A Linear Divisor Extraction Algorithm

Alan Mishchenko Robert Brayton

Department of EECS, University of California, Berkeley

{alanmi, brayton}@eecs.berkeley.edu

Abstract

Divisor extraction applied to a sum-of-product (SOP) representation of multi-output Boolean functions is an important part of logic synthesis. This algebraic approach is more scalable than Boolean decomposition when used to derive a new multi-level structure for functions whose current structure is unavailable or of poor quality. The traditional algorithm for divisor extraction, "fast extract", is quadratic in the number of cubes because it finds common divisors by enumerating cube pairs. Although "fast extract" works well for fanin-bounded networks whose nodes have relatively small SOPs, its performance degrades for flattened logic blocks whose output functions may have thousands of cubes when expressed in terms of the primary inputs. This paper describes a new "fast extract" algorithm, FXCH, which uses cube hashing rather than cube-pair enumeration. While both algorithms are worst-case quadratic in the number of input variables, FXCH is linear in the number of cubes, which leads to speed-ups in practical applications with large SOPs. FXCH can be used to synthesize depth-bounded networks, and features a novel post-processing step to factor out shared logic from the output plane of a multi-output SOP.

1. Introduction

Increase in design size and pressure to reduce runtime motivate a re-evaluation of algorithms used in CAD tools. Any algorithm whose complexity is greater than linear might be too expensive for use in the modern tools.

This paper proposes to improve a fundamental logic synthesis algorithm, which is responsible for slow-downs on large designs. Complexity is reduced from quadratic to linear, while quality largely remains the same.

Another way of dealing with the algorithm complexity is to exploit concurrency. We do not consider concurrency but observe that our approach is orthogonal, potentially resulting in an even faster implementation.

This paper focuses on divisor extraction, which has been a key building block of logic synthesis tools since early days of multi-level synthesis [7]. Its goal is to transform a Boolean function into a multi-level circuit. For small functions with 10-20 inputs, it is typically easy to derive a circuit using (1) algebraic factoring [6] applied to an SOP of the function, or (2) Boolean decomposition [1][10][12] applied to a truth table or a Binary Decision Diagram (BDD) [8]. For larger functions, only algebraic factoring is scalable enough.

In many practical situations, a Boolean function is multi-output. If each output is transformed into a multi-level circuit independently, much sharing of logic is possibly lost, leading to a substantial area increase. Thus, extracting shared logic across multi-outputs is important.

The traditional algebraic method to extract shared logic for multi-output functions, is known as “fast extract” (FX) [17], which is implemented in many logic synthesis tools, in particular, in SIS [18] and ABC [2].

FX can solve two types of sharing extraction problems, depending on whether some circuit structure has to be preserved or a completely new structure is needed:

- In the first case, the input to FX is a Boolean network and the SOPs are single-output functions of nodes of this network. The network structure is preserved while the logic sharing found is limited to the divisors extracted from the single-output functions.
- In the second case, the input to FX is a logic block with an unknown structure whose functionality can be derived as a BDD or as a multi-output table in Espresso PLA format [16]. In this case, FX starts with a multi-output SOP, in which some cubes are shared across outputs. The logic structure derived in this case is not related to the original circuit structure that existed before the SOP was constructed.

In this paper, these two scenarios of divisor extraction are handled uniformly, except for one improvement applicable to multi-output tables.

A detailed overview of FX is deferred to Section 2.3. One aspect of FX important for the present work, is enumeration of cube pairs used to compute a set of shared divisors. This quadratic-time enumeration is the main limiting factor when FX is applied to SOPs with thousands of cubes. Brute-force enumeration results in millions of cube pairs and typically only a small fraction of these (say, 3%) yield useful divisors that do not exceed some size limit.

A standard practice in such cases is to divide each large SOP into parts, each containing up to a hundred cubes, and then limit the enumeration of cube pairs to each part. This reduces runtime but degrades quality because divisors whose cubes ended up in different parts are not considered.

This paper proposes a different approach to divisor extraction from a set of large SOPs. The approach is based on the observation that only divisors with at most four literals are useful in practice. This is because the quality of

logic extraction typically does not suffer from limiting divisors to two-variable functions and MUXes.

The new approach, called FXCH, "Fast eXtract with Cube Hashing", uses a hash table for computing divisors, which reduces the complexity from quadratic to linear in the number of cubes.

