Early Cutpoint Insertion for High-Level Software vs. RTL Formal Combinational Equivalence Verification *

Xiushan Feng Alan J. Hu
Department of Computer Science, University of British Columbia
{xsfeng, ajh}@cs.ubc.ca

ABSTRACT

Ever-growing complexity is forcing design to move above RTL. For example, golden functional models are being written as clearly as possible in software and not optimized or intended for synthesis. Thus, equivalence verification between the high-level software functional model and the RTL is needed. The typical approach is to convert the high-level software into RTL or gate-level hardware, via software path enumeration, symbolic execution, or high-level synthesis techniques, and then use hardware combinational equivalence checking. The principle contribution of this paper is to introduce cutpoints — as in gate-level combinational equivalence verification — early during the analysis of the software model, thereby avoiding exponential path enumeration and the potential logical complexity blow-up of merging execution paths that can occur in the usual approach. The method is conservative, but in our experiments, we did not encounter spurious counterexamples, and the method showed large improvements in runtime and memory usage on a family of IA-32 subset instruction length decoders, an industry-suggested challenge problem.

Categories and Subject Descriptors: J.6 [Computer-Aided Engineering]: Computer-Aided Design

General Terms: Verification

Keywords: software, RTL, formal equivalence checking, cutpoints

1. 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. Koelbl et al. provide a recent tutorial overview of this area [14].

*This work was supported in part by research grants from Intel Corporation and the Natural Sciences and Engineering Research Council of Canada.

DAC 2006, July 24–28, 2006, San Francisco, California, USA.

Our focus is on high-level, functional models for verification. We have observed that several companies have adopted a methodology for complex designs that starts with a golden functional model written in C. This model is meant to be the definitive specification of correct functional behavior and isn't meant for synthesis. Accordingly, it is written as clearly and simply as possible. Optimization for synthesis occurs in lower-level software or RTL models.

Verifying equivalence between the high-level and RTL models is an extremely hard problem, so we further focus on the simplest version of this problem: the case where the software model is pin- and cycle-accurate, making this a combinational equivalence problem. This simplified problem is directly useful for verifying highly complex combinational circuits, and is also a necessary building block for high-level-vs.-RTL sequential equivalence verification. Even in the combinational case, the problem is challenging because the clearest way to express a functionality in software is generally not the most efficient way to implement it in hardware. For example, difficult verification problems arise in superscalar processor designs, where complex sequences of computations are executed in a single cycle on highly parallel hardware. The correct functionality is easy to describe sequentially in software, but the correspondence to the hardware is not at all obvious.

The direct approach to this verification problem is first to convert the high-level software into RTL or gate-level hardware, and then to leverage highly successful techniques from RTL or gate-level combinational equivalence verification — in particular, the introduction of cutpoints — to verify the equivalence of the two low-level models. Unfortunately, as Gupta et al. [8] point out, most high-level synthesis has targeted creating multicycle, resource-constrained designs, rather than creating single-cycle hardware from software models with complex control flow, so off-the-shelf high-level synthesis isn't applicable to our problem. With symbolic simulation, the challenge is complexity blow-up. To handle the control flow of software, symbolic simulation explores all paths through the code. If this exploration is done explicitly, execution time can blow up with the exponential number of paths. If paths are merged to avoid this blow-up, then there is a potential blow-up in the logic that tracks the different results that are computed on the different paths.

The principle contribution of this paper is a novel way to introduce cutpoints *early*, during the analysis of the software model, rather than after a low-level hardware-equivalent has been generated. By doing so, we avoid the exponential enumeration of software paths as well as the logic blow-up of tracking merged paths. We evaluate our method on a challenge problem suggested to us by colleagues in industry: a family of instruction length decoders for varying subsets of the IA-32 instruction set architecture (described in Section 2). Experimental results show large improvements in runtime and memory usage due to our early insertion of cutpoints.

