can have local variables, but cannot retain any state.

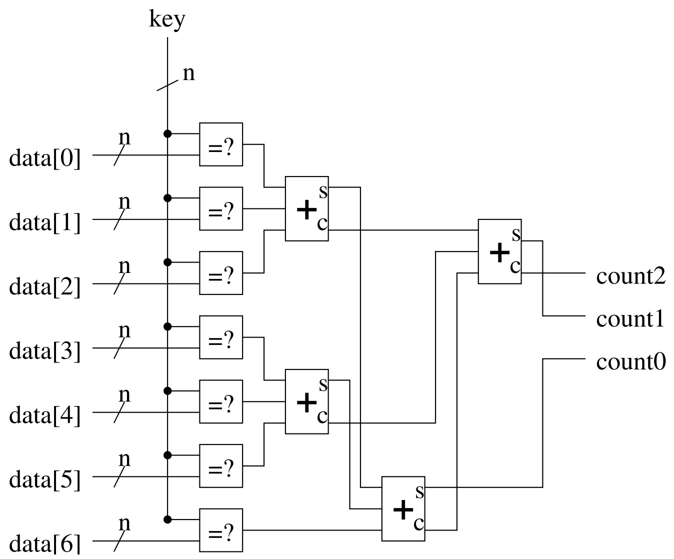

Fig. 5. Hardware Model for Same Example. The comparators are each n-bits wide. The full adders have sum and carry outputs labeled s and c, respectively. Does this circuit compute the same function as Fig. 4?

(In simple loops as in this example, the symbolic simulator can compute the exact value of the loop variable, so it knows whether to continue the loop or not.) In general, the number of paths grows too large, so symbolically simulating one path at a time is not feasible on real verification problems.

Instead, we can try to merge execution paths, using conditional expressions to keep track of the different values on different paths. For example, after one iteration of the loop, we can merge both paths from the first if condition, yielding the symbolic expression for count:

$$\text{ite}(\texttt{key} = \texttt{data}[0], 1, 0)$$

where ite is the if-then-else operator. After another iteration, count would have the symbolic expression:

$$\begin{aligned}&\text{ite}(\texttt{key} = \texttt{data}[1],\\ &\quad \text{ite}(\texttt{key} = \texttt{data}[0], 2, 1),\\ &\quad \text{ite}(\texttt{key} = \texttt{data}[0], 1, 0)\\ &),\end{aligned}$$

and so forth. Unfortunately, now our symbolic expressions are growing exponentially. By merging paths, we have shifted the

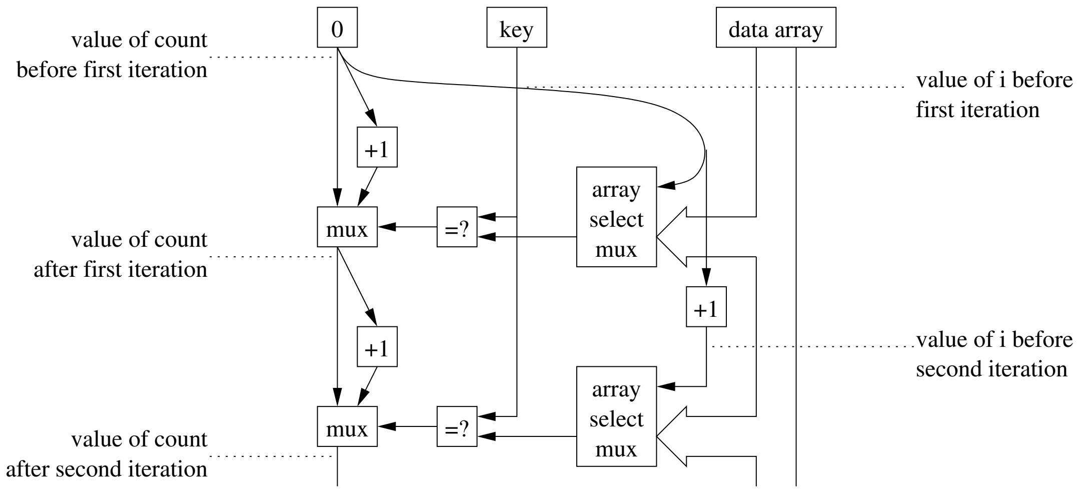

Fig. 6. Symbolic Simulation Using Circuit Graph Representation. The value of a variable at any point during the symbolic simulation is a pointer into the circuit graph. New values are computed by adding gates connected to the existing gates that denote the current values of variables. For example, to increment `count`, we would add a "+1" gate to the circuit, route the output of the current value for `count` to the new gate, and update the pointer for `count` to point to the output of the new gate. (To improve readability, the graph drawn hasn't had all structurally equivalent gates removed.)

computational blow-up from the number of paths to the size of the symbolic expressions.

