CSEE 6861

Prof. Steven Nowick

Richard L. Rudell.
Multi-valued logic Minimization for PLA Synthesis. Master's thesis. UC Berkeley 1986
chs. 4.7–4.8

4.7. LAST_GASP and SUPER_GASP

The basic iteration of Espresso-II (REDUCE, EXPAND, IRREDUNDANT) faces the following obstacles: (1) The EXPAND step uses heuristics to choose one prime implicant (from all of the prime implicants which cover a cube) to replace each cube in the cover; and (2) the REDUCE algorithm is cube-order dependent so that cubes which are reduced first tend to reduce more than cubes which are reduced later. Different minimization algorithms have managed these problems in different ways. For example, MINI uses the *reshape* operation in order to sidestep these problems, and Prestol-II uses the *change_shape* opera-

tion (twice in succession) in order to escape these problems. I describe here the Espresso-II strategy LAST_GASP and the Espresso-MV strategy SUPER_GASP for improving the basic minimization algorithm.

4.7.1. LAST_GASP

This algorithm first computes the maximal reduction of every cube of the ON-set cover F and creates a new cover G. If a cube cannot be reduced it is ignored. A modified version of the EXPAND algorithm expands each of the cubes of G. The EXPAND procedure is modified so that: (1) the expansion of a cube is stopped as soon as it is determined that it cannot cover any other cubes; the cube is removed from G in the case that it cannot expand to cover any other cubes; and (2) all of the cubes are expanded even if they are covered by the expansion of a different cube. As shown in [BMH84], those cubes that succeed in covering some other reduced cube are potentially useful primes for reducing the cardinality of the cover. These new primes are simply added to the cover F, and the IRREDUNDANT procedure then extracts a minimal subcover. Because the number of reduced cubes which can expand to cover other reduced cubes tends to be very small, this technique is applicable to a wide range of problems. In particular, I have not found any examples for which the running time of the algorithm is dominated by the LAST_GASP operation.

4.7.2. SUPER_GASP

Espresso-MV also has an optional routine SUPER_GASP. This algorithm computes the maximal reduction of each cube of the cover F and then generates *all* of the prime implicants which cover the cube (rather than only a single prime implicant which covers the cube). In order to generate all of the prime implicants which cover a cube, the algorithm given in Section 4.3 (EXPAND) is used. By sorting this set of prime implicants, duplicate prime implicants are easily detected. IRREDUNDANT then extracts a minimal subcover from the remaining set of prime implicants. Note that if IRREDUNDANT returns the minimum number of cubes necessary to implement the function, then no single

iteration of REDUCE, EXPAND, and IRREDUNDANT can do any better from the same starting point.

Of course, the process of generating all of the primes which cover the maximally reduced cubes may greatly expand the size of the cover. (In particular, if the original cover were all minterms, the generation of all of the primes covering each minterm would be an inefficient way to generate all of the primes for the function.) The program Espresso-MV is careful to terminate the generation of all of the primes in the case there are too many primes, in which case the LAST_GASP strategy is used instead. In practice, the SUPER_GASP can be selected optionally when the program Espresso-MV is run. In Chapter 6, I report experimental results with this option.

4.8. MAKE_SPARSE

When the outer loop of the Espresso-MV algorithm terminates, the solution consists of an irredundant cover of prime implicants which represents the original function. However, depending on the final implementation of the multiple-valued function, we may desire a final cover which does not necessarily consist of prime implicants. One goal is to reduce the number of transistors needs to implement each literal of a cube. This depends on the number of 0's and 1's in the literal, but it also depends on the type of variable as shown in Table 4.8.1:

Variable Type	Number of transistors	Comment
binary-valued variable	count number number of zeros	sparse
multiple-valued variable (for a two-bit decoder)	count number of zeros	sparse
multiple-valued variable (for the output part)	count number of ones	dense
multiple-valued variable (for the input encoding problem)	count number of ones (unless literal is full)	dense

Table 4.8.1. Transistors per Literal in a PLA

For example, if the function being minimized represents a two-level multiple-output PLA function, then each 0 in the cube for a binary-valued variable corresponds to a

transistor in the AND-plane of the PLA, but each 1 in the multiple-valued output variable corresponds to a transistor in the OR-plane of the PLA.

Another example is minimizing a multiple-valued function for the state-assignment program **KISS**. For these functions, it is preferred that the multiple-valued variables have as few 1's as necessary (which will lead to fewer constraints for the embedding problem).

