Efficient Parallel Binary Decision Diagram Construction Using Cilk
by
David B. Berman

Submitted to the
Department of Electrical Engineering and Computer Science

May 22, 2000

In Partial Fulfillment of the Requirements for the Degree of
Bachelor of Science in Computer Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

Binary Decision Diagrams are theoretical data structures used for formal verification of protocols and digital circuits. Previously, parallel computing algorithms designed to create these structures have experienced limited success due to their inability to exploit the inherent parallelism in the binary decision diagram creation problem. The CilkBDD algorithm uses Cilk, a C-based multithreaded language, to expose this parallelism, which this paper contends is chiefly limited by the size of the BDD being created. The algorithm incorporates hash tables and data caches into basic BDD manipulation algorithms. Despite causing an increase in computation time over previous algorithms, this approach results in a more direct software system, and thus a more optimal parallel approach, with less memory and processor overhead. Ultimately, CilkBDD is able to use multiple processors 10-33% more effectively than other parallel BDD algorithms, and parallelism statistics indicate that as more processors are applied, the processor usage continues to increase.

Thesis Supervisor: Charles Leiserson
Title: Head, Supercomputing Technologies Group, MIT Lab for Computer Science
1. Introduction

A fully reduced and ordered Binary Decision Diagram (BDD) is a unique representation of a Boolean expression [1]. This attribute allows BDDs to be used for formal verification of protocols and comparison of digital circuits [4], just to name two applications. Recently, a BDD-manipulation algorithm called “Partial Breadth-First” (PBF) [10, 11] has been developed that uses parallel computation to reduce Boolean expressions to BDDs efficiently by applying working set control to minimize thrashing and other performance-reducing occurrences. When it was released, PBF significantly improved upon the efficiency of previous BDD algorithms and was the first such algorithm to operate on parallel processors. PBF, while offering a performance improvement over its predecessors, opened a new door for further improving BDD algorithms: the exploitation of parallelism. The algorithm presented in this thesis, CilkBDD, expands upon the work done by PBF by demonstrating how parallelization is the next frontier for BDD algorithms to conquer.

Binary Decision Diagram creation is a problem that inherently contains much parallelism. The approach taken by the creators of PBF only began to utilize this parallelism to create BDDs efficiently. Many of the system problems associated with parallelism were handled by PBF using intricate and complex coding techniques. While effective in resolving such parallelism-related problems as message passing, shared memory handling, and work scheduling, the approach of PBF is also quite complex and involved a lot of overhead, leaving significant room for improvement.
In fact, while PBF achieves at most a 4-times speedup on an 8-
processor machine (implying an inherent parallelism of 4 in the PBF
algorithm), my research reveals that the parallelism in BDD creation is
virtually unlimited, solely depending on the size of the BDD being
created. The parallelism data provided by the PBF authors is
inconclusive, and in fact, it proves difficult to obtain definitive
statistics on the relative performance of PBF and CilkBDD in parallel
processing environments. On functions equivalent in size to those that
produced the largest amount of parallelism for PBF, however, execution
of my algorithm demonstrates that the parallelism in reducing these
functions to BDDs is actually two or three orders of magnitude larger
than what has been previously achieved. This paper explains how my
algorithm, CilkBDD, uses a simpler BDD algorithm than PBF and combines
it with the power of the C-based language Cilk [9] in order to achieve
such a large amount of parallelism.

The serial performance analysis in this thesis indicates that not
only is PBF faster than CilkBDD, its running time and memory usage
scales better with the size of the problem. When analyzing parallel
algorithms, however, a comparison of serial running times does not yield
sufficient information about performance. Fundamentally, parallelism is
a more important efficiency metric than CPU time when dealing with
enormous problems. One is now almost always willing to use more memory
in order to achieve better running time, since memory is inexpensive and
running time is critical. Similarly, the clients that use BDDs for
formal verification frequently possess fast processors at their disposal
since they are, in fact, using BDDs to design faster processors. If the
parallelism does not exist in a problem, however, one can apply as many
processing resources as one wants to solve it without decreasing the
overall running time. Therefore, a critical result from this paper that should influence future work is the amount of parallelism found in the BDD creation problem. Additionally, the parallel results indicate that CilkBDD does not fully exploit the parallelism that the results indicate is available, and we provide suggestions for how higher levels of parallelism might be achieved in the future.

