On Acceleration of Logic Synthesis Algorithms using FPGA-based Reconfigurable Coprocessors

Technical Report: TR-970010

Jason Cong and John Peck
Department of Computer Science
University of California, Los Angeles, CA 90024
cong@cs.ucla.edu peck@cs.ucla.edu
http://ballade.cs.ucla.edu

Abstract

In this technical report, we present our studies on implementing two fundamental logic synthesis algorithms, tautology checking and binate covering, using an FPGA-based reconfigurable application-specific coprocessor. The uses of each algorithm are first discussed followed by the specifics of hardware accelerator implementation and interface to application software. We compare our hardware accelerator for the tautology check algorithm with the software implementation of the tautology check algorithm in Espresso II [RuVi87]. Our experimental results show that our accelerator is capable of achieving a maximum speedup factor of 2.94 and averaging 1.36 on 110 modified industry benchmarks included with the Espresso II package.

1. Introduction

There have been a number of efforts to use FPGAs as reconfigurable application-specific accelerators, such as search accelerators [PrTh93, Ho93], and image processing or visualization processors [KiDr96, CuAm96, Sh95, ChCa95, RaJa95]. They have demonstrated substantial performance gains relative to comparable software implementations. Most of these accelerators, however, apply to applications with extensive data parallelism and use FPGAs as fine-grain parallel processors. The objective of our study is to extend custom reconfigurable computing to a larger class of computation intensive applications with little or no apparent data parallelism. Our current effort focuses on accelerating a number of logic synthesis and optimization algorithms.

There have been extensive studies into using various logic synthesis techniques to optimize FPGA designs. A summary of many of those studies can be found in [CoDi96], but this work is among the first attempt to use FPGAs to accelerate computation intensive logic synthesis algorithms. Our work studies the feasibility of designing and implementing reconfigurable coprocessors used as accelerators for a class of commonly used logic synthesis algorithms. This paper presents the design and implementation of an accelerator for the tautology check algorithm and a coprocessor design for solving the binate covering problem.

2. FPGA-based Reconfigurable Coprocessor Architecture

Figure 1 shows the commercially available hardware platform named the Engineer's Virtual Computer (EVC1s) used for both coprocessor implementations in our study. We used the EVC1s, which is a single-FPGA reconfigurable board from Virtual Computer Corporation [EVC1] . Figure 2 shows a block diagram of the architecture of the EVC1s. It is an Sun-SBUS based card which features a single Xilinx 4013MQ208 FPGA and an interface to a daughterboard which can be populated with a high speed static RAM memory up to 2 MB. The FPGA on the EVC1s can be reconfigured by a Sun SPARCstation in which the board resides. A 32-bit word can be transferred to or from the EVC over the SBUS. The measured latency between a write followed by a read when transferring data from the workstation to the EVC is 9 SBUS cycles[1] on a SPARCstation 5 workstation using a user-mode memory mapped device driver. The SBUS clock rate determines the clock on the EVC1s. Our SPARCstation 5 is equipped with a 85-MHz microSPARC™II CPU and the speed of its SBUS is 21.25 MHz[2].

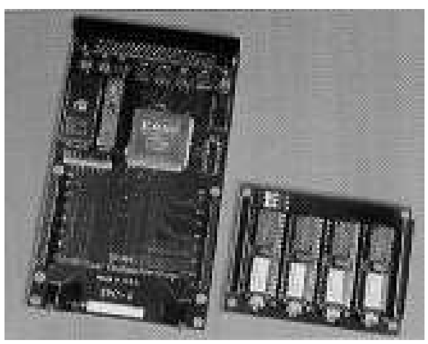

Figure 1: The single-FPGA SBUS-based EVC1s from Virtual Computer Corporation

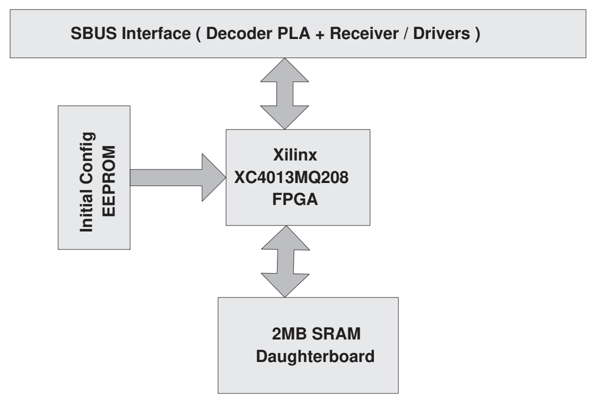

Figure 2: Block Diagram of the EVC1s

3. Tautology Checking Algorithm

The tautology checking operation is one of the most fundamental Boolean operations required in the Espresso II package for two-level logic optimization. The tautology checking algorithm is used to determine if a function $f \equiv 1$.

---

[1] This has been measured experimentally with a cycle counter design implemented on the EVC board.
[2] The SBUS speed in the SPARCstation 5 workstation is ¼ the speed of the CPU (source: Sun Microsystems)

3.1 Basic Definitions

Let B = {0,1}, Y = {0,1,2}. A **logic function** f is a function $f : B^n \rightarrow Y$ where $\mathbf{x} = [x_1,\ldots x_n] \in B^n$ is the **input** and $\mathbf{y} \in Y$ is the **output** of f. Note that in addition to the usual values of 0 and 1, the output **y** may also assume the **don't-care value**, denoted 2 in our representation. We define the **on-set** $X^{ON} \subseteq B^n$, the set of input values x such that $f = 1$, the **off-set** X^{OFF} as the set of values such that $f(\mathbf{x}) = 0$ and the **don't-care** set, $X^{DC} \subseteq B^n$, the set of values such that $f(\mathbf{x}) = 2$.