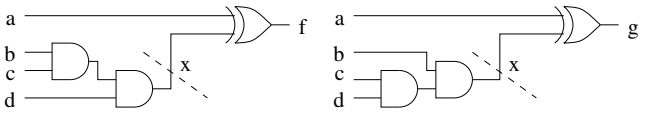


Figure 1: Simple Cutpoint Example. To introduce cutpoint x, we first verify that $(b \wedge c) \wedge d$ is equivalent to $b \wedge (c \wedge d)$. Then, we can verify that f is equivalent to g because both are equal to $a \oplus x$.

1.1 Related Work

Most work in this area leverages the success of RTL and gate-level combinational equivalence checking, and we assume familiarity with these concepts. Some excellent surveys of this material are available, e.g., [12, 10]. A key technique behind this success is the idea of cutpoints [1, 2]: Since the two combinational circuits are presumed to be structurally similar, there should be intermediate points in the two circuits that are logically equivalent. Heuristics search for such possibly equivalent intermediate points, and the tool first tries to prove such points equivalent. If successful, the equivalent logic is cut out of the circuits and replaced by a new primary input, thereby simplifying the verification problem. (Figure 1.) In general, the method is conservative (i.e., success proves equivalence, but failure doesn't prove inequivalence) because constraints on the cutpoint "inputs" are lost; various techniques re-introduce constraints to reduce this problem, e.g., [2, 12, 16]. In the present paper, we also use the concept of cutpoints, but we introduce them much earlier, during the analysis of the software, rather than applying them between two hardware models.

Given the success of lower-level combinational equivalence checking, an obvious approach to our verification problem is to try high-level synthesis on the software model, to reduce the problem to RTL or gate-level. Unlike most high-level synthesis systems, Gupta et al.'s Spark system was specifically designed for the sorts of high-level models we need (highly unoptimized, complex software into single-cycle hardware) [8]. Indeed, they demonstrated the capabilities of their system on a (much simpler than ours, but with the same essential characteristics) IA-32 instruction length decoder, the same challenge problem we use. Unfortunately, the current version of Spark was not able to handle our examples, so we cannot evaluate how well this approach would work.[1] In our initial software analysis phase, we assume that certain standard program analyses (e.g., CFG construction, data flow analyses) have been done. We believe optimizations and analyses as done in Spark could enhance the initial phases of our verification flow.

With the growing importance of C models of hardware, several groups in recent years have published results that verify C against RTL hardware. For example, Séméria et al. [20] reported verifying C against Verilog as part of a C-based design flow. However, their C model was already in RTL C, so the verification aspect was straightforward. More relevant is work by Saito et al. [19] and Clarke and Kroening [5], both of which consider higher-level C models. The former work relies on scanning for textual differences to reduce problem complexity, and then enumerates execution paths and applies symbolic simulation and word-level uninterpreted functions. The latter work provides full support for arbitrary code in full ANSI-C, also via path enumeration and symbolic simulation, but at a fully bit-accurate level and using SAT as the computation engine. A limitation of both of these works is the enumeration of execution paths in the software: the number of paths grows exponentially in the number of branches.

[1] Spark v1.2, released Feb 5, 2004.

Our idea to apply cutpoints early in the analysis of the software, rather than generating a hardware model first, comes from our adaptation of cutpoints to software verification [7]. However, that work also enumerates software paths, and suffers the resulting scalability problem.

An alternative to path enumeration is to merge execution paths as much as possible, keeping track of the different path conditions and possible values in the symbolic expressions. For example, given:

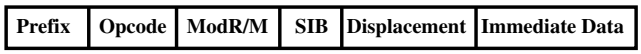
```
if (c1) x=a;
else x=b;
if (c2) x++;
else x--;
```

rather than analyzing each of the four execution paths separately, we could compute some sort of symbolic expression for `x` after the first `if` statement that merges the two branches, e.g., $\text{ite}(c1, a, b)$, and then merge again after the second `if` statement, producing $\text{ite}(c2, \text{ite}(c1, a, b)+1, \text{ite}(c1, a, b)-1)$. Merging paths converts the exponential path enumeration into logical complexity in the symbolic expressions. Early work along these lines [9, 17] suffered from BDD blow-up for non-trivial software models. Very recently, Koelbl and Pixley have proposed a more scalable approach [15]. They use an acyclic circuit representation for the symbolic expressions, greatly reducing blow-up. Furthermore, blow-up of the path conditions is reduced by a two-level representation: branching conditions in the program are abstracted as Boolean variables, and the path condition is stored as a BDD over those variables. This two-level representation allows some fast approximate reasoning, but accurate computation of path conditions is expensive, requiring a combined decision procedure for — or else flattening — the two-level representation. No implementation is publicly available, but we believe their approach is more efficient than the BDDs used in our initial software analysis phase. We also believe that our early cutpoint idea should be readily applicable to their construction, promising substantial savings in verification complexity.