This thesis focuses on a new algorithm for creating BDDs from Boolean functions. The rest of this thesis describes how the algorithm exposes the inherent parallelism of the problem and uses Cilk to exploit it. Section 2 describes the theory and structure behind BDDs and BDD operations. Section 3 presents improvements on the simple BDD creation algorithm, using caching and hash tables to remove redundancy. Section 4 discusses the parallel algorithm used in PBF and how one can improve upon it. Section 5 describes the Cilk language and its uses in parallel programming. Section 6 quantifies the uses of Cilk in BDD algorithms and its effectiveness. Section 7 presents ideas for further work and resulting conclusions.

2. BDD Background

Understanding the theory behind Binary Decision Diagrams is a prerequisite to analyzing the algorithms that create them. This section explains the critical definitions and theory behind BDDs that enable them to be useful in real-life applications. Section 2.1 explains the fundamental problems of determining if two Boolean expressions are equivalent and computing if a Boolean expression is satisfiable. These problems provide the motivation for introducing the binary decision diagram. Section 2.2 presents the definition of a binary decision
diagrams and explains the process for creating BDDs from Boolean expressions. This section concludes by enumerating two critical properties of BDDs that make them particularly useful: ordering and being reduced.

2.1 Boolean Expressions

A Boolean expression is a combination of Boolean variables that employs Boolean operators according to the following grammar:

\[
\text{<EXPRESSION> = VARIABLE | 0 | 1 | ~<EXPRESSION> | <EXPRESSION> ∩ <EXPRESSION> | <EXPRESSION> ∪ <EXPRESSION>,}
\]

where VARIABLE can be any single variable from a complete set of Boolean variables. An expression yields a truth value (0 or 1, representing FALSE or TRUE, respectively) for every different element from the set of assignments of truth values to its variables. Two Boolean expressions are equivalent if they yield the same truth value for every possible assignment of values to their variables.

Equivalence of Boolean Expressions

To determine all the possible truth values of an expression, one can simply instantiate all N variables with every possible set of truth values. To do so, however, one must perform \(2^N\) expression evaluations, since there are N independent variables, each with 2 possible different values. Each expression evaluation runs in linear time in the number of variables, since we need to perform an assignment to each of the N
variables and then reduce expressions according to standard truth tables for the NOT (¬), AND (∧), and OR (∨) operators. We perform \(2^N\) evaluations, and since each evaluation takes \(O(N)\) time, we have a running time on finding the set of truth values for an expression over all of its possible variable assignments bounded by \(O(N \times 2^N)\).

Evaluating the equivalence of two Boolean expressions in \(N\) variables is simple once we have obtained the set of truth values for each expression over all their possible variables assignments. For each individual assignment of values to variables, we compare the truth values of each expression. The expressions are equal if they are equivalent over all possible assignments; otherwise, they are not. The comparison stage requires checking each of the \(2^N\) possible assignments, which is an insignificant quantity compared to the two \(O(N \times 2^N)\) operations we must perform on the expressions in order to create the assignment list. We have described a simple \(O(N \times 2^N)\) algorithm for determining the equivalence of Boolean expressions, but we can improve the entire algorithm by optimizing both the creation of the truth value set and the comparison method.

**Satisfiability of Boolean Expressions**

A Boolean expression is satisfiable if there exists some assignment of truth values to its variables that results in the expression evaluating to true (1). In fact, if we do compute the set of truth values over all possible variable assignments for an expression, then we can determine if the expression is satisfiable, since we can simply search the \(2^n\) assignments for one that yields a true expression.
value. Determining the equivalence of Boolean expressions therefore also determines the satisfiability of the expressions. Cook's theorem \[8\] states that satisfiability of Boolean expressions is NP-complete. Therefore, Boolean satisfiability can be solved in exponential time, but it is highly unlikely that it can be solved in polynomial time. We have achieved an $O(N \times 2^N)$ brute-force algorithm for determining satisfiability (and equivalence), but maybe we can reduce the equivalence problem to a simple exponential algorithm. In fact, we can, which motivates the introduction of a new construct: the binary decision diagram (BDD) \[1, 5\].