Besides cube hashing, several additional improvements to FX are proposed. One allows the extraction of many divisors at once without updating the priority queue. This might lead to a minor degradation in area but gives FXCH a better control over the depth of the resulting circuit. Another improvement enables factoring of the output part of a multi-output SOP.

It should be noted that cube hashing has wider applicability than described in this paper. For example, it can enable fast heuristic minimization of very large SOPs, where all identical cubes and distance-1 cube pairs can be computed efficiently.

The paper is organized as follows. Section 2 contains necessary background. Section 3 describes a linear algorithm to find all shared divisors that do not exceed a size limit. Section 4 describes a modified version of FX that makes use of the proposed algorithm and two additional improvements. Experimental results are given in Section 5, while Section 6 concludes the paper.

2. Background

2.1 Boolean function

Unless stated otherwise, *function* refers to a completely specified Boolean function $f(X)$: $B^n \rightarrow B$, $B = \{0,1\}$. The *support* of f is the set of variables X, which influence the output value of f. Support size is denoted by $|X|$. Functions $f(X)$ and $g(Y)$ have *disjoint supports* if $X \cap Y = \varnothing$.

Expressions $\bar{x}$ and x are the *negative literal* and the *positive literal* of variable x, respectively. "Negative" and "positive" are *polarities* of variable x in the corresponding literals. The AND of literals is a *product* (or a *cube*).

A product is an *implicant* of f if it implies f, that is, if f is equal to 1 for any assignment of variables that makes the product equal to 1. A *prime implicant*, or *prime*, of f is an implicant of f, if removing any literal from it produces a product that does not imply f. A *minterm* of f is an implicant of f containing all the variables. The *distance* between two cubes is the number of different literals in them.

The OR of implicants of f is a *sum-of-products* (SOP) expression. If the value of an SOP for all input minterms is the same as the value of the function f, then SOP is a *cover* of f. The number of products in a SOP C is denoted by $|C|$. An *irredundant sum-of-products expression* (ISOP) of a cover of f is an OR of primes of f, such that no prime can be deleted without changing f. A cube c_1 is contained in cube c_2 if c_1 contains all the literals of c_2. A cover is said to be *single-cube containment-free* if it does not have a cube pair such that one cube of the pair contains the other.

2.2 Boolean network

A *Boolean network* (or *circuit*) is a directed acyclic graph (DAG) with nodes corresponding to logic gates and edges corresponding to wires connecting the nodes. It is assumed that each node has a unique integer called *node ID*.

A node n has zero or more fanins, i.e. nodes driving n, and zero or more fanouts, i.e. nodes driven by n. The primary inputs (PIs) are nodes without fanins. The primary outputs (POs) are a subset of nodes of the network, connecting it to the environment. A fanin (fanout) cone of node n is a subset of nodes of the network, reachable through the fanin (fanout) edges of the node.

2.3 Algebraic factoring

Algebraic *factoring* [6] derives a multi-level circuit composed of AND and OR gates for a completely-specified Boolean function. Factoring is performed by repeatedly applying algebraic *division* to the SOP representation of a Boolean function F. Each division begins by computing an algebraic *divisor* D, followed by deriving a quotient Q and a remainder R, such that $F = D \,\&\, Q + R$, where & and + are Boolean AND and OR operations. Next, factoring is applied to D, Q, and R, as long as they have non-trivial algebraic divisors.

Factoring is *algebraic* because the algorithms for divisor extraction and division handle SOPs as arithmetic expressions. In this case, a variable and its negation are treated as different variables, without exploiting the complete Boolean algebra to manipulate them. For example, idempotence ($x \,\&\, x = x$; $x + x = x$) and complementarity ($x \,\&\, \bar{x} = 0$; $x + \bar{x} = 1$) are not used. As a result, algebraic division leads to divisor/quotient pairs with disjoint supports. This substantially simplifies Boolean division and allows it to scale to SOPs with thousands of cubes, but it also limits its generality compared to Boolean division, which is based on the complete Boolean algebra.

We refer the reader to [6] for the details of factoring algorithms and point out that the main emphasis of factoring is on finding good divisors. Each divisor is a small SOP that is deemed good for decomposing a given function by algebraic division.