A **cube** c in B^n is encoded by a 2n-bit vector $a_1\, a_2, \ldots\, a_{2n-1}\, a_{2n}$ such that: $a_{2i-1}\, a_{2i}$ = { **00** if $\overline{x_i}$ appears in c, **01** if x_i appears in c, **10** if x_i doesn't appear } A set of cubes $C = \{c^1, c^2, \ldots, c^k\}$ is said to be a **cover** for a logic function f, if the set of cubes contain all vertices in the on-set of f and none in the off-set of f. A **minterm** is a cube in which every variable or its cofactor appears. A completely specified logic function f is a **tautology** if its off-set is empty. In other words, the outputs of f are 1 for all inputs.

3.1.1 Tautology Checking in Espresso II

The software implementation of the tautology check algorithm in Espresso II uses the unate recursive paradigm with checks for special cases to enhance its efficiency. Basically, to check if f(**x**) = f(x_1, x_2, …, x_n) is a tautology, the algorithm recursively checks if the two cofactors $f_{x_i}(\underline{x})$ and $f_{\bar{x}_i}(\underline{x})$ are tautology functions. The choice of cofactoring variable x_i is such that one or both of $f_{x_i}(\underline{x})$ and $f_{\bar{x}_i}(\underline{x})$ can become unate functions as quickly as possible. The recursion stops when f(**x**) is a unate function, or satisfies one of several special cases, so that its tautology can be determined easily. A detailed description of the tautology check algorithm can be found in [BrHaSv84]. The tautology checking procedure is primarily from the Espresso II operations of IRREDUNDANT_COVER, REDUCE, ESSENTIAL_PRIMES and LAST_GASP. In Espresso II, functions are represented as cube covers and various operations, including tautology checking are applied to the cube covers directly. We have used the software implementation of tautology checking in Espresso II which is implemented in the C language, as a basis for our analysis and modifications.

3.2 Hardware Implementation of Tautology Check Engine (TCE)

For a function with k variables, there are 2^k possible minterms. If the cubes in the cover of the function satisfy all minterms, then the function is a tautology. Figure 3 shows a block diagram of the TCE coprocessor architecture which makes use of the FPGA as a pipelined fine-grain parallel coprocessor for cube covers comprised of 8 or fewer variables. We explain our choice of 8 variables later on.

Our coprocessor consists of five primary components. The *SBUS interface* handles transfers between the SBUS and the internal state machines. The SBUS interface is capable of transferring 2 cubes (each of 16 bits) at a time. The *alternating selector and multiplexor logic* selects one of the two cubes in turn from the parts of the SBUS-interface input word where they are encoded. The *cube decoder* consumes cubes from the transfer word and decodes the input variable values in each cube into *set* and *unset* signals for each variable. Our cube encoding uses 2 bits to encode each input variable value. The base-2 encoding $\mathbf{00_2}$ of x_i produces an unset signal unset_x_i = 1, $\mathbf{01_2}$ of x_i produces a set signal set_x_i = 1, and $\mathbf{10_2}$ of x_i produces both unset and set signals. The *minterm evaluator array* consists of 256 minterm evaluators whose implementation is shown in Figure 4. Each evaluator unit takes as input one of the 256 possible combinations of set and unset signals which are produced by the cube decoder. The particular minterm evaluator pictured takes all set signals as input and is thus minterm number 189. When the logical AND of the input signals to each evaluator becomes true, the minterm corresponding to this evaluator unit has been turned on by a cube. This event is registered in the evaluator unit and kept for the duration of the tautology checking of this function. The outputs of all minterm evaluators in the array are combined using a *wired-AND network* which computes tautology.

We used the Powerview™ tools from Viewlogic for implementing portions of the TCE coprocessor. We also developed a custom tool which synthesizes the regular structures of the minterm evaluator array, cube decoder, and wired-AND network into mapped Xilinx netlist format (XNF). The TCE implementation consumes 343 CLBs of the 576 CLBs available in the Xilinx XC4013 FPGA. Of these CLBs, 256 are used to implement the minterm evaluator array and 87 are required to implement the SBUS interface logic and cube decoder. Our accelerator size of k = 8 variables appears to be the largest that can be practically implemented in the Xilinx XC4013 FPGA since for k = 9 the minterm evaluator unit cannot fit into a single CLB. Moreover, the evaluator array would grow conservatively to at least 512 CLBs and 512+87=599 > 576. In addition, the efficiency of transferring cubes to the TCE will decline since when k = 9 the cube encoding requires 18 bits and only one cube will be transferred on the 32-bit SBUS. Our k = 8 variable accelerator can encode exactly two cubes in 32 bits.