### 2.2. Binary Decision Diagrams

Binary Decision Diagrams can eliminate much of the work required in evaluating and comparing Boolean expressions. It is useful to build an understanding of BDDs by beginning with basic operations and then describing crucial BDD properties. This section starts by enumerating definitions that are relevant to the creation of BDDs: the "if-then-else" operator, and the Shannon expansion of an expression. Using these definitions, we then provide an example Boolean expression and incrementally develop a compact representation for it. Finally, we define a Binary Decision Diagram, and enumerate two critical properties that enable us to use BDDs to uniquely represent Boolean expressions.

**Boolean Function Representation**

We let $(t \rightarrow t_0, t_1)$ represent the "if-then-else" operator. Logically, this expression means that if $t$ is true, then the expression takes on the value of $t_0$, and otherwise it evaluates to $t_1$. We can
express all operators easily using the \( \rightarrow \) operator and the constants 0 and 1. For example, \( x \) is \((x \rightarrow 1, 0)\), \(-x\) is \((x \rightarrow 0, 1)\), and \(x_1 \land x_2\) is \((x_1 \rightarrow x_2, 0)\).

We denote the Boolean expression obtained by replacing all instances of the variable \( x \) in the expression \( t \) with a Boolean value by \( t[x = v \in \{0,1\}] \). We can then see that \( t = (x \rightarrow t[x = 1], t[x = 0]) \), which is known as the Shannon expansion of \( t \) with respect to \( x \).

Using the Shannon expansion, we can now develop a method for converting any Boolean expression into one composed of solely 0's, 1's, and "if-then-else" operators, and in which all of the tests are performed on single variables. We recursively apply the Shannon expansion to an expression \( t \) composed of \( n \) variables. If \( n = 0 \) then \( t \) is equivalent to either 0 or 1, and we have an expression in the proper form. Otherwise, we take the first variable, \( x_0 \), and apply the expansion of \( t \) with respect to \( x_0 \). As seen above, this process yields \( t = (x_0 \rightarrow t[x_0 = 1], t[x_0 = 0]) \). We now need to evaluate \( t[x_0 = 1] \) and \( t[x_0 = 0] \), each of which contains \( n - 1 \) variables, since \( x_0 \) has been replaced in each of the \( t \) expressions. We can do so by applying the Shannon expansion to each of these expressions with respect to the next variable, say \( x_1 \). Since we reduce the number of variables by 1 each time, we eventually arrive at the base case of \( n = 0 \).

We can simplify our representation of an expression \( t \) in which a set of variables \( \{x_0, x_1, \ldots, x_i\} \) have each been instantiated (as either 0 or 1) from, for example, \( t[x_0 = 1, x_1 = 0, x_2 = 1, \ldots, x_i = 0] \) to
Once we have completed the Shannon expansion of an entire expression, we have a set of expressions that look like the following:

\[
\begin{align*}
t &= (x_0 \rightarrow t_1, t_0) \\
t_1 &= (x_1 \rightarrow t_{11}, t_{10}) \\
t_0 &= (x_1 \rightarrow t_{01}, t_{00}) \\
t_{11} &= (x_2 \rightarrow t_{111}, t_{110}) \\
\ldots
\end{align*}
\]

The base cases consist of at most \(2^n\) expressions in which all \(N\) variables have been instantiated, resulting in an expression we can evaluate to 0 or 1. This set of expressions can be turned into a binary tree in which each node contains the single variable we are testing. The leaves are the expressions with that variable instantiated, which are themselves binary trees. All of the leaves are fully-instantiated expressions, and so are constant nodes. This tree is called a decision tree. An example of a set of expressions and the resulting tree for \(t = (x_1 \land x_2) \lor ((\neg x_3) \land x_2)\) follows. In this example expansion, we have not simplified any of the expressions, and so the tree is quite general until we reach the leaves.