Algebraic factoring is frequently used in CAD tools because, being close to linear in the number of cubes, it scales well with an increase in SOP size. The following are its main limitations:

- Divisor extraction and division are both performed for a given SOP. A different SOP could lead to different results. There is no good recipe what SOP should be used, although it has been observed that the canonical irredundant sum-of-products (ISOP) computed as shown in [14] creates SOPs favorable for factoring. This is likely because a fixed variable order, which is imposed during computation, forces some regularity on the literals appearing in the cubes.
- The divisor computation algorithms are heuristic and may produce multiple divisors, which may lead to different results. It is not clear what divisor is better until at least one iteration of division is performed.
- Divisors are computed for each function without considering other functions of the same network. As a result, a good divisor for a function is not necessarily a good divisor for the network. Even if the network has only one output, the components generated by applying

the division algorithm (functions D, Q, and R above) represent a multi-output decomposition problem, which could benefit from logic sharing extraction.

- To address the above limitations, it has been proposed to enumerate all algebraic divisors for each function and then search for a good shared one. However, the number of different divisors can be very large. Moreover, once a divisor is used, updating the scores of other divisors is non-trivial.

Understanding these limitations led to the development of the algebraic divisor extraction algorithm known as "fast extract" (FX) [17] outlined below.

2.4 Original "fast extract"

The following are the key ingredients of the original FX algorithm for extracting shared algebraic divisors from a set of logic functions comprising a logic network.

- Input functions are given as a set of SOPs.
- Two-cube and single-cube-two-literal divisors are considered at the same time. In practice the quality of results does not suffer if only two-input functions and MUXes are considered because other divisors can be expressed in terms of these.
- Divisors and their complements are treated uniformly.
- The weight of each divisor is the number of SOP literals it can save when extracted from a set of SOPs.
- All divisors are stored in the queue, which is repeatedly accessed to extract divisors with the highest weight.
- After each extraction, the SOPs and divisor weights are updated, new divisors are created and added to the queue. These new divisors are expressed in terms of other divisors, resulting a multi-level network.
- To find the initial set of divisors, cube pairs of the SOPs are enumerated and, for each pair, a divisor is found. In practice, no loss in quality is seen if divisors are restricted to to those whose SOP literal count does not exceed four.

2.5 Hashing

Hashing is a fundamental technique used to check if an element belongs to a set. For this, a *hash table* is created, which maps each element into an integer index pointing to a *bucket* with a linked list of entries stored under the same index. By iterating through the entry list, one can check if a given element is in the table, that is, belongs to the set.

Given an efficient *hash function*, which uniformly distributes integer indexes across the array of buckets, hashing of one element is done in constant time, while hashing of all elements in the set is done in time linear in the number of elements.

In this work, we deal with SOP cubes, from which divisors are to be extracted. Each cube is an ordered set of literals represented as integer numbers. Negative and positive literals of the same variable are treated as unrelated because of the algebraic nature of the divisor extraction.

The following is a summary of the hashing technique used in the paper to hash the cubes and detect the divisors.

Hash value of a literal l is a random N-bit unsigned integer, $h(l)$. Hash value of a cube $c = (l_0, l_1, \ldots, l_{n-1})$ is the sum of the hash values of its literals: $H(c) = h(l_0) + h(l_1) + \ldots + h(l_{n-1})$. *Hash function*, which maps a cube into a hash table bucket, is the hash value of the cube modulo the table size: $F(c) = H(c)$ % HashTableSize.

This hash function can efficiently hash not only the original cubes, but also *sub-cubes*, that is, cubes derived from the given cube by removing several literals. To hash sub-cubes, we subtract from the hash value of the cube, the hash values of literals removed from it. For example, given cube c with literal l removed, the hash value of the sub-cube $c' = c \setminus l$ is $F(c') = [H(c) - h(l)]$ % HashTableSize.

If 32-bit integers are used to store hash values, it is important that bit-width N of the random numbers used to hash literals are large enough to avoid an overflow after adding hash values for all literals to derive hash values of cubes and sub-cubes. Since, in our application, the number of literals in a cube is asserted to be less than 256 (=2^8), we can use 24-bit random numbers to represent hash values of the literals. This allows us to create hash tables with up to 2^24 entries without degrading the quality of hashing.