Many techniques have been proposed to avoid this blow-up. For example, as we saw in gate-level equivalence checking, we can introduce fresh variable names for the result of each assignment along an execution path, reducing expression size blow-up (e.g., [2]) at the expense of much greater time verifying equivalence (e.g., [8]). Heuristic, local simplification of sub-expressions can be highly effective in some applications [7], [8]. Perhaps the most elegant solution, though, is to use a maximally shared circuit graph to represent the symbolic expressions [13]. In this approach, the "symbolic expression" is simply a pointer into a circuit that is built gradually during symbolic simulation. To symbolically simulate a computation, a new "gate" is added to the circuit, and wires are added to supply the arguments from the current value of the variables. Fig. 6 illustrates this process for a few iterations of our running example (Fig. 4). To keep the graph compact, before any new gate is added, a hash table is consulted to see if a structurally equivalent (same inputs and same operator) gate exists already; if so, the existing gate is reused.

Symbolic simulation with maximally shared circuit graphs essentially reduces the high-level-vs.-RTL equivalence checking problem to a gate-level problem. At that point, we can try to use our existing RTL- and gate-level techniques, mentioned earlier. For high-level models that are not *too* high-level, this approach works very well. As the software model becomes more complex or more high-level, however, several problems emerge. The first problem is that the circuit graph is not conducive for quickly proving outputs to be valid (stuck-at-1) or unsatisfiable (stuck-at-0). This means that if the software model has loops that are more complex than in our simple example, it is very hard to compute when the symbolic simulation can stop. On the other hand, if we try to convert the logic for loop tests into BDDs or SAT, we will have a blow-up in memory or run time. Similarly, it can be expensive to compute the logical conditions that determine what values are possible at which points in the program, e.g., to show that certain paths are infeasible. Finally, since the circuit graph will grow with the size of the software model *with loops unrolled*, it can still blow up in size before we can apply gate-level cutpoints. Accordingly, the next step is to try to apply the idea of cutpoints directly to the software model, before constructing an entire circuit graph from the software.

IV. Cutpoints for Software

In recent work, we have proposed a way to introduce cutpoints *during* the analysis of the software, rather than afterwards [9]. A cutpoint in the software is defined as some part of the program state at some point during the symbolic simulation that is provably equal to some point in the hardware model (definition adapted from [8]). If we find a cutpoint, we can cut out the corresponding parts of the software and hardware models and insert a new input in its place, exactly as in RTL or gate-level cutpoints. Successful verification with the cutpoints proves the two models equivalent.

For example, returning to the models in Figs. 4 and 5, when the symbolic simulation reaches the `if` condition on the first iteration $\texttt{key} = \texttt{data}[0]$, this condition is provably equivalent to the output of the topmost comparator in Fig. 5. Accordingly, we could replace the comparison with a fresh Boolean variable/input `x0`, as shown in Figs. 7 and 8. Continuing this process will replace all of the comparators in the hardware model with new cutpoints `xi`. The software model is processed into a circuit graph representation, as in Fig. 6. Combined with the cutpoint insertion, the result is the circuit in Fig. 9. Clearly, cutpoints can potentially greatly simplify the equivalence-checking problem.

As a more realistic test, we tried the cutpoint technique on an industrial challenge problem: verifying the equivalence of a

```
int count_matches(..., int x0, ...)
{
  int i, count = 0;
  if (x0) count++;
  for (i=1; i<7; i++) {
    if (key==data[i]) count++;
  }
  return count;
}
```

Fig. 7. Software Model After First Cutpoint. An actual implementation wouldn't rewrite the software as shown here, but simply insert the cutpoint `x0` in the circuit graph model being constructed.