\[
\begin{align*}
t &= (x_1 \rightarrow t_1, t_0) \\
t_1 &= (x_2 \rightarrow t_{11}, t_{10}) \\
t_0 &= (x_2 \rightarrow t_{01}, t_{00}) \\
t_{11} &= (x_3 \rightarrow t_{111}, t_{110}) \\
t_{10} &= (x_3 \rightarrow t_{101}, t_{100}) \\
t_{01} &= (x_3 \rightarrow t_{011}, t_{010}) \\
t_{00} &= (x_3 \rightarrow t_{001}, t_{000}) \\
t_{111} &= (x_4 \rightarrow 1, 1) \\
t_{110} &= (x_4 \rightarrow 1, 1) \\
t_{101} &= (x_4 \rightarrow 0, 0) \\
t_{100} &= (x_4 \rightarrow 1, 0) \\
t_{011} &= (x_4 \rightarrow 0, 0) \\
t_{010} &= (x_4 \rightarrow 1, 0) \\
t_{001} &= (x_4 \rightarrow 0, 0) \\
t_{000} &= (x_4 \rightarrow 1, 0)
\end{align*}
\]
In fact, aside from ordering the way in which we instantiate the variables, we have not accomplished anything yet. We still have \(2^N\) evaluations to do, each of which requires instantiating all \(N\) variables. We can perform some simplifications on this set of expressions, however. In particular, if we have an expansion in which \((x_i \rightarrow t, t)\), the entire expression can be simplified to \(t\), since the resulting expression does not depend on the value of \(x_i\). If we apply this simplification wherever possible in the preceding example, we obtain the following expressions:

\[
\begin{align*}
t &= (x_1 \rightarrow t_1, t_0) \\
t_1 &= (x_2 \rightarrow 1, t_10) \\
t_0 &= (x_2 \rightarrow t_{01}, t_{00}) \\
t_{10} &= (x_3 \rightarrow 0, t_{100}) \\
t_{01} &= (x_3 \rightarrow 0, t_{010}) \\
t_{00} &= (x_3 \rightarrow 0, t_{000}) \\
t_{100} &= (x_4 \rightarrow 1, 0) \\
t_{010} &= (x_4 \rightarrow 1, 0) \\
t_{000} &= (x_4 \rightarrow 1, 0)
\end{align*}
\]

We can now convert these expressions into a binary decision tree, in which each node represents a variable. Each node has two branches, the right branch representing the expression if the variable at the node is set to true (1), and the left representing the expression if the variable at the node is set to false (0). The decision tree for the example expression \(((X_1 \land X_2) \lor ((-X_3) \land X_4))\) is depicted in Figure 1.

Looking at the tree and the expression list, we see many repeated subexpressions. It makes sense to replace all of the identical expressions with just one copy of the expression. This common subexpression elimination cuts down on the size of the tree, and it may
also enable us to cut down on the number of expansions we perform during the expansion phase, as we shall see later.

Figure 1:

If we apply this idea of eliminating repeated subexpressions, we find that \( t_{100} \), \( t_{010} \), and \( t_{000} \) are identical, and so we can replace all instances of \( t_{010} \) and \( t_{000} \) in the expressions with \( t_{100} \). This transformation then makes \( t_{10} \), \( t_{01} \), and \( t_{00} \) identical, and so we can replace all instances of \( t_{01} \) and \( t_{00} \) with \( t_{10} \). We now notice that \( t_{0} = (x_{2} \rightarrow t_{10}, t_{10}) \), which we can simplify to just \( t_{0} = t_{10} \), as we did before. We end up with the following greatly compacted set of expressions:
\begin{align*}
  t &= (x_1 \rightarrow t_1, t_{10}) \\
  t_1 &= (x_2 \rightarrow 1, t_{10}) \\
  t_{10} &= (x_3 \rightarrow 0, t_{100}) \\
  t_{100} &= (x_4 \rightarrow 1, 0)
\end{align*}