2. INDUSTRIAL CHALLENGE PROBLEM

Academic research on software-to-RTL verification has been stymied by the lack of good benchmark examples. Companies are loath to give away such valuable intellectual property, and substantial engineering effort is required to create examples. Fortunately, we had a well-defined, industrial challenge problem suggested to us, which epitomizes this class of verification problem: an instruction length decoder for Intel's IA-32 instruction set architecture [11]. The circuit's functionality (described below) is conceptually simple and easy to describe in software, although the actual code is long and has complex control flow. The RTL implementation does not resemble the high-level software. We have created a public set of example software and RTL models, implementing increasingly larger and more complete decoders. (See Section 4.)

The IA-32 instruction set architecture (ISA) descends directly from the 16-bit Intel 8086/8088 through the latest Pentium processors, and this line has dominated desktop computing for over two decades and several orders of magnitude increase in processing power. Backwards software compatibility has always been important, so the ISA has grown by accretion, resulting in extremely complex instruction encodings. Instructions can range from 1 byte to over 15 byes in length. All IA-32 instruction encodings obey the format shown as Figure 2. The actual length of an IA-32 instruction depends on the operating mode (protected mode, real-address mode and system management mode), the prefix bytes (if any), the opcode byte(s), the ModR/M byte (if present), and the Scale Index Base (SIB) byte (if present). For example, in protected mode,



Prefix:	0~4 Bytes	SIB:	0~1 Byte
Opcode:	1~2 Bytes	Displacement:	0~4 Bytes
ModR/M:	0~1 Byte	Immediate:	0~4 Bytes

Figure 2: IA-32 General Instruction Format

the default operand and address sizes are both 32 bits. But the operand-size override prefix (66H) and the address-size override prefix (67h) allow a program to switch to non-default operand and addressing sizes, which are 16 bits. For some instructions, with a current operand-size attribute determined by the operating mode and operand-size override prefix (if present), the size of operands can further be changed by the operand size bit (w) of the opcode. If w is 0, the operand size is 1 byte regardless of the current operand-size attribute. If w is 1, it has no change on the current operand size. In addition, if there is a ModR/M byte field in an instruction, the 256 values of ModR/M will define different addressing forms which will affect the length of displacement field and decide whether there is an SIB byte to follow. The addressing forms are different for different addressing modes (16-bit or 32-bit).

Because of the complex instruction encoding, a high-performance IA-32 implementation must pipeline instruction decoding. A piece of this puzzle is the instruction length decoder (ILD). (Our description is based on [13].) Each cycle, the ILD is given an n-byte *parcel* of the next bytes in the instruction stream, enough additional lookahead bytes in the instruction stream to determine the end of the last instruction in the current parcel (which can extend into the next parcel), and a *wrap pointer* that indicates how far the last instruction from the previous parcel extends into this parcel. The output is two n-bit vectors `begin` and `end`, which indicate the beginning and end of each instruction in the parcel, and the wrap pointer for the next parcel.

A high-level software model of the ILD is straightforward, if a bit convoluted. The software simply starts at the wrap pointer and scans the parcel byte-by-byte, parsing each instruction one at a time. Figure 3 gives pseudocode for the main loop of the software ILD. The loop body is basically a simple syntax-directed parser for the instruction format, with the various functions communicating via global variables, such as `wrap` pointing to the current location in the parcel or the lookahead bytes, `current_byte` being the content of that location, and the `_mode` variables remembering the current operand and address sizes. The `handle_` helper functions consist of nested `if` statements that continue the parsing for as many bytes as needed into the details of the instruction format. Note that the loop body can end with `wrap` being incremented by many different values, from 1 to the longest instruction length, depending in a complex manner on the input parcel.

