Since hierarchical structure occurs in many systems (the author is aware of at least three other hierarchically structured cache protocols), there is some prospect that BDD trees will be useful in practice, in spite of their relatively complicated structure in comparison to OBDD's. As a practical matter, one would expect the BDD tree algorithms to be somewhat less efficient in their implementation because of their greater complexity. This inefficiency is mainly offset, however by the use of OBDD's inside the representation. Profiling shows that most of the time is spent in the OBDD calculations – thus, optimizing the BDD tree code is not a major concern.

For general circuits, it is possible that practical algorithms can developed for tree decomposition for widths at least as large as can be handled by BDD trees – at least there are no theoretical results precluding this. Any progress in this area would make BDD trees more widely applicable.

Acknowledgements I would like to acknowledge discussions with Randy Bryant and Jerry Burch, who suggested a structure similar to BDD trees in 1992. I would also like to thank Bob Kurshan and David Long for critical reading.

References

[BCL91] Jerry R. Burch, Edmund M. Clarke, and David E. Long. Symbolic model checking with partitioned transition relations. In A. Halaas and P. B. Denyer, editors, *Proceedings of the IFIP International Conference on Very Large Scale Integration*, Edinburgh, Scotland, August 1991.

[BCM+90] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science*, June 1990.

[Bod93] H. L. Bodlaender. A linear time agorithm for finding tree-decompositions of small treewidth. In *ACM STOC '93 (25th)*, CA, USA, May 1993.

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.

[Dil88] D. Dill. Trace theory for automatic hierarchical verification of speed-independent circuits. Technical Report 88-119, Carnegie Mellon University, Computer Science Dept, 1988.

[Kar89] K. Karplus. Using if-then-else DAGs for multi-level logic minimization. In C. L. Seitz, editor, *Advanced Research in VLSI: Proc. Decennial Caltech Conf. VLSI*, pages 101–117, March 1989.

[McM92] K. L. McMillan. Symbolic model checking: an approach to the state explosion problem. Technical Report 92-131, Carnegie Mellon University, Computer Science Dept, 1992.

[MS91] K. L. McMillan and J. Schwalbe. Formal verification of the Encore Gigamax cache consistency protocol. In *International Symposium on Shared Memory Multiprocessors*, 1991.

[RS86] N. Robertson and P. D. Seymour. Graph minors. II. algorithmic aspects of tree-width. *J. Algorithms*, 7:309–322, 1986.

[Sei80] C. L. Seitz. Ideas about arbiters. *Lambda*, 10(14), 1980.

due to "artificial correlations" between clusters in the fixed point approximations caused by interleaving on the global bus. This could possibly be corrected using a "modified search order" [BCL91], but clearly BDD trees by themselves are not sufficient to allow us to model check large examples of the cache protocol with the basic symbolic method. On the other hand, BDD trees would be very effective in proving invariants of the protocol, which does not require fixed point calculations.

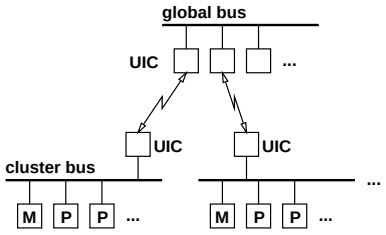

Fig. 8. Gigamax memory architecture.

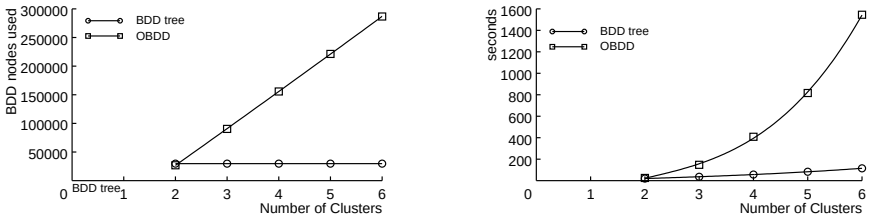

Fig. 9. For Gigamax verification, transition relation size, and transition relation construction time.

8 Conclusion

BDD trees provide a canonical boolean representation, which has a linear upper space bound for circuits of fixed tree width. This is a generalization of a similar result for OBDD's which applies only to path width. At the present time, tree decompositions of a given width appear to be obtainable algorithmically only for very small widths [Bod93], but certain systems have a natural hierarchical structure that lends itself to easy tree decomposition. In fact, the variable orderings for the above experiments were obtained by adding a simple primitive called "group" to the description language that creates a subtree in the tree decomposition. Finding good locations for these "group" commands was a trivial matter.

level of the hierarchy are identical, hence share the same space. As a result, the total space used to represent the transition relation is logarithmic in the number of arbiter cells. Figure 7 plots the space used to represent the transition relation usings OBDD's and using BDD trees, as a function of the number of users of the arbiter.

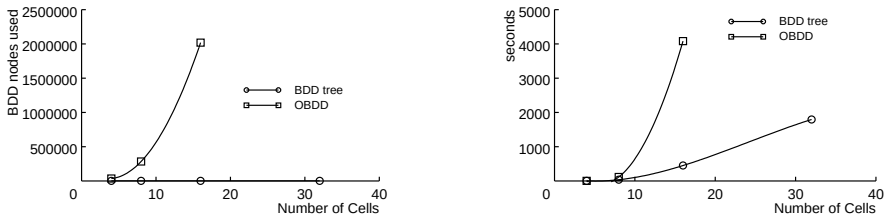

Fig. 7. For model checking tree arbiter, transition relation size and verification time.