We have now identified all equal subexpressions of the original expression. If we make a binary decision tree from these expressions, we finally have our binary decision diagram (BDD), shown in Figure 2.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure2.png}
\caption{A BDD is a directed acyclic graph (DAG), in which each node has either two or zero children. Every node in a BDD represents a subexpression of the original expression. A node with zero children (a leaf of the tree) is a terminal node and represents either 0 or 1. Nonleaf nodes are therefore nonterminal. Observe that in the original decision tree, we had 9 nonterminal nodes, whereas we have only 4 in the corresponding BDD.}
\end{figure}
Ordered BDDs

While BDDs can be powerful tools for analyzing Boolean expressions, they can be made more useful with a few additional constraints. The first of these is ORDERING. A BDD is ordered (an OBDD) if the variables occur in the same order on all paths from the root of the BDD to the leaves. One can imagine that we might choose to instantiate variables in a different order depending on the result of previous instantiations, but this policy adds complexity to the representation and makes it difficult to perform reductions. The BDD in figure 2 is ordered.

An ordered binary decision diagram is said to be reduced (ROBDD) if it meets the following conditions in addition to those required to make it a valid OBDD:

1) No individual node appears more than once in the OBDD. That is, for each different variable $x_1, \ldots, x_n$, every node that instantiates that variable must have different left and right children than all of the other nodes of that variable.

2) No redundant tests exist in the OBDD. If a node's left and right children are identical, then the entire node can be eliminated from the final BDD.

The final BDD in Figure 2 is reduced.

ROBDDs have many interesting and useful properties, the most useful of which is canonicity. The canonicity lemma states that for any
Boolean function $f$ of $n$ Boolean variables, there is exactly one ROBDD with a given variable ordering $(x_1 < \ldots < x_n)$ representing it.

This canonicity lemma has many important implications. For one, we can now check in constant time if an ROBDD is constantly true or false, since there can only be one expression for a Boolean expression that evaluates to a constant: a constant node. Additionally, we can compare two Boolean expressions for equivalence by creating a ROBDD for each and then seeing if the results are identical. If they are, then the Boolean expressions must be equivalent; otherwise they are not equal. Now, we see that the comparison of ROBDDs for determining Boolean expression equivalence can be done in constant time. To truly make BDDs useful in practice, however, we need to develop an efficient algorithm for creating an ROBDD from a given Boolean expression.

3. BDD Algorithm Improvements

We seek to develop an efficient method of creating binary decision diagrams from Boolean functions. We have already devised one method of creating an ROBDD: We first impose an ordering on the variables and recursively expand the expression using the Shannon expansion method, described on page eight. This method results in an OBDD, which we can then reduce by eliminating common subexpressions and redundant nodes. A better approach, however, is to reduce the BDD as we construct it.

This section outlines the necessary data structures and the algorithm CilkBDD uses to construct BDDs. Section 3.1 explains why a hash table is necessary to preserve the "reduced" property required of a BDD, and then describes a simple yet inefficient algorithm that uses a hash table to construct a BDD from a Boolean expression. Section 3.2
begins with an explanation of how caching can be applied to the BDD construction problem to improve its efficiency. The section continues by providing code for the procedures Sift and Apply, which form the majority of the CilkBDD algorithm and both utilize the described caching method. Finally, with Sift and Apply as specific examples, implementation details of the caches in CilkBDD are presented.

3.1 Hash Table

The first tool we need is a node hash table to eliminate redundancy. As explained in Section 2, BDDs are not unique if they contain any repeated nodes, and thus a hash table is required to provide a guarantee on the absence of any redundancies. This section presents a short description of how a hash table can be used in a BDD construction algorithm and then provides code that implements such an algorithm. The section concludes with an analysis of how the use of the hash table affects the algorithm’s performance.

When creating a new node to insert into the BDD, we can first check against the hash table to see if we have already created a node for the same variable with the same children. If we are attempting to create a new node that is identical to one that already exists, we can simply return the old value in the table. We can write a simple procedure to create a new node that performs a table lookup before returning the new node. If we always use this procedure whenever we want to insert a new node into the BDD, we can maintain the invariant that the BDD is reduced. The newNode function takes as arguments the variable index of the node to be created and the right and left children of the
resulting node. The function returns a newly created node with the specified index and children.

```c
newNode (index, left, right) {
    if (left == right)
        return left;
    //This enforces the second ROBDD constraint:
    //no redundant nodes
    n = tableLookup (index, left, right);
    //Enforces first ROBDD constraint:
    //no repeated subexpressions
    if (n != NULL)
        return n;
    n = new Node (index, left, right);
    tableInsert (index, left, right, n);
    return n;
}
```