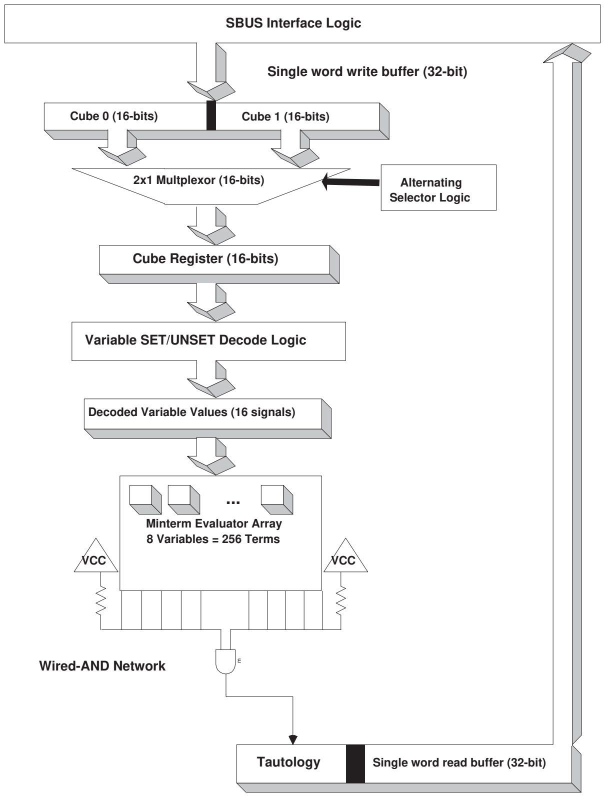

Figure 3: Tautology check engine architecture

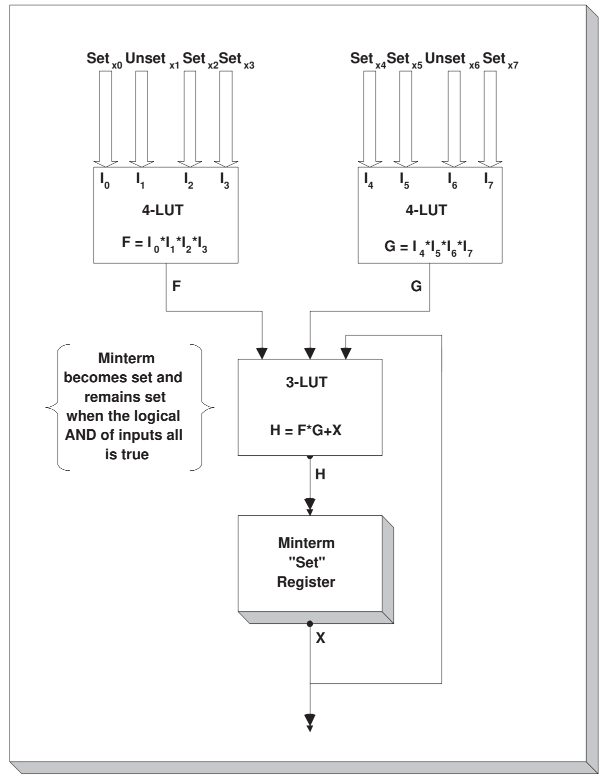

Figure 4: Minterm evaluator architecture for $x_0*x_1'*x_2*x_3*x_4*x_5*x_6'*x_7$

3.3 Major Cycle Operation of the TCE Coprocessor

We assume that the coprocessor configuration has already been downloaded to the EVC prior to the beginning of any major cycle. A major cycle of operation performs tautology checking of a function of ≤ 8 variables. The workstation first resets the coprocessor by writing to it the reserved word encoding of 32 consecutive 1's. The reset word is detected by a comparator in the coprocessor which then generates a reset signal distributed to all pipeline circuitry. Next, all cubes in the cover are transferred from the workstation to the coprocessor two at a time where they proceed through the pipeline. Cubes are first decoded into set and unset signals which are sent to the minterm evaluator array which evaluates all 256 possible minterms in parallel. The output of each minterm evaluator in the array is combined using a cascaded wired-AND to generate the tautology output. A latency of 6 clock cycles is required to process a pair of cubes from beginning to end of the pipeline; however, our architecture permits issuing of a new pair of cubes every 2 clock cycles thus achieving an effective throughput of 1 cube per clock cycle. The reason for pairing cubes together is to lessen the bottleneck that arises from the 9 cycle transfer latency over the SBUS. In practice however, it was not possible to sustain the maximum cube issuing rate to the

TCE which will be discussed in Section 3.5. At the end of the major cycle, the answer to tautology is read back from the coprocessor.

We illustrate the operation of the TCE on a cover with two variables for the sake of simplicity although the same concept easily extends to 8 variables. Assume two variables named x_1, and x_0, which have $2^k = 4$ minterms. The four minterm evaluators would have input wiring as follows: { m_0 = unset_x_1 unset_x_0, m_1 = unset_x_1 set_x_0, m_2 = set_x_1 unset_x_0, m_3 = set_x_1 set_x_0 }. Assume that a 2-variable function with three encoded cubes { **0001**, **1000, 1001** } is fed to the TCE. Recalling the 2-bit representation of a cube and interpreting each cube from right to left, the cube decoder produces the signals unset_x_1, set_x_0, for the first cube, set_x_1, unset_x_1, unset_x_0, for the second, and set_x_1, unset_x_1, set_x_0 for the third. The first set of signals turns on m_1, the second m_0, m_2, and the third m_1, m_3. Since all minterms having been turned on, the output wired-AND becomes 1 and the tautology is confirmed.