The second requirement of the symbolic model checking method is to compute the fixed point series that characterize the temporal logic operators. Of particular importance in this process is computing a representation of the reachable states of the model. The basic step in this computation is to compute

$$q_i(V') = q_{i-1}(V') \vee \exists V.\ (q_{i-1}(V) \wedge R(V, V'))$$

where q_i represents the ith approximation to the reachable states (q_0 is the set of initial states), and R represents the transition relation. This step is repeated until a fixed point is reached. Figure 7 shows a plot of the time required for this computation using the OBDD representation and the BDD tree representation, as a function of the number of users of the arbiter. There are several factors that go into this time. First, the number of iterations required to reach a fixed point increases as roughly $n \log n$ where n is the number of users. The most distant state is one where every user has requested the resource, received acknowledgement, removed its request, and is about to receive a negation of its acknowledgement. For the case of 32 users, this required 433 steps. A second factor is the size of the transition relation, and a third is the size of the fixed point approximations.

As a second example, we consider a distributed cache consistency protocol, studied in [MS91] and [McM92]. This system also has a natural tree decomposition, as depicted in figure 8. For the transition relation, we find a result similar to that of the tree arbiter – the number of BDD nodes used remains constant as the number of cluster buses increases. The time to construct the transition relation using BDD trees increases roughly linearly. Thus, as the size increases, BDD trees soon outperform OBDD's by an order of magnitude (see figure 9). However, in the reachable states computation, we find the size of the fixed point approximations increasing exponentially for both methods. This appears to be

The first example we will use is a tree structured speed-independent arbiter circuit studied by Dill [Dil88] and based on work of Seitz [Sei80]. Dill likened the operation of this circuit to an elimination tournament. The arbiter cells are organized in a hierarchy, each cell arbitrating between two cells at the next lower level of the hierarchy (see figure 6). Requests and acknowledgements follow a four phase signaling protocol. The implementation of a single cell as a speed independent circuit (due to Seitz) is also depicted in the figure.

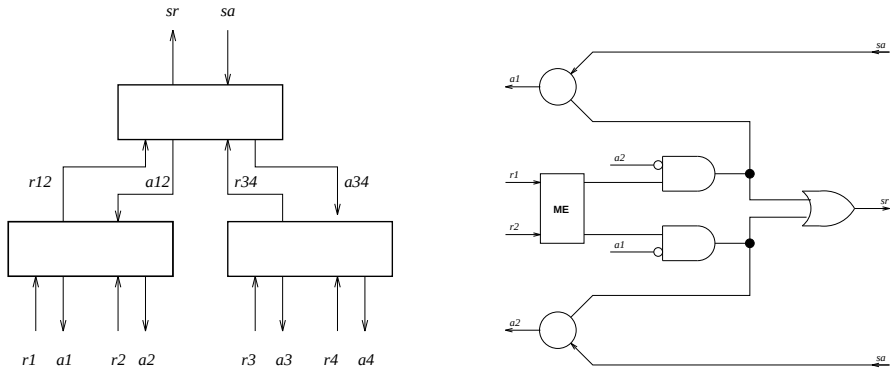

Fig. 6. A tree arbiter with four users and three cells; a speed-independent tree arbiter cell.

The first requirement of the symbolic model checking method is to represent the transition relation as a boolean formula. There is an obvious tree decomposition of this formula, where we simply group all of the gates and inputs of a single arbiter cell in one vertex of the tree. The width of this decomposition is the number of inputs of the cell plus twice the number of gates in the cell, since we have two boolean variables for each gate, representing the old and new logic values of its output[3]. Since this number is a constant independent of the number of cells, we expect a linear upper bound on the size of the transition relation represented as a BDD tree, using a suitable tree ordering of the variables. In fact, something more interesting happens – we find the total number of OBDD nodes used in the BDD tree representation is bounded by a constant. This is because all cells of the arbiter are identical, hence all of the OBDD's (except for those representing the users and server) are identical and share the same space. This is accomplished by using the same OBDD variables in the boolean representations at each point in the tree. For a related reason, the total space used by the matrices M is logarithmic in the number of cells, since all BDD trees at a given

[3] Note that the transition relation is represented by a boolean formula, but this does not correspond to any combinational logic function computed by the circuit. In this case, the transition relation is a conjunction of formulas, each constraining the output behavior of one gate. See, for example, [McM92, BCM+90] on how to represent the transition relation of asynchronous circuits.

Lemma 19. *If circuit g has a tree decomposition of width k and degree d, it has a tree decomposition of width k and degree d such that $|T| \leq |g| - k + 1$.*

This lemma is similar to lemma 2.5 in [Bod93], regarding "smooth" decompositions.

Theorem 20. *Let C_k be the class of circuits of tree width k. The optimum BDD tree representation of $\bar{g}$, where $g \in C_k$, has size $O(|S(g)|)$.*

Proof. Start with a tree decomposition of the circuit of size proportional to the circuit and degree 3, per the above lemmas. Select a root χ for the tree, where $g \in X_\chi$, making T a binary tree. For each variable v let $t(v)$ be the vertex where the variable v appears in the tree decomposition closest to the root. This gives us the necessary variable decomposition to produce a BDD tree representation. We now put an upper bound on the number of equivalence classes for the child function $\bar{g}_\nu$ of each vertex ν. For any truth assignment σ to $S(\nu)$, let $A_\nu(\sigma)$ be the set of truth assignments to X_ν that are consistent with σ. This set can be characterized inductively – $A_\nu(\sigma)$ is the set of all truth assignments ρ to X_ν such that

- for all gates $h = (f, x_1, \ldots, x_n) \in X_\nu$ having all inputs also present in X_ν, $\rho(h) = f(\rho(x_1), \ldots, \rho(x_n))$,
- for all children μ of ν, ρ agrees with some $\phi \in A_\mu(\sigma \downarrow S(\mu))$ over the intersection of their domains,
- ρ agrees with σ.