To summarize, the hashing technique presented in this section is designed to speed up computation of hash values, which, in turn, allows us to efficiently check the existence of a cube or a sub-cube in a hash table populated with all available cubes and sub-cubes.

3. Cube hashing algorithm

The main technical contribution of this paper is an algorithm for finding cube pairs resulting in useful algebraic divisors. The algorithm is linear in the number of cubes in all the SOPs representing the problem.

Consider a simpler problem of finding identical cubes in a given SOP. A naïve quadratic-time approach requires iterating over all cube pairs and comparing them. A smarter quasi-linear-time approach sorts the cubes by their integer value and compares pair-wise only adjacent cubes. A better linear-time approach uses a hash table to hash each cube. In the end, identical cubes can be found in the hash table by exploring a bucket with entries having the same hash value.

Now consider the problem of finding all cubes that differ in one literal. The key insight is that, for such cubes, there exists a literal in one of the cubes, such that if it is removed, the two cubes become identical. To this end, we first create a hash table containing all of the original cubes. Next, for each cube, we try to remove each literal and check if the reduced cube is in the hash table. If so, this cube and the given cube differ in the given literal only.

Now consider the case of finding cube pairs whose divisors have at most four literals. In this case, for each cube we hash (a) the cube as it is, (b) all sub-cubes reduced by removing each of its literals, and (c) all sub-cubes reduced by removing each pair of literals. The hash table can be used to find pairs of identical sub-cubes, which point to pairs of original cubes with up to two literals removed. The removed literals correspond to a *divisor* while the two identical sub-cubes correspond to the *base* of the divisor, that is, the common part of the two original cubes remaining after the two-cube divisor is extracted.

For example, consider Boolean function $F = c_1 + c_2$ where cubes, $c_1 = abcd$ and $c_2 = abef$, result in the same base, *ab,*

after literals *c* and *d* (*e* and *f*) are removed from c_1 (c_2). The divisor *cd* + *ef* after extraction leads to *F* = *ab*(*cd* + *ef*).

It should be noted that a large SOP may lead to a large hash table. For example, an SOP having 1M cubes with 20 literals each can lead to 1M * 20 * 19 / 2 ≈ 200M sub-cubes and as many entries in the hash table. One hash table entry takes 16 bytes, which calls for a 3.2GB hash table.

Fortunately, there is a way to dramatically reduce the hash table size, by observing that only those sub-cubes need to be hashed, which have at least one other sub-cube with the same hash value. Otherwise, a sub-cube can be ignored without impacting the number and quality of extracted divisors. In practice, only about 5% of the sub-cubes are useful and need to be hashed, which reduces hash table memory from 3.2GB to 160MB in the example above.

To remove useless sub-cubes, we perform one round of pruning using a Bloom filter [5]. The Bloom filter with the probability of a false-positive close to 10% (meaning that only 10% of useless sub-cubes will be in the hash table) requires about 1 byte per entry. This is a 16x reduction, compared to 16 bytes per entry in the hash table. The resulting Bloom filter takes 1B/entry * 200M = 200MB.

We also note that the Bloom filter and hash table do not have to be allocated at the same time, if we keep track of the sub-cubes that passed the Bloom filter. This can be done with a bit-map, which requires 1 bit per sub-cube, or 25MB in the example above. In our implementation, we allocate one contiguous array memory and first use it as the Bloom filter and later reuse it for the hash table.

4. Modifications to "fast extract"

We discuss modifications to the top-level FX procedure motivated by the use of cube-hashing rather than cube-pair enumeration in the proposed FXCH algorithm.

As pointed out, FXCH is linear in the number of cubes and quadratic in the number of variables. FX is also quadratic in the number of variables (due to two-literal divisor enumeration for each cube). In practice, this is not a limiting factor because cubes rarely contain more than 20 literals. It is also easy to restrict literal enumeration for cubes with many literals without sacrificing much quality.

FXCH expects that the input SOP (1) does not have identical cubes and (2) is single-cube containment free. Otherwise, the algorithm may overlook some divisors. For example, cube c_1 = *ab* contains c_2 = *abcdef*. This pair is not detected by FXCH because detection requires removing four literals from c_2, without removing any literals from c_1, while FXCH can remove at most two literals from a cube.