3.4 Interfacing Espresso II with the TCE Coprocessor

Espresso II encodes cubes in a bit-vector data structure which makes good use of CPU instructions which perform bit-wise operations on words. The bit-vector uses 2 bits to encode the value of each input variable and 1 bit to encode the output of each function since *m*-output functions are allowed in Espresso II. The execution profile of the tautology check algorithm shows that in practice, the joint tautology of two or more functions is computed simultaneously. This is problematic for the TCE since it can only compute tautology for one function at a time. One solution would be to serialize calls to tautology check when this situation arises. This would require making multiple calls to the TCE, one for each output function to be checked, or until an output function is found not to be a tautology. Another solution will be discussed in Section 5.

In the tautology checking algorithm, a variable is said to be active if it has not yet been cofactored against. In our modified version of the tautology check algorithm we determine if the number of active variables becomes less than or equal to 8, and then transfer control to an interface function which computes tautology via the TCE and returns its result without further recursion.. As mentioned earlier, the software implementation of the tautology checking algorithm in Espresso II is recursive with checking of special cases which enhance its efficiency by precluding unnecessary recursion. Our comparative results in Section 3.5 is obtained by placing the detection code that determines the condition of ≤ 8 active variables immediately after the regular software check in for special cases. Considerably less favorable speedup results (average speedup=0.86) were obtained when the placement of this detection code is moved before the special cases.

The TCE interface function incurs some overhead because the active variables in the cover must be identified and their input values extracted from the bit-vector representation of each cube. Padding of the cubes sent to the TCE is also necessary when the number of active variables is less than 8.

3.5 Experimental Results for the TCE Coprocessor

We have performed a profiling and timing analysis of 110 industry benchmarks included with the Espresso II package to determine the theoretical maximum speedup of the tautology check algorithm. The data gathered included CPU time spent executing the entire tautology check algorithm as well as the time spent in checking tautologies of 8 or fewer active variables. The theoretical maximum speedup shown in Figure 5 reflects the assumption of a perfect tautology checking accelerator which returns the result of tautology for covers of 8 or fewer variables in zero time. Only half of the benchmarks in Figure 5 and Figure 6 labeled for clarity.

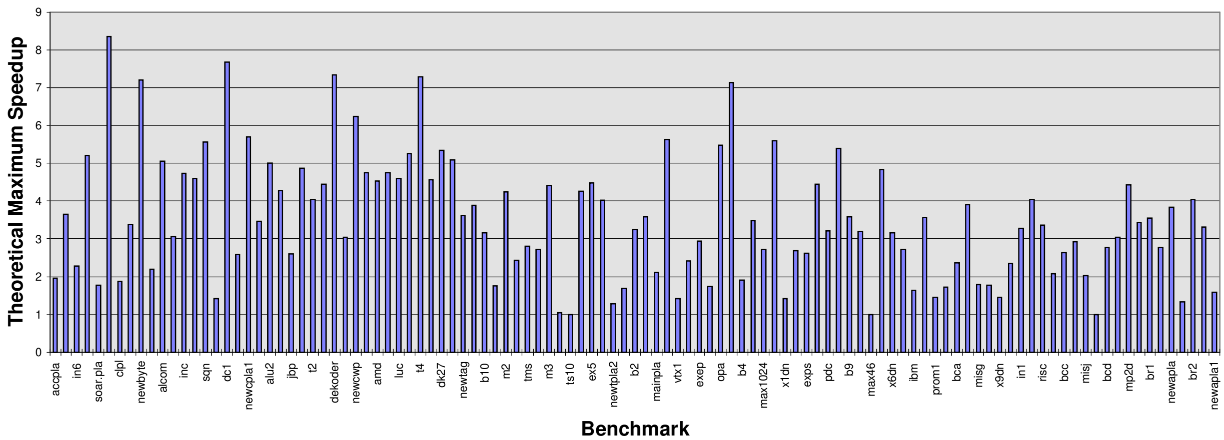

Figure 5: Theoretical maximum speedup with 8 active variable tautology check accelerator

The actual performance results for the TCE are shown graphically in Figure 6. The benchmarks used are derived from the industry benchmarks included with the Espresso II package. Since the TCE is only capable of evaluating tautologies of at most one output function at a time, we generated reduced benchmark files containing only one output variable which was chosen according to the most binate output function. Comparative speedup results are shown using these 110 modified benchmarks.

Accelerated and non-accelerated run-times for each benchmark were gathered on a SPARCstation 5 workstation running in single user mode where it is reasonable to assume that *gettimeofday()* operating system function call (which is accurate to µsec) is reasonably close to CPU time spent in application processing. The minimum resolvable time quantum of our SunOS 4.1.3 in multi-user mode is 10 milliseconds which proved to be too coarse for accurate measurements. We also performed experiments to evaluate the effectiveness of intermediate reading of the TCE's partial tautology checking result after each cube is written to the TCE. This intermediate reading entails reading the partial tautology result from the TCE after each pair of cubes is sent and returning tautology immediately in the event it is found. The motivation was to preclude sending the remainder of the cubes in the cover if tautology is found early on in the major cycle. In practice, the performance is slightly worse (max speedup=2.83, average speedup=1.34) than the implementation that awaits the final result and which only requires one read operation (max speedup=2.94, average speedup=1.36)