We can show that A_χ maps every total truth assignment to a singleton $\{\rho\}$ that determines the truth value of the circuit output g. From this we can infer that for any vertex ν, two truth assignments σ and σ' such that $A_\nu(\sigma) = A_\nu(\sigma')$ are in the same equivalence class. This follows because $A_\nu(\sigma)$ and the values of the remaining variables are sufficient to determine the circuit output, by our inductive construction[2]. Since the size of X_ν is bounded by k, the number of sets of truth assignments to X_ν, and hence the number of equivalence classes, is at most 2^{2^k}. This bounds the number of rows and columns of the matrix at any vertex. In addition, the number of variables assigned to every vertex is at most k, which bounds the size of any OBDD to be at most 2^k. Since the number of vertices is $O(|g|)$, it follows that the total size of the BDD tree representation is $O(|g|2^{k2^{2k}})$. □

7 Experimental results

We now consider the application of the above results to the verification of tree structured circuits and systems, using the symbolic model checking technique. This technique uses operations on boolean formulas to construct a symbolic representation of the transition relation of a model, and to compute fixed point series that characterize formulas in temporal logic that are given as specifications [BCM+90, McM92].

[2] In fact, from this construction, we can derive a linear time SAT algorithm for g.

1. v, where v is a variable, or
2. $(f, x_1, \ldots, x_n)$ where f is an operator of arity n and $x_1, \ldots, x_n$ are circuits. The circuits $x_1, \ldots, x_n$ are said to be the *children* of $(f, x_1, \ldots, x_n)$.

The set of subcircuits of a circuit g, denoted $S(g)$ is either

1. $\{g\}$, if g is a variable, or
2. $\{g\} \cup_{i=1}^{n} S(x_i)$ where $x_1, \ldots, x_n$ are the children of g.

Definition 15. The *function* $\bar{g}$ associated with a circuit g is defined recursively, such that

- $\bar{v}(\sigma) = \sigma(v)$ if v is a variable and
- $\bar{g}(\sigma) = f(\bar{x}_1(\sigma), \ldots, \bar{x}_n(\sigma))$ if $g = (f, x_1, \ldots, x_n)$.

Definition 16. A tree decomposition of a circuit g is an undirected tree T with vertices $V(T)$ and edges $E(T)$ where each vertex $i \in V(T)$ is associated with $X_i \subseteq S(g)$ such that

1. for all circuits $(f, x_1, \ldots, x_n)$ in $S(g)$, for some $i \in V(T)$,
$$\{(f, x_1, \ldots, x_n), x_1, \ldots, x_n\} \subseteq X_i$$
and
2. for all $i, j, k \in V(T)$, if j lies on the path in T from i to k, then $X_i \cap X_k \subseteq X_j$.

Remark: In other words, each vertex of the tree is labeled with a subset of the gates. The set of vertices labeled with a given gate is connected, and every gate is somewhere in the same set with all of its inputs.

Definition 17. The *width* of a tree decomposition is the max of $|X_i|$ for $i \in V(T)$. The tree width of a formula g is the least w such that g has a tree decomposition of width w.

A tree decomposition may have any degree (that is, any number of vertices adjacent to a given vertex). However, given a tree decomposition of width k, we can construct a *binary* tree decomposition of width k (that is, where T is a binary tree).

Lemma 18. *If circuit g has a tree decomposition of width k, it has a tree decomposition of width k and degree at most 3.*

Proof. Suppose there is a vertex $i \in V(T)$ with degree $d > 3$. Divide the neighbors of i into two subsets L and M. Replace i with two vertices l and m, such that $X_l = X_m = X_i$, where l is adjacent to m and all elements of L, and m is adjacent to l and all elements of M. The degree of l is $\leq \lceil d/2 \rceil + 1$, which is less that d, and likewise for m. The width of the result is still k. Thus we may reduce the degree of any node of the tree, until the degree of the graph is 3. □

5 Quantification algorithm

Symbolic model checking requires computing $\exists V.\ f$, where f is a boolean formula and V is a collection of boolean variables. The variables of V may be eliminated from f one at a time. In practice this is very slow, however, since it involves traversing the entire structure of f once for every variable in V. There is a more efficient approach that is analogous to the "bottom up" quantification procedure for OBDD's [McM92]. Space allows only a cursory treatment of it here.

The procedure traverses the BDD tree in pre-order. We first eliminate the variables in V from the root, using the quantification procedure for OBDD's. Next, we eliminate the variables in V from the left child f_l. This is done by first replacing each OBDD leaf i in the left child by row i of f_M, and then recursively calling the quantification procedure on f_l. This is essentially treating the rows of matrix M as n-ary decision nodes, and the the right child as an n-ary free variable. Since the quantification procedure on OBDD's requires the ability to compute the logical disjunction of leaves, we must be able to perform disjunction on rows of the matrix. This is done simply by taking their component-wise disjunction. As with other OBDD operations, we use dynamic programming, indexing the rows in a hash table so that we never generate the same row twice. The result of this is that we never take the disjunction of the same pair of rows twice, since the the result of all disjunctions is stored in a cache.

Once the quantification of the left child is complete, it is traversed in lexical order, gathering its leaves (*i.e.*, rows) into a matrix, and replacing each leaf by the index to the corresponding row. The variables in V are then eliminated from the right child in a similar manner, treating the columns as decision nodes. Finally, a reduction procedure eliminates redundant rows and columns in the matrix, yielding the BDD tree representation of the result.