The RTL model is very different. For performance reasons, all decoding must be done in parallel. One can easily imagine the logic required to decode the length of an instruction starting at a fixed location. This logic is replicated for each possible instruction alignment in the parcel, speculatively computing the length assuming that an instruction starts there. A priority-encoding network determines which blocks of instruction-length logic are the valid ones: the length computed at the input `wrap` position is valid, and the length starting at a position j is valid iff the length starting at a position i is valid and $j = i + length\_from(i)$.

The verification task is to prove the two models compute the same output `begin` and `end` vectors, for all possible input parcels and wrap values. Given the large difference between software specification and hardware implementation, verifying functional equivalence is a challenge.

```
while (wrap < PARCEL_SIZE) {
  begin[wrap]=1; /* Start of instruction */

  /* Set default sizes. */
  operand_mode = INIT_OPERAND_MODE;
  address_mode = INIT_ADDRESS_MODE;

  get_next_byte();
  ret = handle_prefixes();
  /* If there were any prefixes,
     get the next byte for opcode. */
  if (ret) get_next_byte();

  if (current_byte != ESCAPE) {
    handle_one_byte_opcodes();
  } else { /* Escape to two-byte opcode */
    /* Skip over the escape code. */
    get_next_byte();
    handle_two_byte_opcodes();
  }
}
```

Figure 3: Software Model Main Loop Pseudocode. The software model parses instructions one byte at a time, updating `wrap` depending on the parsed instruction.

3. VERIFICATION ALGORITHM

The verification algorithm we have implemented compares a high-level model, given as an annotated control-flow graph (CFG), to a gate-level model, given in BLIF. The translation from a programming language like C or C++ to a CFG is well-known; starting from the CFG keeps our tool language-neutral and saves considerable front-end implementation effort. We also assume that functions have been inlined, and a simple, intraprocedural dataflow analysis has been done. Algorithms for these steps are available in any compiler reference, e.g., [18].

Proceeding from the annotated CFG, the verification algorithm has two main phases: a preliminary analysis of the software to unroll loops, merging paths where possible; and the actual formal equivalence check, where we try to insert cutpoint during the processing of the unrolled CFG.

3.1 Preliminary Software Analysis

Because we are treating the software as "combinational," we perform a preliminary pass to unroll loops and obtain an acyclic CFG. Simple loops, e.g., with constant bounds and increments, can be unrolled in the obvious manner (cf. [15]). For more complex loops, such as the main instruction length decoder loop (Figure 3), more sophisticated techniques are needed, as in advanced compiler/synthesis optimization (cf. [8]). In our method, the dataflow analysis tells us that `wrap` is the only loop-carried dependence. Therefore, we know that `wrap` is the only information that must be tracked in distinguishing different iterations of the loop body. Accordingly, as we unroll the loop, any two paths that (re)enter the loop with the same value of `wrap` can be merged. The net result is that the tool automatically unrolls the original CFG into the graph shown in Figure 4.

There are two key points about this preliminary phase. First is that the construction, and the resulting graph, are linear in the size of the software (with loops unrolled). There is no exponential blow-