This observation suggests that cube hashing should be performed in phases, first detecting and removing identical cubes, next detecting and removing distance-1 cubes, etc. This is a natural way to proceed if the original cover is a set of minterms. If the original cover is of unknown quality, a fast single-cube containment check, which also removes duplicates, can be used. Although this check is worst-case quadratic in the number of cubes, it is close to linear in practice due to efficient heuristics that guide the cube pair comparisons, as evidenced by the scalability of the CNF preprocessor SatELite [11], which uses a similar technique to check clause subsumption (a dual of cube containment).

The pseudo-code of FXCH is shown in Figure 1. The procedure takes a multi-output SOP, and returns a SOP after factoring and a set of divisors. The SOP after factoring depends on the original variables as well as new divisors.

FXCH performs several rounds of divisor extraction. In each round, cubes and sub-cubes of the current SOP are hashed, cubes pairs of distance four or less are found, and the resulting divisors are added the priority queue. Next, a fixed number of best divisors is chosen and extracted from the current SOP. The process is repeated as long as there are non-trivial divisors and a resource limit have not been reached. Resource limits include: (a) the number of divisors to extract; (b) the type and weight of divisors to extract, (c) logic levels of the generated divisor network, (d) runtime, etc.

```
(sop; divisors) performFastExtractWithCubeHashing (
   sop S,              // S is the original multi-output SOP
   parameters P ) // P  user preferences and resource limits
{
    preprocess the SOP by removing identical/contained cubes;
    while ( there are divisors and resource limits allow ) {
        perform hashing of the cubes and sub-cubes of the SOP;
        detect all cube pairs that are distance four or less;
        compute divisors and add them to the queue by weight;
        find a fixed number of best divisors using the queue;
        extract these divisors from the current SOP and save them;
    }
    return (SOP after factoring; the set of divisors extracted);
}
```

Figure 1. Pseudo-code of FXCH.

5. Experimental results

The proposed algorithm is implemented in ABC [2] as command *fxch*. The command takes as input a multi-output SOP and produces a multi-level logic network in ABC. The correctness of the created network is verified by combinational equivalence checking (command *cec*), which compares the derived AIG against the original multi-output function represented by the SOP.

The following experiments are reported.

5.1 Cube hashing

In this experiment, we generated N random minterms in terms of 24 variables while avoiding duplicated minterms. Next, we compared the runtime needed to compute the set of all pairs of distance-1 minterms among those generated: (1) using the brute-force enumeration of cube pairs and (2) using the proposed cube hashing. Table 1 lists the runtimes in each case. The number of pairs of distance-1 minterms is the same in both cases. The runtimes are on a 2.70GHz i7 Intel Core CPU.

Table 1: Runtime comparison for quadratic and linear algorithms to detect all pairs of distance-1 minterms.

N	Distance-1 pairs	Quadratic, s	Linear, s
1K	0	0.01	0.01
10K	71	0.28	0.01
100K	7194	28.25	0.18
1M	714712	2915.43	2.67

5.2 Comparing against traditional FX

We compare the performance of *fxch* against that of FX implemented in ABC (command *fx*). The benchmarks used are N-input single-output Boolean functions whose output is 1 if and only if the input is an N-bit prime number.

Table 2 shows the runtime as well as the number of nodes and levels after converting the resulting multi-level network into an AIG and optimizing it with *dc*2 in ABC. Only the *fx/fxch* runtimes are reported, excluding the time needed to perform AIG-based optimization. The question mark in the table means that divisor extraction using *fxch* succeeded, but factoring of the leftover SOP in ABC did not complete due to reaching a resource limit.

Our current implementation is preliminary. It only considers two-literal divisors and ignores XORs and MUXes, which require considering four-literal divisors. This is the reason for the degradation in terms of the number of AIG nodes and levels, compared to *fx*.

Table 2: Comparison of *fxch* against *fx* for functions detecting a prime number of the given bit-width.

N	Primes	*fx* (node/lev/time)	*fxch* (node/lev/time)
10	172	248 / 13 / 0.05 sec	284/ 14 / 0.02 sec
12	564	784 / 15 / 1.43 sec	876/ 16 / 0.03 sec
14	1900	2296 / 18 / 26.67 sec	2420/ 18 / 0.05 sec
16	6542	timeout after 900 sec	7567/ 21 / 0.23 sec
18	23000	timeout after 900 sec	? / ? / 1.15 sec
20	82025	timeout after 900 sec	? / ? / 6.15 sec
22	295947	timeout after 900 sec	? / ? / 31.05 sec
24	1077871	timeout after 900 sec	? / ? / 152.60 sec

The final version of the paper will include updated experimental results. In particular, four-literal divisors will be handled, additional data-structure optimizations will be enabled, and the problem with factoring of large SOPs in ABC will be fixed.