6 Tree width and BDD trees

BDD trees efficiently represent a class of boolean circuits that are hierarchically structured in a certain quantifiable sense. The concept of the tree width of a graph was introduced by Robertson and Seymour in their work on graph minors [RS86]. For circuits, we can define an analogous concept. For any natural number k, the class of circuits of tree width k can be represented by BDD trees using linear space.

A *tree decomposition* of a circuit associates the gates of a circuit with vertices of a tree. Intuitively, the width of a tree decomposition is the maximum number of wires that would run through any vertex if the circuit were wired through the tree. The tree width of a circuit is the least width of any tree decomposition of that circuit.

To make this definition precise, we first need a formal definition of a circuit.

Definition 14. Given a collection of boolean *operators* and *variables*, a *circuit* is either

---

nodes once per call to BDDT_pairs, which would make the overall time complexity $O(|\langle f\rangle|^2|\langle g\rangle|^2)$.

```
function BDDT_apply(o,f,g)
    if f and g are constants return o(f,g)
    if f is a constant, f = ([f], 0, 0)
    if g is a constant, g = ([g], 0, 0)
    let p_l = BDDT_pairs(f_l, g_l)
    let p_r = BDDT_pairs(f_r, g_r)
    create matrix P of dimension length(p_l) × length(p_r)
    for j = 1 ... length(p_l), let (r_f, r_g) = p_l(j) in
        for k in 1 ... length(p_r), let (c_f, c_g) = p_r(k) in
            P(j,k) = BDD_apply(o, f_M(r_f, c_f), g_M(r_f, c_g))
    let H be an empty hash table, n_l = 0
    for j in 1 ... length(p_l)
        let r be row j of P, p = p_l(j)
        if (r,k) is stored in H (for some k), then let φ_l(p) = k
        else let φ_l(p) = n_l, ρ_l(n_l) = j, n_l = n_l + 1
    {compute φ_r and ρ_r in similar manner, using columns instead of rows}
    let r_l = BDDT_apply(φ_l, f_l, g_l)
    let r_r = BDDT_apply(φ_r, f_r, g_r)
    create a matrix M of dimension n_l × n_r
    for j in 0 ... n_l − 1
        for k in 0 ... n_r − 1
            M(j,k) = P(ρ_l(j), ρ_l(k))
    return BDDT_find(M, r_l, r_r)
end function
```

Fig. 5. The function BDDT_apply. Note, BDD_apply is "apply" for OBDD's, as described in [Bry86].

BDDT_apply(ϕ_l, f_l, g_l), by induction. Similarly, the representation of the right child function is $c_1 =$ BDDT_apply(ϕ_r, f_r, g_r). Finally, $M(j,k)$ is equal to $h\sigma_r\sigma_l$ for σ_r, σ_l such that $\phi_l(p_l(\sigma_r)) = j$ and $\phi_r(p_r(\sigma_r)) = k$, which is the same as saying σ_r is in left child equivalence class j and σ_r is in left child equivalence class k. Hence $(M, c_0, c_1) = \langle h \rangle$. □

Theorem 13. *BDDT_apply$(o, \langle f \rangle, \langle g \rangle)$ runs in time $O(|\langle f \rangle| \times |\langle g \rangle|)$.*

Proof. At each vertex in the BDD tree structure, BDD_apply uses time proportional to the size of the product matrix P, which is bounded by $|f_M||g_M|$, the product of the sizes of the f and g matrices. The same is true of BDDT_pairs (which executes only once for each vertex in the tree structure because of caching). The number of executions of BDD_pairs and BDD_apply is bounded by the number of pairs of BDD nodes in f and BDD nodes in g, also due to caching[1]. Hence, the overall execution time is $O(|\langle f \rangle| \times |\langle g \rangle|)$. □

[1] However, if the BDD implementation is such that BDD nodes can be shared between different matrices in the tree structure, then BDD_pairs may visit every pair of BDD

```
procedure BDD_pairs(f,g,var H, var i)
    procedure recurse(f,g)
        if (f, g) stored in J return
        else if f and g are both leaves then
            if (?, (f, g)) not in H, store (i, (f, g)) in H, i = i + 1
            return
        else
        let f0,f1,g0,g1 be the cofactors of f and g
            w.r.t. their least common variable.
        recurse(f0,g0), recurse(f1, g1)
        end if
    end procedure
    let J be an empty hash table
    recurse(f,g)
end procedure
```

Fig. 4. The function BDD_pairs.

it possible to group the left and right child pairs into equivalence classes, and number the equivalence classes canonically. Two left child pairs are in the same equivalence class if they index equal rows in the product matrix, and similarly, two right child pairs are equivalent if they index equal columns of the product matrix. BDDT_apply then generates a map ϕ_l (ϕ_r) that takes each left (right) child pair and returns the number of its equivalence class. An inverse map ρ_l (ρ_r) is also generated that maps each equivalence class to some left (right) child pair in that class. BDDT_apply is then called recursively with ϕ_l (ϕ_r) to produce the left (right) child function. The root matrix can then be culled from the product matrix, using the inverse function ρ_l (ρ_r) to select a row (column) of the product matrix for each row (column) of the root matrix.