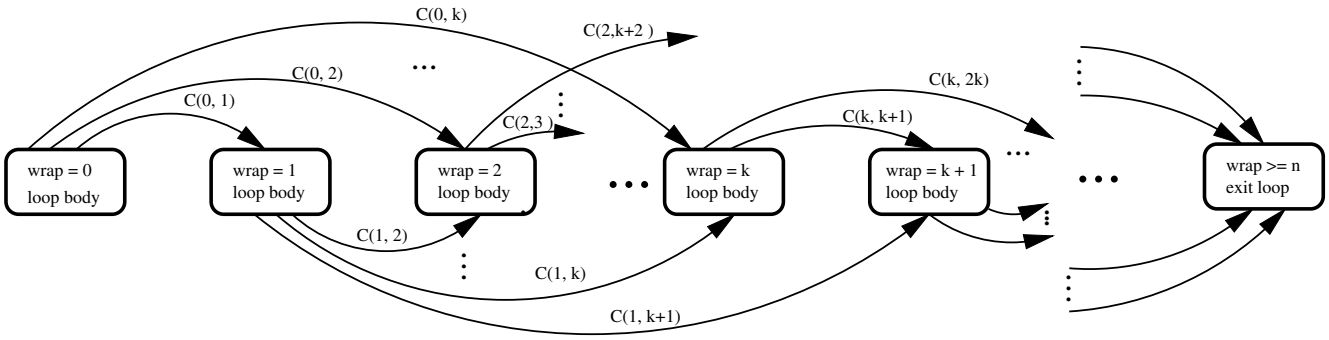


Figure 4: Unrolled and Merged Control Flow Graph. $C(i,j)$ denotes the logical condition such that the loop iteration with `wrap` $= i$ will continue to the loop iteration with `wrap` $= j$. For clarity, we haven't drawn the graph edges and vertices inside the loop bodies; the actual CFG, of course, does have those details.

up of path enumeration, because the algorithm explores each *edge* once, not each path. The second key point is that we haven't performed a full symbolic simulation yet. Symbolic simulation would compute expressions giving the values of all the variables of the circuit, e.g., the `begin` and `end` vectors. These outputs *do* depend on the *path* taken to reach a given point in the graph. For example, whether the kth byte in a parcel is the start of an instruction depends on where the previous valid instruction ended. Thus, symbolic simulation requires computing, for each variable at each point in the unrolled graph, a logical expression that gives the correct value depending on the path taken to arrive there. These expressions are liable to blow-up.

3.2 Formal Equivalence Check

We now proceed to the main phase of the verification algorithm. To formally verify equivalence between the software and hardware models, the algorithm must derive some representation of the function computed at the output of each model. For both gate-level hardware or an execution path in software, symbolic simulation is the standard method to derive these representations. We use BDDs [4] to represent the functions in symbolic simulation: they are reasonably efficient, and canonicity makes proving equivalence and finding cutpoints easy.

As mentioned above, the value of a variable at a given point in the program depends on the path taken through the program to reach that point. A direct approach is to enumerate all paths through the software and verify equivalence for each path individually. Unfortunately, the number of paths can be exponential.

Alternatively, to avoid enumerating all the paths, we can merge paths during symbolic simulation, using conditional expressions to track the different values possible along different paths. Path merging, therefore, requires computing what input conditions will cause a given point in the program to execute (and affect the variables). For example, in Figure 4, the software code in loop body k will affect the values of the variables iff the execution starting from the initial value of `wrap` followed a path that eventually reached loop body k, i.e., there is some sequence of values $v_0, \ldots, v_l$, where v_0 equals the initial value of `wrap` when the code starts, $v_l = k$, and for all i, the edge conditions $C(v_i, v_{i+1})$ are all true. (And the conditions for control flow within the loop bodies, not drawn in Figure 4, have to be true as well.)

Fortunately, we can compute these conditions in a number of BDD operations linear in the graph size. The algorithm works as follows: Let $N(k)$ denote the *node condition* for basic block k, i.e., the logical expression that indicates what inputs will cause the software to execute basic block k. We can compute $N(k)$ recursively: $N(k) = \bigvee_i (N(i) \wedge C(i,k))$, i.e., for basic block k to execute, it must be true that some basic block i executed and then the branching condition $C(i,k)$ that control flowed from i to k was also true. Because the unrolled CFG is acyclic, this computation examines each edge exactly once, yielding the linear complexity. This linear construction is a state-of-the-art symbolic simulation approach to converting the software model into BDDs or some other function representation. We are now ready to consider early cutpoint insertion.

The key idea of early cutpoint insertion is to look for cutpoints *during* the above computation of $N(k)$ rather than after it is completed. If we find some $N(i)$ (or any other BDD during symbolic simulation) that is equal to some point in the gate-level circuit, we can cut out the equivalent logic in the software and the hardware, and introduce a new primary input in its place. This eliminates the complexity of the logic for $N(i)$ in subsequent computations. As with gate-level cutpoints, if we can continue this process to the outputs of the software and hardware models, then we have formally verified equivalence.