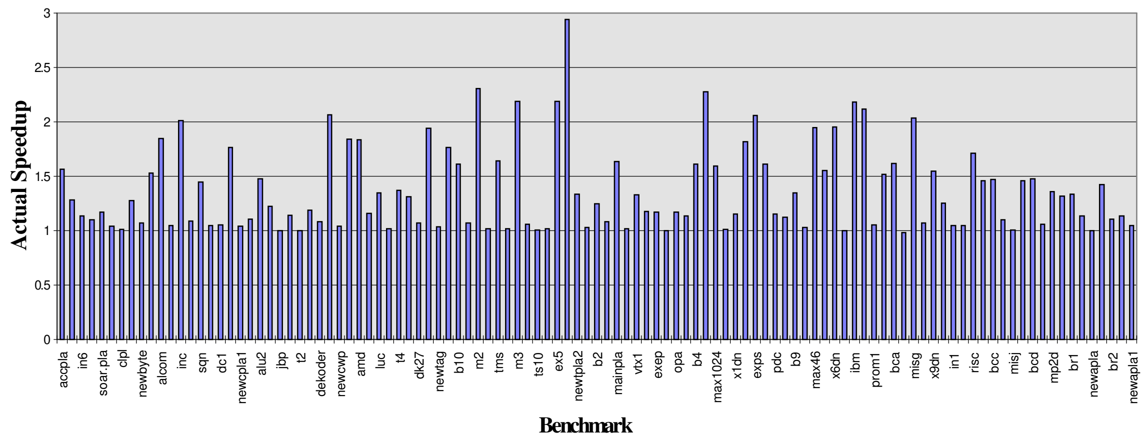

Figure 6: TCE actual performance results on 110 modified industry benchmarks

4. Binate Covering Algorithm

The binate covering problem is a well known NP-hard problem. It has several important applications in logic synthesis, such as two-level logic minimization, two-level Boolean relation minimization, three-level NAND implementation, state minimization, exact encoding, and DAG covering [Mi94] [DeGhKe94] [BrHaSv84] . The binate covering problem consists of finding a minimum cost truth assignment that evaluates *f* to 1.

4.1.1 Binate Covering Problem Formulation

Many synthesis and optimization problems can be reduced to a fundamental one: *satisfiability*. A Boolean function is satisfiable if there exists an assignment of Boolean values to the variables that makes the function true. The most common formulation of the satisfiability problem requires the function to be expressed in a *product of sums* form. The sum terms in the *product of sums* form are called *clauses*. Thus the satisfiability decision problem is the quest for a variable assignment that makes all clauses true. As an example, consider the following *product of sums* form: $(a + b)(a' + b' + c)$. The expression can be satisfied by choosing $a = 1$; $b = 1$; $c = 1$. Note that the assignment is not the only one that satisfies the expression.

Given a set of decision variables $\mathbf{x} \in \mathbf{B}^n$ and a Boolean function $f(\mathbf{x})$, in the *product of sums* form, the *minimum-cost satisfiability* problem is to find a truth assignment with the minimum cost to satisfy $f(\mathbf{x})$, where the cost of a truth assignment is the # of 1's in the assignment. For the example shown above, the assignment $a = 1$, $b = 0$, $c = 0$ is a minimum cost truth assignment (of cost 1). The binate covering problem is equivalent to the minimum-cost satisfiability problem. A comprehensive discussion of covering problems can be found in [Mi94].

4.2 Hardware Implementation of Binate Covering Engine (BCE)

Figure 7 shows a block diagram of our BCE coprocessor architecture. The coprocessor consists of five primary components. The *SBUS interface logic* performs transfer functions similar in nature to the TCE. The *programming clause select logic* decodes the clause to be selected from the input word in the SBUS interface logic. For a BCE with *n* clauses, the clause select logic produces *n* clause enable signals, though at most one is active at a time. The *clause evaluator units* are each used to evaluate one clause of the binate covering problem being solved during BCE operation. The architecture of a clause evaluator is discussed in Section 4.2.1. The *weighted min-cost counter* is used to iterate through a series of variable assignments in min-cost order. The variable assignments of increasing costs are precomputed and downloaded into the EVC on-board memory using a design that implements a DMA write mechanism prior to re-configuring the FPGA with the BCE coprocessor. By making use of reconfiguration in this way, no FPGA resources in the BCE are consumed by the logic to accomplish the preloading of input variable assignment vectors. The *wired-AND network* is used to compute whether the input variable assignment vector is a solution to the binate covering problem.

4.2.1 Programmable Clause Evaluator Architecture

Figure 8 shows the architecture of a programmable clause evaluator which is used to determine truth for one clause in a binate covering problem. Each clause evaluator unit consists of one or more variable evaluator units which are used to determine if a variable condition has been met. All variables evaluators of a particular clause evaluator share a common clause enable signal which is used during programming. Before use, each evaluator unit is programmed with the variable condition that it is to detect. When a clause enable signal becomes active, each variable evaluator in the selected clause loads its variable condition registers from the part of the SBUS interface input word which contains the variable programming information. The workstation programs each clause one at a time in this fashion until all variable evaluators of each clause evaluator have been initialized. A 3-LUT is used to compute whether the assignment to variable *k* along with the variable condition makes the variable evaluator true. The registered outputs of all variable evaluators are combined using a (wired) OR network that computes the sum of the variable evaluators to determine if the clause evaluator output is true.