Theorem 12. *If o is a binary operator in the natural numbers, and f and g are boolean functions on V, then*

$$BDDT\_apply(o, \langle f \rangle, \langle g \rangle) = \langle o(f, g) \rangle$$

Proof. Let $h = o(f, g)$ and $(M, c_0, c_1) = \text{BDDT\_apply}(o, \langle f \rangle, \langle g \rangle)$. For any truth assignment σ_l to the left child variables, let $p_l(\sigma_l)$ be the pair $(f_l(\sigma_l), g_l(\sigma_l))$. Similarly, for any truth assignment σ_r to the right child variables, let $p_r(\sigma_r)$ be the pair $(f_r(\sigma_r), g_r(\sigma_r))$. If σ_c is a truth assignment to the remaining variables, then $P(p_l(\sigma_l), p_r(\sigma_r))(\sigma_c) = h(\sigma_r \cup \sigma_l \cup \sigma_c)$. Thus row $p_l(\sigma_l)$ of P determines the left child equivalence class of σ_l and column $p_r(\sigma_r)$ determines the right child equivalence class of σ_r. Hence, the number of the left child equivalence class of σ_l is one less than the number of distinct rows $p_l(\sigma'_l)$ where $\sigma'_l \leq \sigma_l$. Since the rows of the product matrix are ordered lexically (by BDDT_pairs) this one less than the number of distinct rows of index $\leq p_l(\sigma_l)$, which is $\phi_l(p_l(\sigma_l))$. Hence $\phi_l(f_l, g_l)$ is the left child function of h, and its representation is $c_0 =$

of subtrees, as we will see later. If the matrix M is simply $[n]$, where n is a constant, then BDDT_find returns n.

The function "apply" is used to combine two BDD trees using a given basis function. For example, given the BDD trees for a and b, "apply" can be called to build the BDD tree for $a \wedge b$ or $a \vee b$. We will consider the case of binary operations only. Algorithms for unary and ternary operations can easily be derived. The binary "apply" function takes any binary function o on the natural numbers, and the representation of two boolean functions $\langle f \rangle$ and $\langle g \rangle$, and returns $\langle o(f,g) \rangle$ (where $o(f,g)(\sigma) = o(f(\sigma), g(\sigma))$). A major component of "apply" is a function BDDT_pairs that takes two BDD trees representing functions f and g, and returns the list of pairs (j,k), such that for some truth assignment σ, $f(\sigma) = j$ and $g(\sigma) = k$. The list is ordered lexically by the least truth assignment producing each pair. Note: in the following f_l is used to denote the left child component of f, and f_r the right child, while f_M is used to denote the matrix (*i.e.*, root function).

function BDDT_pairs(f,g)
  if f and g are constants **return** $((f,g))$
  if f is a constant, $f = ([f],0,0)$
  if g is a constant, $g = ([g],0,0)$
  if (f,g,r) stored in hash table **return** r
  let p_l = BDDT_pairs(f_l, g_l)
  let p_r = BDDT_pairs(f_r, g_r)
  create empty hash table H, let $i = 0$
  for $j = 1 \ldots \text{length}(p_l)$, let $(r_f, r_g) = p_l(j)$ in
    for k in $1 \ldots \text{length}(p_r)$, let $(c_f, c_g) = p_r(k)$ in
      BDD_pairs$(f_M(r_f, c_f), g_M(r_f, c_g), H, i)$
  let r be the list $(p_0, \ldots, p_{i-1})$ where (j, p_j) in H
  store (f,g,r) in hash table
  return r
end function

Note that we first check a hash table to see if BDDT_pairs has previously been computed for the same inputs. This means that BDDT_pairs will execute $O(|f|+|g|)$ times during the execution of "apply", no matter how many times it is called. Note also that we traverse the list of left child pairs in the *outer* loop. This means that we are assuming the left child variables are more significant in the lexical order than the right child variables, which in turn are more significant than the root variables. The procedure BDD_pairs (figure 4) traverses a pair of binary decision diagrams, storing each new pair of leaves reached in the hash table H, and incrementing the counter i with each new pair. The traversal order guarantees that the pairs are stored in lexical order.

The BDDT_apply function (figure 5) uses BDDT_pairs to compute a list of output pairs for the left and right child functions. It then computes a "product matrix" P, that yields the value of the function $o(f,g)$ for each combination of a left child pair and a right child pair. Having the product matrix makes

in σ, and false if v is false in σ. Thus, there is a literal for each variable, a literal being a boolean function, whereas a variable is just an arbitrary element from a finite collection. As to basis functions, the "nand" function by itself is a complete basis. "And", "or" and "not" are also complete. To generate the representation of any circuit, we require algorithms to generate the representation of literals, and to apply the basis functions to the representations of their arguments.

We begin with the literals. The representation of a literal consists of one OBDD node at a suitable place in the hierarchy. For example, figure 3 shows a BDD tree representing the literal v, where the tree address of v is $t(v) = 01$. Literals are constructed by the following function:

function BDDT_literal(v,t)
  if $t = \epsilon$ **return** ([BDD_literal(v)], 0, 0)
  else if head(t) = 0
    return BDDT_find($\begin{bmatrix} 0 \\ 1 \end{bmatrix}$, BDDT_literal($v$, tail($t$)), 0)
  else return BDDT_find($\begin{bmatrix} 0 & 1 \end{bmatrix}$, 0, BDDT_literal($v$, tail($t$)))
end

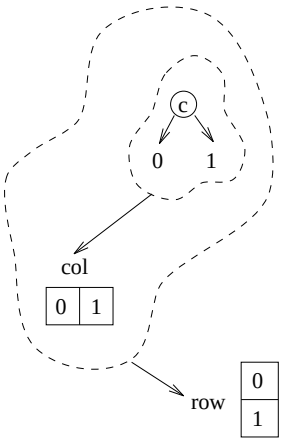

Fig. 3. BDD tree representing the literal v.