For example, Figure 5 shows some cutpoints added to Figure 4. If we prove that the node condition for loop body 0 is the same as some logic in the gate-level circuit, we can cut out the logic and introduce a new primary input x_0 at the cut. Repeating the process introduces cuts at x_1, x_2, etc. With cutpoints, the logic for node condition $N(k)$ is simplified from $\bigvee_i (N(i) \wedge C(i,k))$ to:

$$\bigvee_i (x_i \wedge C(i,k)).$$

Like most cutpoint methods, this approach is conservative: introducing cutpoints loses information and may result in being unable to prove equivalent models equivalent. In the other direction, cutpoints won't erroneously prove inequivalent models equivalent, but the algorithm may not find enough cutpoints to reduce verification complexity. Either failing is a theoretical possibility; the only way to evaluate practical usefulness is via experiments.

4. EXPERIMENTAL RESULTS

We have run our experiments using 4 different versions of the instruction length decoder.[2]

TOY is an example taken from [13] to describe what an instruction length decoder looks like. This toy example has only

[2]Examples are available at `http://www.cs.ubc.ca/~ajh`.

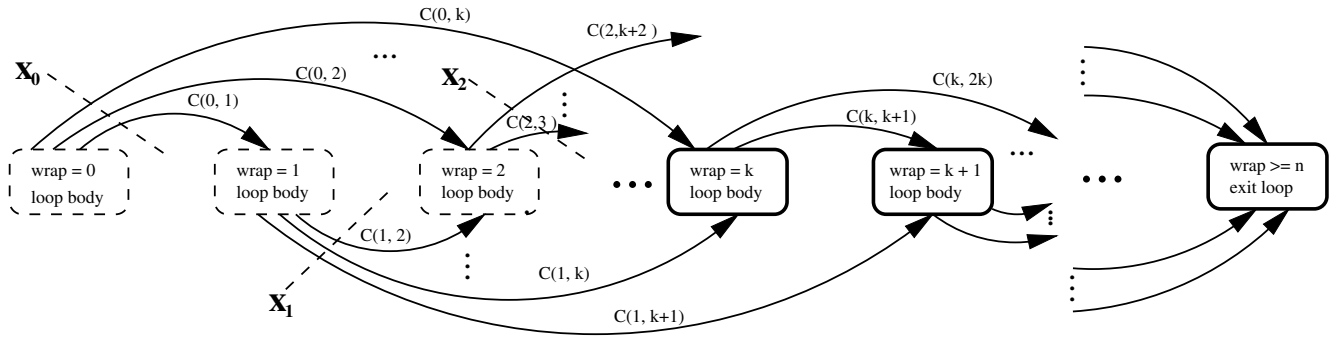


Figure 5: Early Cutpoint Insertion. Possible cutpoints, like x_0, etc. are checked against the hardware model and cutpoints are inserted during the analysis of the software, not after.

Example	Size (4-LUTs)
TOY-8	138
TOY-16	331
TOY-32	723
EX20-8	467
EX20-16	912
EX20-32	2251
EX20-64	9012

Example	Size (4-LUTs)
EX97-8	1637
EX97-16	3255
EX97-32	6448
EX97-64	17540
EX251-12	6199
EX251-16	8312
EX251-32	16770
EX251-64	131002

Table 1: Circuit Sizes of Examples.

6 instructions: 3 one-byte opcode instructions, and 3 two-byte opcode instructions. Furthermore, the size of a "byte" in TOY is only 2 bits, and there are no prefixes, ModR/M, and displacement bytes. The two-byte opcode instruction format consists of an escape opcode byte followed by a second opcode byte. For this example, we have 3 versions, with parcel sizes of 8, 16 and 32 "bytes".

EX20 has 20 IA-32 instructions with lengths from 1 to 6 (8-bit) bytes. It includes three forms of instructions. Form one is a simple one-byte opcode instruction form without ModR/M, immediate, and displacement fields. Form two is a simple two-byte opcode instruction form without ModR/M, immediate, and displacement field. Form three is a one-byte opcode instruction with "w" bit in its opcode and with immediate data, but without ModR/M and displacement fields.

EX97 has 97 instructions (lengths from 1–8 bytes). It includes all the forms of EX20 and a new form that has a one-byte opcode instruction and ModR/M byte field.