Hence, the binary-valued variables and multiple-valued variables resulting from a bit-paired PLA are desired to be **dense** (i.e., have many 1's), and the multiple-valued variable resulting from the output-part of a PLA are desired to be **sparse** (i.e., have few 1's). Finally, the multiple-valued variables resulting from a symbolic variable (as in **KISS**) should be sparse unless the cube does not depend on this particular variable. With these observations we define, for each variable, whether the variable is to be a **sparse variable** or a **dense variable**. The MAKE_SPARSE procedure then attempts to satisfy these goals.

MAKE_SPARSE consists of two steps: LOWER_SPARSE removes redundant parts from the sparse variables and RAISE_DENSE attempts to add parts to the dense variables (which may be possible following LOWER_SPARSE because the cubes are no longer prime implicants). These two algorithms are iterated until there is no more reduction of any sparse variable, or until there is no more expansion of any dense variable. This algorithm is iterated in Espresso-MV (as opposed to Espresso-II which only executed each step once) because the total literal reduction is worth the extra expense.

During the first iteration of LOWER_SPARSE and RAISE_DENSE the cardinality of the cover cannot decrease (because the cover is an irredundant, and consists of prime implicants). However, in extreme cases, it is possible for the cardinality to decrease in subsequent iterations. In fact, the procedure MAKE_SPARSE can be viewed as a complete minimization algorithm. (The **pop** program from Berkeley [Sim83] uses essentially this simple algorithm, but without the powerful techniques for each of the basic steps as in MAKE_SPARSE. However, this minimization algorithm is restricted in the size of the set of prime implicants which it can explore.)

In the discussion that follows, we assume, as usual, that F is a cover for the ON-set, D is a cover for the DC-set and R is a cover for the OFF-set.

4.8.1. LOWER_SPARSE — Reduce the Sparse Variables

The goal of LOWER_SPARSE is to remove parts from the sparse variables so as to reduce (if possible) the number of 1's in these variables for each cube. This procedure can be viewed as cube reduction applied to each cube with the reduction retained only for the the multiple-valued variables. However, this technique suffers from the same problem as REDUCE, namely that the order in which the cubes are processed can greatly affect the total amount of reduction possible.

Instead, the IRREDUNDANT routine is used to select, for a particular part, which cubes are redundant; this part is set to 0 for the redundant cubes. This way the cube ordering problem is avoided, and the more powerful heuristics of IRREDUNDANT are used to find a good reduction of the sparse variables.

For each value j of a sparse variable X_i, define e^j to be the cube of $X_i^{|j|}$. By finding an irredundant cover for $(F \cup D)_{e^j}$ we can determine which cubes of F can have part j removed. If a cube does not belong to the irredundant subcover of $(F \cup D)_{e^j}$, then the part in the cube is redundant and can be removed. These parts are removed, and, after all parts for a variable have been processed, the next variable is processed.

Note that by using the IRREDUNDANT algorithm rather than REDUCE, the order in which the cubes are examined in part j of variable X_i is immaterial. (Further, the order in which the parts of any variable is processed is also immaterial.) But, the order in which the sparse variables are processed does influence the reduction of variables which are not processed first. In Espresso-MV, LOWER_SPARSE is applied to sparse variables corresponding to multiple-valued variables resulting from the input-encoding problem. This is done to simplify the constraints which arise from the multiple-valued parts. The last variable processed is the multiple-output variable. Admittedly, this heuristic is a little crude.

4.8.2. RAISE_BV — Expand the Dense Variables

As mentioned earlier, we desire that the binary-valued variables, and the variables resulting from bit-pairing be dense. After reducing the multiple-valued variables with LOWER_SPARSE, the resulting set of cube is no longer prime. Hence, we can try to expand this set of cubes by expanding only the dense parts of each cube. This is done with a modified version of EXPAND which removes all of the sparse parts from the *free* set (cf. sec 4.3) before finding the expansion of a cube. Hence, none of the sparse parts will be expanded.

Interestingly, EXPAND will still check for cubes which, when limited to only the dense variables, can expand to cover another cube. As mentioned earlier, on subsequent iterations of MAKE_SPARSE it is possible for the cardinality of the cover to decrease. If it is possible for a cube to be covered, EXPAND will expand the dense variables so as to cover the cube.