Note that all the leaves of the BDD tree are the boolean function 0. In the above procedure, the function BDD_literal returns an OBDD consisting of one decision node labeled with v. The function BDDT_find looks up a triple, consisting of a matrix and two BDD tree pointers, in a hash table. Finding a match, it returns the previously created BDD tree structure, otherwise it creates a new one and returns it. This guarantees that *two BDD trees that are equal always have the same memory address*. Thus equality comparisons between BDD trees can be made in constant time. Significant space can also be saved by sharing

– otherwise $R_\nu(f)$ is a triple (M, c_0, c_1) where M is a $|f|_{\nu 0} \times |f|_{\nu 1}$ matrix,
  - $M(j,k) = f f^j_{\nu 0} f^k_{\nu 1}$
  - $c_0 = R_{\nu 0}(f_{\nu 0})$
  - $c_1 = R_{\nu 1}(f_{\nu 1})$

In the example, $R_{00}(f_{00}) = 0$. This is because $S(00)$ is the empty set, hence there is only one equivalence class of truth assignments to $S(00)$, hence f_{00} is a constant function returning 0. Likewise $R_{01}(f_{01}) = 0$. Thus $|f|_{00} = |f|_{01} = 1$ (that is, both children of 0 have only one equivalence class). This means that $R_0(f_0)$ is a triple (M, c_0, c_1), where M is a one-by-one matrix and $c_0 = c_1 = 0$. The only element of this matrix is just the function f_0, since f^0_{00} and f^0_{01} are both empty assignments.

Definition 10. For any function $f : (V \to \{0,1\}) \to \mathbf{N}$, let $\langle f \rangle = R_\epsilon(f)$.

Now consider $R_\epsilon(f)$ in our example. Since $|f|_0 = |f|_1 = 2$, M is a 2-by-2 matrix. The 01 element of this matrix, for example, is f cofactored by f^0_0 and f^1_1 (that is, by representatives of equivalence class number 0 of the left child and equivalence class number 1 of the right child). This gives us 00 for ab and 1 for c, which yields $00 + 1d = d$. See the figure 2 for the rest of the representation.

We will refer to $\langle f \rangle$ as the *abstract representation* of f. The *concrete representation* will use OBDD's to stand for the boolean functions occurring in the matrices. For most purposes, there will be no need to distinguish between the two.

Theorem 11. $\langle\rangle$ *is one-to-one.*

Proof. Let I be a function that maps BDD trees onto boolean functions, s.t.

– $I(x)(\sigma) = x$, if $x \in \mathbf{N}$ and
– $I((M, c_0, c_1))(\sigma) = M(I(c_0)(\sigma), I(c_1)(\sigma))\sigma$.

By induction, we show that $I(R_\nu(f)) = f$. The base case is when $R_\nu(f)$ is a constant x. In this case, f is a constant function returning x and so is $R_\nu(f)$. In the inductive case, let $R_\nu(f) = (M, c_0, c_1)$. For $i = 0, 1$, $c_i = R_{\nu i}(f_{\nu i})$. Hence, by induction $I(c_i) = f_{\nu i}$. [This induction is justified by the fact that, as the length of ν increases, eventually $S(\nu)$ becomes empty, making f_ν a constant function, which is the base case.] Therefore, $I(R_\nu(f))(\sigma) = M(f_{\nu 0}(\sigma), f_{\nu 1}(\sigma))\sigma$. By definition 9, this equals $f f^{f_{\nu 0}(\sigma)}_{\nu 0} f^{f_{\nu 1}(\sigma)}_{\nu 1}(\sigma)$. However, since $f^{f_{\nu i}(\sigma)}_{\nu i}$ is by definition in the same equivalence class with σ (for $i = 0, 1$), this is equal to $f(\sigma|_{S(\nu 0)})(\sigma|_{S(\nu 1)})\sigma = f(\sigma)$. Hence $I(R_\nu(f)) = f$, hence $I(\langle f \rangle) = I(R_\epsilon(f)) = f$. □

4 BDD tree algorithms

In this section, we set out algorithms that allow us to build to the concrete BDD tree representations for the boolean functions of circuits. The boolean functions can be generated from literals using a suitable set of basis functions (*i.e.*, gates). A literal is a boolean function that yields true for σ if a given variable v is true

If ν is a binary tree vertex, $S(\nu)$ is the set of variables assigned to any descendent of ν. In our example, $S(0) = \{a, b\}$, while $S(\epsilon) = \{a, b, c, d\}$. Now we define the lexical order on truth assignments that gives us the canonical numbering of equivalence classes:

Definition 4. For any functions $\sigma, \sigma' : V \to \{0, 1\}$, $\sigma < \sigma'$ exactly when there is some $1 \leq i \leq n$ such that $\sigma(v_i) = 0$ and $\sigma'(v_i) = 1$ and for all $1 \leq j < i$, $\sigma(v_j) = \sigma'(v_j)$.

This is just the ordinary lexical order on truth assignments expressed as boolean vectors. So, for example, $0010 < 0011$. Now we define what it means to *cofactor* a function by a partial truth assignment:

Definition 5. For any function $f : (V \to \{0, 1\}) \to \mathbb{N}$, any $V' \subseteq V$ and any $\sigma : V' \to \{0, 1\}$, let $f\sigma$ be the function $(V \setminus V' \to \{0, 1\}) \to \mathbb{N}$ such that $(f\sigma)(\rho) = f(\sigma \cup \rho)$.

In the example, suppose σ is a partial truth assignment that assigns 01 to ab. Then $f\sigma = 01 + cd = cd$.