The amount of CLB resources needed to implement the BCE coprocessor depends on the number of clauses to be evaluated and the number of variables in each clause. Each variable evaluator unit requires resources from about 1.5 CLBs using the three required registers as the determiner of scarce resource. The amount of CLB resources required by each clause evaluator is proportional to the number of variables in each binate covering problem clause. The number of clause evaluators required also determines the size of the decoder which is needed to generate the clause enable signals during clause evaluator programming.

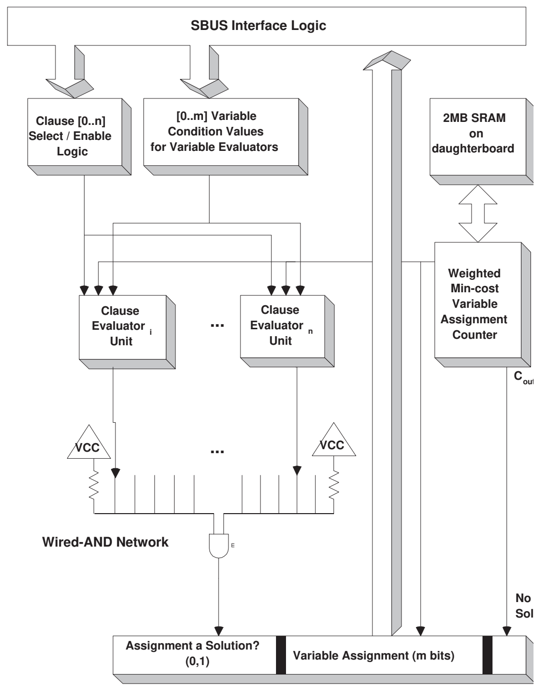

Figure 7: Binate covering engine architecture

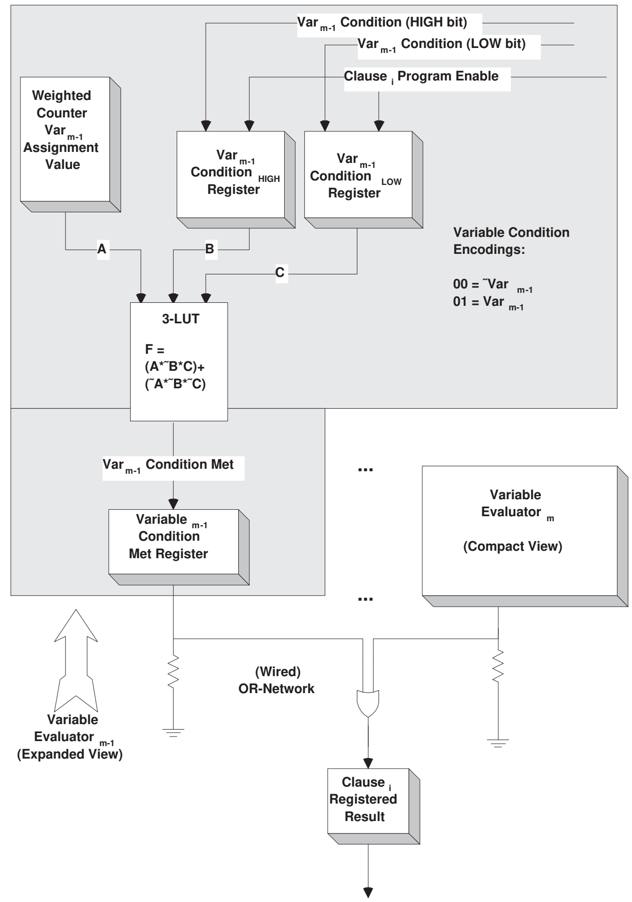

Figure 8: Binate covering clause evaluator architecture

4.3 Major Cycle Operation of the BCE Coprocessor

We assume that the on-board memory has been preloaded with the all possible variable truth assignments of increasing cost prior to the beginning of any major cycle which solves the binate covering problem. A major cycle of operation begins with programming the clause evaluators with the variable conditions that satisfy each as detailed in Section 4.2.1.

After all clause evaluators have been programmed in the BCE, the workstation writes a solve initiate bit to the SBUS-interface input word of the coprocessor which triggers the BCE intermediate clause evaluator pipeline state and the weighted counter to reset. The weighted counter then begins incrementing through variable assignments in min-cost order until the first assignment which simultaneously makes all clauses true is found. This truth is detected by the cascaded wired-AND network as shown in Figure 7. Once the truth condition has been established, the counter ceases incrementing and the min-cost variable assignment (or counter value) that established the truth condition is read back from the SBUS interface logic at the end of the major cycle. In the case where there is no variable assignment that results in truth, the carry-out signal from the variable assignment counter signals the no solution condition which is marked in a bit in the SBUS interface logic.