EX251 has 251 instructions (lengths between 1–11 bytes). It includes all forms of EX97 plus a form that allows immediate data after the ModR/M byte field.

All IA-32 examples allow operand-size override and address-size override prefixes. We have 4 different parcel sizes for each of the IA-32 examples: 8 or 12, 16, 32, and 64 bytes. (The parcel size must be larger than the longest instruction, so the smallest version of EX251 uses a 12-byte parcel.)

The high-level software is given in C code, then manually translated into our CFG intermediate format. The hardware model is given in Verilog. We use VIS [3] to translate it into BLIF and then SIS [21] with script.rugged to do optimization. As a rough indicator of complexity, we have also mapped the optimized circuits into 4-input lookup tables using Flowmap/Flowpack [6] (Table 1).

	Path Enumeration		Linear BDD	
Example	Time(s)	Mem(MB)	Time(s)	Mem(MB)
TOY-8	2.25	57	0.02	56
TOY-16	time out		5.35	56
TOY-32	time out			mem out
EX20-8	241.24	28	0.28	61
EX20-16	time out		89.01	1746
EX20-32	time out			mem out
EX20-64	time out			mem out
EX97-8	4229.44	183	1.46	92
EX97-16	time out		1187.72	1800
EX97-32	time out			mem out
EX97-64	time out			mem out
EX251-12	time out		309.18	1843
EX251-16	time out			mem out
EX251-32	time out			mem out
EX251-64	time out			mem out

Table 2: Path Enumeration vs. Linear BDD.

All experiments were on a 2.6Ghz Pentium 4 with 4GB of RAM. The runtime limit was 2 hours, and the memory usage limit was 2GB. Memory usage is the peak as reported by `top`. For BDD experiments, memory usage varies depending on machine memory size, because the CUDD package [22] aggressively pre-allocates memory. However, runs with different memory sizes produced the same comparative results. All times are for the full verification. Verification was successful on all examples (i.e., no spurious counterexamples), showing that the cutpoints were not too conservative.

Avoiding Path Enumeration: As mentioned earlier, we can always enumerate execution paths in the software model, and prove the equivalence for each path: under the same path condition and same inputs, prove that the software and hardware models have equivalent outputs.

The first experiment is to compare path enumeration with the linear BDD construction from Section 3.2. This measures the effect of path merging in eliminating the exponential path enumeration at the cost of a possible expression-size blow-up.

Table 2 gives the results. We can see that the execution time of path enumeration blows up quickly. The linear-time BDD building runs much faster by avoiding explicit exploration the paths.

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

Table 3: Linear BDD vs. Early Cutpoint

	hw-CBMC		Early Cutpoint	
Example	Time(s)	Mem(MB)	Time(s)	Mem(MB)
TOY-8	6.84	38	0.01	56
TOY-16	502.59	522	0.02	56
TOY-32	time out		0.06	56

Table 4: hw-CBMC vs. Early Cutpoints

Effect of Early Cutpoints: Next, we examine the effect of inserting cutpoints early. Table 3 compares our verification tool using the linear BDD construction without and with early cutpoint insertion. We see that the early cutpoint method vastly reduces both memory usage and run time.

Comparison to Other Tools: We know of only one freely available software-to-RTL verification tool that can handle the software complexity of our challenge problem: hw-CBMC.[3] As mentioned earlier, hw-CBMC does path enumeration [5], so it can handle only the smaller instances of our TOY example (Table 4). This is not a fair comparison, since hw-CBMC parses arbitrary ANSI-C, whereas we start from a CFG and exploit some assumptions about program structure. Nevertheless, the benefit of early cutpoint insertion and not enumerating paths is clear.

5. CONCLUSION AND FUTURE WORK

We have developed a novel way to introduce cutpoints early, during the analysis of the software model, to reduce the complexity of software-to-RTL equivalence verification. Experimental results on an industry-suggested challenge problem show large improvements in runtime and memory usage.