Definition 6. For any function $f : (V \to \{0, 1\}) \to \mathbb{N}$, any vertex $\nu \in \{0, 1\}^*$ and any truth assignment σ to V, let $f_\nu(\sigma) = |\{f\sigma' \mid \sigma' \leq \sigma\}| - 1$.

The function f_ν is the child function at vertex ν of the tree. It yields the number of the equivalence class of a given truth assignment σ to $S(\nu)$. This is one less than the number of functions that can be obtained by cofactoring f by truth assignments to $S(\nu)$ that are lexically less than or equal to σ. In our example, $S(0) = \{a, b\}$. The equivalence classes of assignments to ab are, in lexical order, $\{00, 01, 10\}, \{11\}$. The first equivalence class corresponds to the cofactor cd, while the second corresponds to 1 (true). Thus, for example $f_0(01) = 0$ and $f_0(11) = 1$.

Definition 7. For any function $f : (V \to \{0, 1\}) \to \mathbb{N}$ and any $\nu \in \{0, 1\}^*$, let $|f|_\nu = |\{f\sigma \mid \sigma : S(\nu) \to \{0, 1\}\}|$.

The notation $|f|_\nu$ stands for the number of equivalence classes of f at vertex ν. In the example, $|f|_0 = 2$.

Definition 8. For any function $f : (V \to \{0, 1\}) \to \mathbb{N}$ and any $\nu \in \{0, 1\}^*$ and any $0 \leq j < |f|_\nu$, let f_ν^j be the lexically least σ, such that $f_\nu(\sigma) = j$.

Thus, f_ν^j is a representative of the jth equivalence class of assignments at vertex ν. That fact that is it the least in its class is immaterial. In the example, $f_0^0 = 00$ and $f_0^1 = 11$.

The function R_ν, defined below, yields the BDD tree representation of a function f at binary tree vertex ν. This representation is a triple, consisting of the root matrix, the left and the right child functions.

Definition 9. For any function $f : (V \to \{0, 1\}) \to \mathbb{N}$ and any $\nu \in \{0, 1\}^*$, let

- $R_\nu(f) = x$ if f is a constant function yielding x, for $x \in \mathbb{N}$,

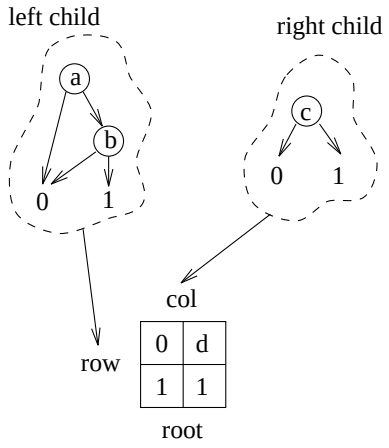

Fig. 2. BDD tree example (the functions in the matrix are also represented by OBDD's, though this representation is not shown).

torization can be done only when the assignments to the variables in question divide into exactly two equivalence classes modulo the function represented by the root of the subdag. In BDD trees, on the other hand, the number of equivalence classes is not limited, but only one (integer valued) function of the given variables can be used.

3 Formal definition of BDD trees

We now give a formal definition of the BDD tree representation, defining a function $\langle\rangle$ that maps each boolean function onto its BDD tree representation. We prove that this function is invertible. Hence two functions are equal exactly when their BDD tree representations are equal. Each definition will be illustrated using the example function, $f = ab + cd$.

Definition 1. Let $V = \{v_1, \ldots, v_n\}$ be an ordered set (of variables).

For our example function, let $V = \{a, b, c, d\}$.

Definition 2. Let t be a function $V \to \{0,1\}^*$.

This function defines the decomposition. The function t maps every variable to its "tree address", which is a boolean vector. We can think of the boolean vector as a vertex in a binary tree. If ν is a vertex of the tree, then $\nu 0$ is the left child of ν and $\nu 1$ is the right child of ν. The empty string ϵ is the root. In our example, $t(a) = t(b) = 0$, $t(c) = 1$ and $t(d) = \epsilon$. This is because a and b are assigned to the left child, c to the right child, and d to the root. If the left child in turn had a left child, its tree address would be 00.

Definition 3. For any $\nu \in \{0,1\}^*$ let $S(\nu)$ be the set

$$\{v \in V \mid \nu \text{ is a prefix of } t(v)\}.$$

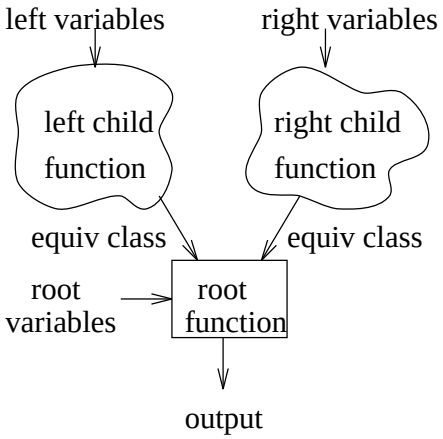

Fig. 1. Hierarchical decomposition of a function.

column represents the resulting function of the remaining variables. Since the child functions are numeric valued rather than boolean, and are also represented recursively using OBDD trees, we will generalize OBDD's slightly, using natural numbers as the OBDD leaves rather than booleans. In general, a *boolean function* in this paper will refer to a function that takes a truth assignment to some boolean variables and returns a natural number, rather than a truth value.