To illustrate the operation of our BCE coprocessor, we consider the same 3-variable example in product of sums form: $(a + b)(a' + b' + c)$. $\text{Clause}_0 = (a + b)$ produces clause-enable$_0$ and variable condition $01_a01_b10_c$ and $\text{clause}_1 = (a' + b' + c)$ produces clause-enable$_1$ and variable condition $00_a00_b01_c$. The clause programming data is transferred to the BCE during programming, and then the variable assignment counter begins at $0_a0_b0_c$. Recalling the 3-LUT function given in Figure 8, the variable assignment $0_a0_b0_c$ makes clause$_0$-var$_a$ = 0, clause$_0$-var$_b$ = 0, and clause$_0$var$_c$ = 0 because it is not in the clause. The same assignment makes clause$_1$-var$_a$ = 1, clause$_1$-var$_b$ = 1, and clause$_1$var$_c$ = 0, but the product of sums is 0 and thus this is not a solution to the binate covering problem. The next variable assignment $1_a0_b0_c$ makes clause$_0$-var$_a$ = 1, clause$_0$-var$_b$ = 0, and clause$_0$var$_c$ = 0, while clause$_1$-var$_a$ = 0, clause$_1$-var$_b$ = 1, and clause$_1$var$_c$ = 0, and the product of sums becomes 1 and this is a solution to the binate covering problem.

4.4 Experimental results on FPGA resource utilization for the BCE

We used the Powerview™ tools from Viewlogic for implementing portions of the BCE coprocessor. We also developed a custom tool which synthesizes the regular structures of the programmable clause evaluators and wired-AND network into mapped XNF. At present, our BCE coprocessor is fully debugged but has not yet been interfaced to a suitable software implementation of the binate covering algorithm to determine speedup under actual operating conditions. We present below the amount of CLB resources required to implement a family of BCE coprocessors with different clause/variable tradeoffs. Table 1 shows CLB counts for a family of BCE coprocessors, each with a different tradeoff in clause and variable counts. A family of BCE coprocessors configurations may be kept in the workstation where each can be downloaded as the configuration to the EVC board as appropriate for the binate covering problem to be solved. The static memory onboard the EVC is preloaded with a corresponding set of increasing cost variable assignment vectors for the counter, each set at its own starting address.

#Occupied CLBs out of 576 total		
	#Variables	
#Clauses	8	10
8	178	220
12	231	263
16	293	330
20	331	381
24	356	430
28	412	475
32	444	522

Table 1: CLB counts for BCE coprocessors with different clause/variable tradeoffs.

5. Limitations and Ongoing Work

Our approaches to implementing reconfigurable hardware coprocessors for the tautology check and binate covering algorithms have been limited by the SBUS transfer latency between the workstation and EVC1s and by the amount CLB resources on the Xilinx XC4013 FPGA. Our experiments have shown that there is a 9 cycle (423ns) latency between two consecutive workstation accesses of the coprocessor, even in a tightly coded program. The resulting transfer bandwidth causes our TCE to remain idle for 7 cycles out of every 9 or 78% of the time. For a pair of cubes, the result of the hardware computation can be fully computed and available 3 SBUS cycles before the workstation is able to issue the next pair for processing. Our study clearly indicates that the transfer latency of the bus technology and operating system/device driver efficiency is a bottleneck that must be overcome in order to fully exploit reconfigurable architectures as effective coprocessors for applications that require a high communication bandwidth. One solution to this problem would involve engineering much tighter coupling between CPU and FPGA-based coprocessors such as utilizing the CPU-memory bus as an interface between the CPU and re-configurable coprocessor. We believe that much higher speedup can be achieved when the bottleneck of bus bandwidth and operating system overhead is eliminated or substantially minimized. Our speedup factor will also be enhanced greatly when large FPGAs, or mult-FPGA boards are available to us. We are currently considering the possibility of using a hardware platform with 3-FPGAs and PCI-bus interface from Altera Corp.

The architecture of our 8 variable TCE takes advantage of the Xilinx 4K CLB architecture and consumes one CLB per evaluator. Checking tautology for cube covers with higher numbers of active variables requires a number of minterm evaluators proportional to 2^k and such an evaluator array requires more CLB resources per evaluator unit. The software implementation of the tautology check algorithm in the Espresso II package is capable of checking tautology efficiently for multiple output functions simultaneously through use of a clever bit-vector data structure for cube representation. Our current TCE coprocessor is only capable of checking tautology for only one function at a time and so would possibly need to be run once for each output function or until a function is found not to be a tautology to accomplish the same task.

On the other hand, we can extend our TCE design for tautology checking of multi-output functions. Figure 9 shows a straightforward extension of our current TCE to enable it to check tautology for 2-output functions simultaneously while sacrificing encoding space currently occupied by one cube in the pair currently transferred. An additional set of 2^k minterm-set registers and 2^k 3-LUTs are necessary for implementation of the second function along with two new bits in the input word which would encode the value of each of the two function outputs. The additional minterm-set registers and 3-LUTs can be implemented in $2^k/2 = 2^{k-1}$ CLBs since there are 2 4-LUTs and 2 registers per CLB. For $k = 8$, we would need 128 additional CLBs which is reasonable since our current TCE uses 256+87=343 and the total remaining CLBs are 576-343=233. Each of the function output values would accompany the cube through the pipeline and be used as a clock enable signal for a set of minterm-set registers. One clock enable would be tied to the set of minterm-registers that are the same as in Figure 4, and the other enable is tied to the new set of minterm-set registers for the second function. If a cube is present for a function the output of the function is 1, thus it generates an active clock enable signal and its corresponding set of minterm evaluators can be set. The wired-AND will also need to be extended to include the minterms from both functions 0 and 1, as labeled in Figure 9. When all minterms are set, the wired-AND output is active and the joint tautology is true.