We now have the capabilities to create a simple ROBDD construction algorithm. The function Construct takes in a Boolean expression to reduce to a BDD, and the index of the next variable to remove from the expression. When initially called with $i = 0$, it returns the ROBDD that represents the passed to it.

```c
Construct (expression, i) { expression
    if (i > n)
        //n is a global variable containing the
        //number of variables in expression
        if (expression is false)
            return FALSE_NODE;
        else
            return TRUE_NODE;
    left = Construct (expression [xi = 0], i + 1);
    right = Construct (expression [xi = 1], i + 1);
    return newNode (i, left, right);
}
```

A call to Construct (expression, 1), in which expression has $n$ variables, requires $2^n$ calls to Construct. Additionally, at the base case, we need to evaluate the "(expression is false)" clause, which runs
in $O(n)$ time, since we have $n$ instantiated variables. This algorithm yields the same total running time of $O(N \cdot 2^n)$, as does the algorithm from Section 2, but it reduces the total storage size necessary, since at no point need we ever store an unreduced BDD to reduce later.

3.2. Caching to avoid repeated subproblems

To further improve upon our BDD construction algorithm, the next step is to cache the results of subproblems. This section first explains how caching can be applied to the creation of BDDs, using a recursive procedure called Apply. We then present code for Apply, which uses Shannon expansion to recursively performs a Boolean operation on two BDDs and returns the result. The caching methodology is extended further, for use in the next procedure we describe, Sift. The functionality of Sift is described, and then code for it is provided. Sift is used to remove operator nodes from a tree by having Apply perform the operator node’s operation on its children. The section concludes by analyzing how the both caches are implemented in CilkBDD.

In instantiating the values of all of the expressions at the base cases of the call to Construct, we typically evaluate subexpressions multiple times. If we can cache the expression evaluations, then we can avoid doing any more work in evaluating expressions than we have to. We use a new cache called the bddCache: We store a resulting ROBDD node in the cache based on its operation and two operands. Because we have the hash table to avoid redundancy, we guarantee that there is a one-to-one mapping between pointer values and the actual BDD representation the pointers point to. Thus, our cache mapping from an operation and operands to a node is simple: We take the pointer value of the first
operand, shift it four bits to the right, and add the shifted value to
the pointer value of the second operand. Then, we add a 0, 1, or 2 to
the result, depending on if the operation is a NOT, AND, or XOR.
Finally, we obtain the index of the proper cache row by taking the
result modulo the size of the cache.

Apply

We develop a new function, called Apply, which takes a Boolean
operation and applies it to two ROBDDs. It returns a single ROBDD that
represents the result when the Boolean operation has been applied to the
expressions represented by bdd1 and bdd2.

Apply (operation, bdd1, bdd2) → Node {
    n = bddCacheLookup (operation, bdd1, bdd2);
    if (n != NULL)
        return n;
    if ((bdd1 is a constant node) && (bdd2 is a constant node))
        n = operation (bdd1, bdd2);
    //For comparison, constant nodes are given the maximal index
    else if (index (bdd1) == index (bdd2))
        n = newNode (index (left),
            Apply (operation, left_branch (bdd1),
                left_branch (bdd2)),
            Apply (operation, right_branch (bdd1),
                right_branch (bdd2)));
    else if (index (bdd1) < index (bdd2))
        n = newNode (index (bdd1),
            Apply (operation, left_branch (bdd1), bdd2),
            Apply (operation, right_branch (bdd1), bdd2));
    else
        n = newNode (index (bdd2),
            Apply (operation, bdd1, left_branch (bdd2)),
            Apply (operation, bdd1, right_branch (bdd2)));

    bddCacheInsert (operation, bdd1, bdd2, n);
    return n;
}
The size of the bddCache can be adjusted in order to tune the performance of the algorithm. For now, a direct-mapped cache works quite well, as long as its size is scaled with the number of bddNodes we create. Unlike the hash table, the bddCache is not required for correctness of the implementation. Were we to eliminate the bddCache, we would still get the same results, yet it would take much longer, as many redundant operations would need to be performed.