An example of the BDD tree representation is shown in figure 2. This BDD tree represents the boolean function $ab + cd$. The left child contains variables a and b, the right child variable c and the root variable d. Note that the truth assignments $\{00, 11, 10\}$ to ab fall into the same equivalence class, which is numbered 0. The remaining equivalence class, containing only 11 is numbered 1. This reflects the canonical numbering of the equivalence classes. We fix a total order on the boolean variables, in this case (a, b, c, d). This defines a lexical order on truth assignments. That is, two truth assignments are ordered based on the first variable in which they differ. This is the order in which the truth assignments would appear in a dictionary if represented as boolean vectors. The equivalence classes are numbered starting from zero, ordered by the least truth assignment they contain. Hence, the set $\{00, 01, 10\}$ comes before $\{11\}$.

BDD trees are, of course, closely related to OBDD's. For example, if we assign all variables to the root, we obtain exactly the OBDD for the given function. More subtly, if we assign exactly one variable to each left child (or right child), we get as structure that is very similar in form to an OBDD, or to a DFA recognizing the given truth function. The essential difference between OBDD's and BDD trees is that the former are based on a linear decomposition of a function, while the latter are based on a hierarchical decomposition. Karplus' if-then-else dags [Kar89] appear to have a similar structure, but in fact are quite different. In an if-then-else dag, interior subdags with only two exits are eliminated by factoring their variables into the "if" part of a single if-then-else node. This fac-

To use the BDD tree representation, one must provide a hierarchical decomposition of the boolean variables rather than the linear order required for ordered binary decision diagrams. This decomposition is easily obtained in the case of hierarchically structured circuits, since it simply follows from the modular structure of the circuit. Two examples are presented here of the use of BDD trees in verifying tree structured systems by symbolic model checking. In one case, the ability of BDD trees to capture redundancy in the system hierarchy results in logarithmic growth in the representation of the transition relation as the system is scaled up. Significant speedups are obtained over the OBDD method, even for small system sizes. In the other case, BDD trees are significantly better for representing the transition relation, but do not improve the performance of checking temporal logic formulas.

2 Informal description of BDD trees

A BDD tree is essentially a hierarchical decomposition of a discrete function. As an example of a hierarchical decomposition, suppose that we wish to represent a function of some set of boolean variables. We can divide the variables into three parts – those variables belonging to the *root* of the hierarchy, those belonging to the *left child* of the hierarchy, and those belonging to the *right child*. The truth assignments to the left child variables can be partitioned into equivalence classes. Two such partial truth assignments are in the same equivalence class if they are equivalent *cofactors* of our function – that is, if fixing the values of the left child variables according to the two truth assignments yields the same function of the remaining variables. These equivalence classes of truth assignments to the left child variables can be numbered in some canonical way. We can then define a discrete function for the left child of the hierarchy that takes a truth assignment and returns the number corresponding to its equivalence class. This left child function tells us exactly the information we need to know about the left child variables in order to evaluate the function. We can define a similar function for the right child variables. The root function then takes the result of the right and left child functions, and an assignment to the root variables, and returns the value of the original function. This decomposition of the function is depicted in figure 1.

Assuming we have fixed a canonical numbering of the equivalence classes, this hierarchical representation is canonical. That is, for each function we have a unique representation. The left and right child functions may themselves be represented hierarchically by dividing their variables into left, right and root functions recursively, until the child functions become constants (*i.e.*, functions of zero variables). Such a decomposition leaves us only with the question of how to represent the root function at each point in the hierarchy. For this, we will use *ordered binary decision diagrams* (OBDD's), in order to take advantage of the very simple and efficient algorithms available for manipulating them [Bry86]. To represent the root function, we will use a matrix of OBDD's, with one row for each left child equivalence class and one column for each right child equivalence class. Thus, the left child variables determine the row of the matrix, the right child variables determine the column, and the OBDD in the given row and

Hierarchical representations of discrete functions, with application to model checking

K. L. McMillan

AT&T Bell Laboratories
Murray Hill, NJ

Abstract. BDD trees provide a hierarchically structured canonical representation for boolean functions, based on ordered binary decision diagrams (OBDD's). We describe algorithms for function application and boolean quantification on BDD trees, allowing them to be used in applications such as symbolic model checking. Experimentally, we find that BDD trees can be greatly more efficient than ordinary OBDD's in verifying tree structured systems using symbolic model checking. In one case, sublinear growth is observed in the size of the transition relation representation. Analytically, we find that for a class of circuits of fixed tree width, BDD trees are asymptotically efficient.

1 Introduction

Binary decision diagrams have been widely used as a representation for boolean formulas in the verification of digital systems [Bry86]. They can be applied to the comparison of switching functions, or to the verification of temporal properties using a technique called symbolic model checking [BCM+90]. The sharing of substructure within BDD's allows them to efficiently capture and exploit certain kinds of regularity within the truth table of a boolean function. Heuristically, they provide a compact representation for most switching functions occurring in digital systems, given an appropriate total ordering on the boolean variables.

OBDD's are asymptotically efficient for any class of circuits of fixed *path width*. The path width is the optimum wiring channel width for any linear arrangement of the circuit elements. For such a class of circuits, the size of the optimum OBDD representing the circuit's function is linearly bounded in the size of the circuit. Tree width is the analog of path width for tree structured arrangements. For fixed tree width, the optimum OBDD size is polynomial in the circuit size, but of degree that is exponential in the tree width [McM92].

This paper proposes a canonical representation that is *linear* in the tree width and like OBDD's, has quadratic algorithms for applying logical operations. It is based on the notion of a hierarchical decomposition of a discrete function. The representation uses a generalized form of binary decision diagrams as part of its structure. For this reason, we will refer to representations of this type as *BDD trees*. In some sense, BDD trees are a natural generalization of OBDD's from linear structure to tree structure. In fact, one can draw an analogy from BDD trees to deterministic finite tree automata in much the same way one can from OBDD's to ordinary DFA's.