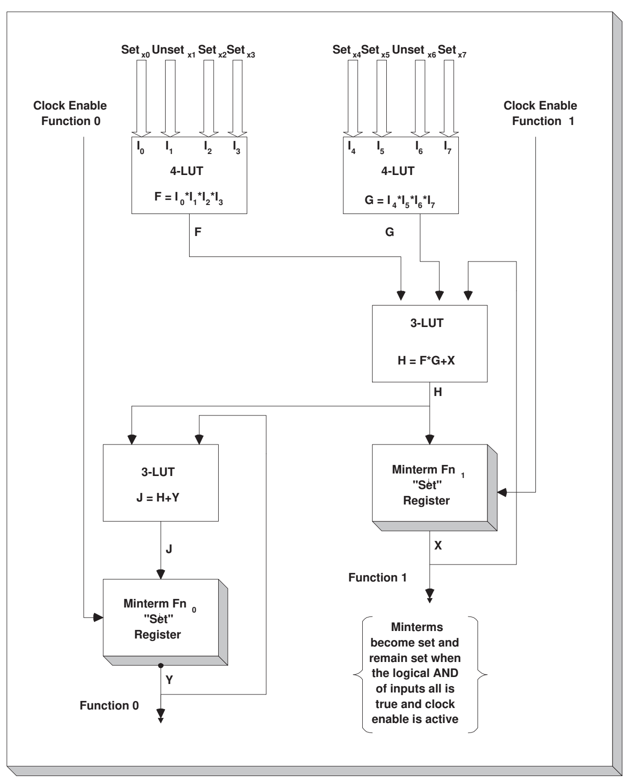

Figure 9: Minterm evaluator architecture for simultaneous tautology checking on 2 output functions

As FPGA technologies continue to improve it may be possible to preload an FPGA with several different TCE coprocessors, each optimized for accelerating particular cases. For example, if FPGA configuration contexts can be switched at run-time relatively quickly such as in the DPGA [De96], it may be possible to have two TCE implementations, one which evaluates covers of large numbers of variables, but with small numbers of cubes, and the other which evaluates covers of small numbers of variables, but with large numbers of cubes. The combinations of both on a single FPGA will enable the coprocessor to be applied to more cofactor functions during tautology checking.

Acknowledgment

This work is partially support by DARPA/CSTO under Contract DABT63-93-C-0055, NSF Young Investigator Award MIP9357582, and grants from Lucent Technologies, and Xilinx Corporation under the Cal-Micro Program.

References

[BrHaSv84] Brayton, R. K., G. D. Hachtel, C. T. McMullen, A.L. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Acad. Pub., 1984.

[ChCa95] Chow, H., S. Casselman, H. Alnuweiri, "Implementation of a Parallel VLSI Linear Convolution Architecture Using EVC1s," *Proc. IEEE Symposium on FPGAs for Custom Computing Machines*, 1995.

[CoDi96] Cong, J. and Y. Ding, "Combinational Logic Synthesis for LUT Based Field Programmable Gate Arrays," *ACM Trans. on Design Automation of Electronic Systems*, 1996 V1, 2 145-204.

[CuAm96] Culbertson, W., R. Amerson, R. Carter, P. Kuekes, G. Snider, "Exploring Architectures for Volume Visualization on the Teramac Custom Computer," *Proc. IEEE Symposium on FPGAs for Custom Computing Machines*, 1996.

[Mi94] de Micheli, G., *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, 1994.

[De96] DeHon, A., "DPGA Utilization and Application," *Proc. ACM/SIGDA Intl. Symposium on Field Programmable Gate Arrays*, 1996.

[DeGhKe94] Devadas S., S., A. Ghosh, K. Keutzer, *Logic Synthesis*, McGraw-Hill, 1994.

[Ho93] Hoang, D. "Searching Genetic Databases on Splash 2," *Proc. IEEE Workshop on FPGAs for Custom Computing Machines*, 1993.

[KiDr96] King, W., T. Drayer, R. Conners, P. Araman, "Using MORPH in an Industrial Machine Vision System," *Proc. IEEE Symposium on FPGAs for Custom Computing Machines*, 1996.

[PrTh93] Pryor, D., M. Thistle, N. Shirazi, "Text Searching on Splash 2," *Proc. IEEE Workshop on FPGAs for Custom Computing Machines*, 1993.

[RaJa95] Ratha, N., A. Jain, D. Rover, "Convolution on Splash 2," *Proc. IEEE Symposium on FPGAs for Custom Computing Machines*, 1995.

[RuVi87] Rudell, R. and A. Sangiovanni-Vincentelli. Multiple-Valued Minimization for PLA Optimization. IEEE Transactions on Computer-Aided Design of Integrated Circuits, 6(5):727-751, September 1987.

[Sh95] Shand, M. "Flexible Image Acquisition using Reconfigurable Hardware," *Proc. IEEE Symposium on FPGAs for Custom Computing Machines*, 1995.

[EVC1] The EVC1s user's manual, 1994. (http://vcc.com)