6. Conclusions

The paper follows the trend of revising traditional algorithms to reduce complexity and enable faster runtimes and improved scalability for larger problems. The classic divisor-extraction algorithm "fast extract" widely used in the present-day logic synthesis tools is considered. An improvement of this algorithm is proposed, which reduces its complexity from quadratic to linear and enables its applicability to larger problem.

Future work will explore

- Computing shared multi-output SOPs for the primary outputs in terms of the primary inputs directly from the circuit, without computing truth tables or BDDs.
- Enhancing algebraic factoring for the case when SOPs contain external don't-cares.
- Applying on-the-fly factoring to compact large sets of satisfying assignments of a satisfiable SAT instance.

Acknowledgements

This work is partly supported by SRC contract 1875.001 and NSA grant "Enhanced equivalence checking in crypto-analytic applications". We also thank industrial sponsors of BVSRC: Altera, Atrenta, Cadence, Calypto, IBM, Intel, Mentor Graphics, Microsemi, Synopsys, and Verific for their continued support.

7. REFERENCES

[1] R. L. Ashenhurst, "The decomposition of switching functions". *Computation Lab*, Harvard University, 1959, Vol. 29, pp. 74-116.

[2] Berkeley Logic Synthesis and Verification Group. *ABC: A System for Sequential Synthesis and Verification*. http://www-cad.eecs.berkeley.edu/~alanmi/abc

[3] V. Bertacco and M. Damiani, "Disjunctive decomposition of logic functions," *Proc. ICCAD '97*, pp. 78-82.

[4] P. Bjesse and A. Boralv, "DAG-aware circuit compression for formal verification", *Proc. ICCAD '04*, pp. 42-49.

[5] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors", Communications of ACM, 13 (7), pp. 422–426, 1970.

[6] R. K. Brayton and C. McMullen, "The decomposition and factorization of Boolean expressions," *Proc. ISCAS '82*, pp. 29-54.

[7] R. Brayton, G. Hachtel, and A. Sangiovanni-Vincentelli, "Multilevel logic synthesis", *Proc. IEEE*, Vol. 78, Feb.1990.

[8] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. Comp.*, Vol. C-35, No. 8 (August, 1986), pp. 677-691.

[9] M. R. Choudhury and K. Mohanram, "Bi-decomposition of large Boolean functions using blocking edge graphs". Proc. ICCAD'10, pp. 586-591.

[10] A. Curtis. *New Approach to the Design of Switching Circuits*. Van Nostrand, Princeton, NJ, 1962.

[11] N. Eén anfd A. Biere. "Effective preprocessing in SAT through variable and clause elimination". Proc. SAT'05, LNCS, vol. 3569, Springer 2005.

[12] V. N. Kravets. *Constructive Multi-Level Synthesis by Way of Functional Properties*. Ph. D. Thesis, University of Michigan, 2001.

[13] V. N. Kravets and P. Kudva, "Implicit enumeration of structural changes in circuit optimization". *Proc. DAC'04*, pp. 438-441.

[14] S. Minato, "Fast generation of irredundant sum-of-products forms from binary decision diagrams". *Proc. SASIMI'92*, pp. 64-73.

[15] A. Mishchenko, R. K. Brayton, and S. Chatterjee, "Boolean factoring and decomposition of logic networks", *Proc. ICCAD'08*, pp. 38-44.

[16] PLA Format. http://www.ecs.umass.edu/ece/labs/vlsicad/ece667/links/espresso.5.html

[17] J. Rajski, J. Vasudevamurthy, "The test-preserving concurrent decomposition and factorization of Boolean expressions", *IEEE Trans. CAD*, Vol.11 (6), June 1992, pp.778-793.

[18] E. Sentovich, et al, "SIS: A system for sequential circuit synthesis", *Tech. Rep. UCB/ERI, M92/41*, ERL, Dept. of EECS, Univ. of California, Berkeley, 1992.