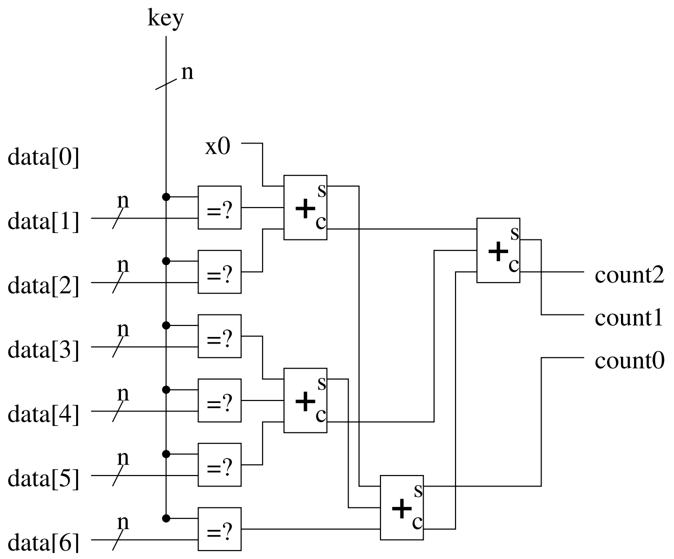

Fig. 8. Hardware Model After First Cutpoint. The equivalent comparator has been removed and replaced by a new input `x0`, as in Fig. 7.

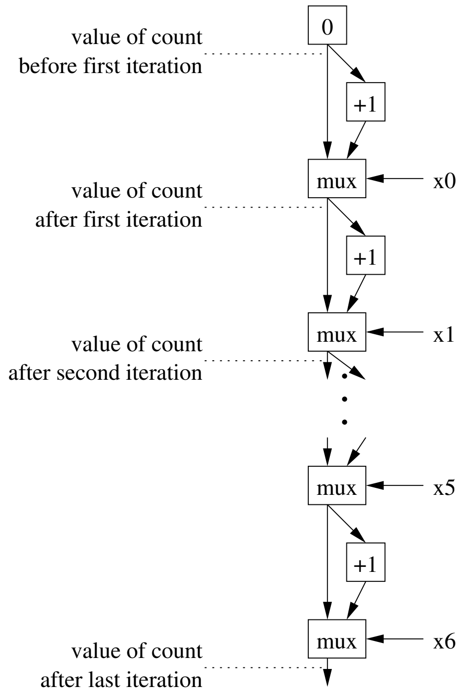
value of count before first iteration
0
+1
mux
x0
value of count after first iteration
x1
value of count after second iteration
x5
x6
value of count after last iteration

Fig. 9. Circuit Graph Generated From Software Model With Cutpoints. All the comparator logic has been cut away and replaced by new inputs `xi`.

high-level specification versus a gate-level implementation of an Intel IA-32 instruction-length decoder. The software model has complex control flow, with many branches and loops whose iteration count and stride depends on the input data. It is completely serial, essentially parsing each instruction one after another. The hardware model has very different architecture, simultaneously attempting to decode an instruction at each possible alignment in the input buffer, and then using a priority network to determine which decodings are valid. Fig. 10 shows results comparing verification with and without cutpoints. (Details of this experiment are available in [9].)

While the promise of software cutpoints is clear, there are many remaining challenges. The biggest problem is how to find good candidate cutpoints. As with RTL and gate-level cutpoints, a heuristic search for similar structure still works and was the technique used in the preceding experiment. Unfortunately, the main technique for RTL and gate-level cutpoints — random simulation to generate signatures — doesn't work for software cutpoints. The problem is that a given random input causes only a single path in the software to be executed. Since there is an exponential number of paths

	w/o Cutpoints		with Cutpoints	
Example	Time	Mem	Time	Mem
TOY-8	0.02s	56MB	0.01s	56MB
TOY-16	5.35s	56MB	0.02s	56MB
TOY-32		mem out	0.06s	56MB
EX20-8	0.28s	61MB	0.11s	58MB
EX20-16	89.01s	1746MB	0.24s	60MB
EX20-32		mem out	0.53s	64MB
EX20-64		mem out	1.35s	72MB
EX97-8	1.46s	92MB	0.51s	64MB
EX97-16	1187.72s	1800MB	1.10s	73MB
EX97-32		mem out	2.35s	95MB
EX97-64		mem out	5.41s	136MB
EX251-12	309.18s	1843MB	0.64s	66MB
EX251-16		mem out	1.09s	71MB
EX251-32		mem out	7.45s	170MB
EX251-64		mem out	16.81s	327MB

Fig. 10. Sample Results Showing Effect of Cutpoints. Each example is a differently scaled version of an IA-32 instruction-length decoder. The number after the dash is the size of the input buffer. The TOY examples have only 6 highly-simplified instructions for a fictitious machine with 2-bit "bytes". The other examples are for subsets of actual IA-32 instructions, with the number indicating how many instructions are implemented. The largest examples support instruction lengths from 1-11 bytes with a wide range of prefixes and addressing modes. The benefits of cutpoints are clear. (Results are from [9].)