Early cutpoint insertion improves one point in the overall equivalence verification flow. Important future work will be to combine our contribution with the best ideas for other parts of this flow, e.g., preliminary textual pruning [19], a full-fledged software front-end [5], powerful software analyses and optimizations [8], and more general and efficient symbolic representations [15]. Industrial-strength high-level-to-RTL equivalence verification will require many advances; early cutpoint insertion is one.

[3] Version 1.6 from http://www.cs.cmu.edu/~modelcheck/cbmc.

Acknowledgments

We would like to thank Robert Jones of Intel Corporation for suggesting the IA-32 instruction length decoder example, and Rajesh Gupta and Sudipta Kundu of UC San Diego for assistance in obtaining and using the Spark system and examples.

6. REFERENCES

[1] C. L. Berman and L. H. Trevillyan. Functional comparison of logic designs for VLSI circuits. *ICCAD*, 1989, pp. 456–459.

[2] D. Brand. Verification of large synthesized designs. *ICCAD*, 1993, pp. 534–537.

[3] R. K. Brayton, G. D. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S.-T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. K. Ranjan, S. Sarwary, T. R. Shiple, G. Swamy, and T. Villa. VIS: A system for verification and synthesis. *Computer-Aided Verification: 8th Intl Conf*, 1996, pp. 428–432. LNCS 1102.

[4] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans Computers*, C-35(8):677–691, Aug 1986.

[5] E. Clarke and D. Kroening. Hardware verification using ANSI-C programs as a reference. *ASPDAC*, 2003, pp. 308–311.

[6] J. Cong and Y. Ding. FlowMap: An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs. *IEEE Trans CAD*, 13(1):1–12, Jan 1994.

[7] X. Feng and A. J. Hu. Cutpoints for formal equivalence verification of embedded software. *5th Intl Conf on Embedded Software*, 2005, pp. 307–316.

[8] S. Gupta, T. Kam, M. Kishinevsky, S. Rotem, N. Savoiu, N. Dutt, R. Gupta, and A. Nicolau. Coordinated transformations for high-level synthesis of high performance microprocessor blocks. *39th DAC*, 2002, pp. 898–903.

[9] A. J. Hu, D. L. Dill, A. J. Drexler, and C. H. Yang. Higher-level specification and verification with BDDs. *Computer-Aided Verification: 4th Intl Workshop*, 1992. LNCS 663.

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

[11] Intel Corporation. *The IA-32 Intel Architecture Software Developer's Manual*, 2004. Four volumes. Intel Order Numbers 253665–253668.

[12] J. Jain, A. Narayan, M. Fujita, and A. Sangiovanni-Vincentelli. Formal verification of combinational circuits. *Intl Conf on VLSI Design*, 1997.

[13] R. B. Jones. *Applications of Symbolic Simulation to the Formal Verification of Microprocessors*. PhD thesis, Stanford Univ, 1999.

[14] A. Koelbl, Y. Lu, and A. Mathur. Embedded tutorial: Formal equivalence checking between system-level models and RTL. *ICCAD*, 2005, pp. 965–971.

[15] A. Koelbl and C. Pixley. Constructing efficient formal models from high-level descriptions using symbolic simuluation. *Intl J of Parallel Programming*, 33(6):645–666, Dec 2005.

[16] A. Kuehlmann and F. Krohm. Equivalence checking using cuts and heaps. *34th DAC*, 1997, pp. 263–268.

[17] S. Minato. Generation of BDDs from hardware algorithm descriptions. *ICCAD*, 1996.

[18] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[19] H. Saito, T. Ogawa, T. Sakunkonchak, M. Fujita, and T. Nanya. An equivalence checking methodology for hardware oriented C-based specifications. *Intl High-Level Design, Validation, and Test Workshop*, 2002, pp. 139–144.

[20] L. Séméria, A. Seawright, R. Mehra, D. Ng, A. Ekanayake, and B. Pangrle. RTL C-based methodology for designing and verifying a multi-threaded processor. *39th DAC*, 2002, pp. 123–128.

[21] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. Tech Report UCB/ERL M92/41, Electronics Research Lab, Univ of California Berkeley, May 1992.

[22] F. Somenzi. CUDD: CU decision diagram package. Available from `ftp://vlsi.colorado.edu/pub/`.