(or more, if we consider looping), computing good signatures requires a prohibitively large number of runs. Good heuristics for finding candidate software cutpoints is an important direction for future research.

V. Future Directions

We have seen a brief survey of the main ideas behind high-level-vs.-RTL combinational equivalence verification. This introductory treatment does not imply that this is all there is on this topic. On the contrary, much additional research has been done, and much more remains to be done.

As mentioned already, heuristics for finding good candidate software cutpoints is an important direction for future research. One possibility is to use random simulation, but bias the random simulation to explore more paths and find a way to extract signature information for paths not taken. Another possibility is to use program analysis and optimization techniques to try to derive more information about the software model, and use that information to screen possible cutpoints.

More generally, program analysis and optimization techniques might improve other aspects of the symbolic simulation. For example, a major mismatch between combinational software models and hardware models is the amount of parallelism: the software models tend to be very serial, for ease of understanding, whereas the hardware models are parallel for performance. Researchers in high-level synthesis and optimizing compilers have long worked on extracting parallelism from sequential descriptions, so techniques from those areas may help expose similarities between the high-level software and RTL hardware models.

More sophisticated program analysis techniques will likely be needed for another reason: handling richer control flow and data structures. The techniques presented here work well for moderately complex software. More generally, however, software that has loops or recursion that can't be statically unrolled, or complex heap-allocated storage, cannot be handled currently. Harnessing existing, or developing new, specialized techniques for summarizing loops and recursive functions and for reasoning about heap storage might greatly expand the range of high-level software models that can be verified.

This article has largely ignored the problem of false inequivalence, because nothing has been published specifically for resolving false inequivalence for high-level-vs.-RTL. Obviously, all the techniques for RTL-vs.-gate-level false inequivalence handling are still applicable, but there might be software-specific techniques as well.

Acknowledgments

This work was supported in part by grants from Intel Corporation and the Natural Science and Engineering Research Council of Canada. Thanks to the organizers of ICCD 2006 for the opportunity to prepare this introductory paper.

References

[1] C. L. Berman and L. H. Trevillyan, "Functional comparison of logic designs for VLSI circuits," in *International Conference on Computer-Aided Design*. IEEE, 1989, pp. 456–459.

[2] C. Blank, H. Eveking, J. Levihn, and G. Ritter, "Symbolic simulation techniques — state-of-the-art and applications," in *International Workshop on High-Level Design, Validation, and Test*. IEEE, 2001, pp. 45–50.

[3] D. Brand, "Verification of large synthesized designs," in *International Conference on Computer-Aided Design*. IEEE/ACM, 1993, pp. 534–537.

[4] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, vol. C-35, no. 8, pp. 677–691, August 1986.

[5] ——, "A methodology for hardware verification based on logic simulation," *Journal of the ACM*, vol. 38, no. 2, pp. 299–328, April 1991.

[6] E. Clarke and D. Kroening, "Behavior consistency of C and Verilog programs using bounded model checking," Carnegie Mellon University, Tech. Rep. CMU-CS-03-126, May 2003.

[7] D. W. Currie, A. J. Hu, S. Rajan, and M. Fujita, "Automatic formal verification of DSP software," in *37th Design Automation Conference*. ACM/IEEE, 2000, pp. 130–135.

[8] X. Feng and A. J. Hu, "Cutpoints for formal equivalence verification of embedded software," in *5th International Conference on Embedded Software*. ACM, 2005, pp. 307–316.

[9] ——, "Early cutpoint insertion for high-level software vs. RTL formal combinational equivalence verification," in *43rd Design Automation Conference*. ACM/IEEE, 2006, pp. 1063–1068.

[10] S.-Y. Huang and K.-T. Cheng, *Formal Equivalence Checking and Design Debugging*. Kluwer Academic Publishers, 1998.

[11] J. Jain, A. Narayan, M. Fujita, and A. Sangiovanni-Vincentelli, "Formal verification of combinational circuits," in *International Conference on VLSI Design*, 1997.

[12] A. Koelbl, Y. Lu, and A. Mathur, "Embedded tutorial: Formal equivalence checking between system-level models and RTL," in *International Conference on Computer-Aided Design*. IEEE/ACM, 2005, pp. 965–971.

[13] A. Koelbl and C. Pixley, "Constructing efficient formal models from high-level descriptions using symbolic simuluation," *International Journal of Parallel Programming*, vol. 33, no. 6, pp. 645–666, December 2005.

[14] A. Kuehlmann and F. Krohm, "Equivalence checking using cuts and heaps," in *34th Design Automation Conference*. ACM/IEEE, 1997, pp. 263–268.