**Sift**

Our Apply procedure can combine two BDDs using a Boolean operator, but the tree that represents the original Boolean function (the operator tree, or OpTree) only contains BDDs at the leaves. It is necessary to develop a function that calls Apply on trees containing no operator nodes after the function has removed them. This function is called Sift, as it must sift the operator nodes out through the bottom of the tree by repeatedly calling Apply from the bottom up.

In fact, there is no reason not to use a cache here as well. Although Apply is typically called more often and is likely to perform repeated operations that can be cached, Sift is also a candidate for optimization using a cache. We call this new cache the "opCache" and use another scalable, direct-mapped cache. In practice, this cache is usually optimally effective when it is a couple of orders of magnitude smaller than the bddCache. Sift takes the root of a tree of operator nodes as its argument, and returns the ROBDD that represents the expression the operator tree describes.
Impact of Caches

These two caches combine to increase the efficiency of BDD construction significantly. Of course, the performance increase generated depends highly on the number, size, and closeness in time of occurrence of the redundant operations that are required by the particular BDD to be created. CilkBDD allows the user to vary the size of the caches independently, since different types of expressions exhibit varying amounts of redundancy and therefore benefit from problem-specific "fine-tuning". In general, it is practical to use as much of the available physical memory as possible for the caches, without spilling over into virtual memory. Section 6 contains specific examples and statistics comparing the relative size and performance of the caches.
4. Previous Parallel BDD Work

This section presents the history behind the development of CilkBDD. First, we discuss the motivation behind introducing parallelism into BDD construction algorithms. Then, the latest and most successful parallel algorithm in the literature, Partial Breadth-First, is described. The experimental results obtained by the algorithm's authors are reported, and the hybrid style of the algorithm is described. Partial Breadth-First combines the advantages of two popular BDD approaches, depth-first and breadth-first, to achieve speedups of 1.6 over previous algorithms, while also employing parallelism. The section concludes with a discussion of how the hybrid approach leads to the use of context switching to improve memory allocation and to introduce parallelism into the Partial Breadth-First algorithm.

Binary Decision Diagrams have been used as a tool in formal verification with increasing frequency over the past decade. As circuit diagrams became more complex, BDDs emerged as the preeminent representation due to their unique and compact representation of Boolean expressions. More recently, as circuits have continued to grow in complexity, their BDD representations have increased in size as well. Thus, the BDD construction problem has become more complex as well. To combat this complexity, parallel BDD algorithms provide an attractive weapon.

Thus far, the most successful parallel BDD construction algorithm is called "Partial Breadth-First Expansion" or "PBF", developed in 1997 by Bwolen Yang and David O'Hallaron at Carnegie-Mellon University [10, 11]. This algorithm uses a combination of programming techniques and
intelligent auxiliary data structures to achieve speedups of 1.6 or
greater over previous algorithms when executed on a serial processor.
PBF does support parallelism, though, and exhibits over 2 times speedup
on a 4-processor machine, and up to 4 times speedup on an 8-processor
machine. These results are based on experimental data that Yang and
O'Hallaron collected using the ISCAS85 benchmarks, a set of ten circuits
used in industry [3, 7]. They ran the experiments on an SGI Power
Challenge with 1 gigabyte of physical memory and twelve 196MHz
processors. For the serial tests, they used only one of the processors.

The algorithm PBF uses is quite complex. In general, the idea is
to improve on the two different prevailing approaches by combining the
best features of both. Depth-first algorithms seek to traverse the
entire tree and sift out operator nodes from the bottom up. This
approach is straightforward, but it exhibits poor memory locality,
because accesses to nodes that are physically close in the operator tree
are scattered in time. When the operator tree does not fit in physical
memory, depth-first algorithms must spend significant time swapping
pages in and out of memory. In contrast, breadth-first algorithms
exhibit much better memory locality. This type of approach seeks to sift
out all operator nodes at a given level of the tree at once, keeping
around in memory the operations necessary to reduce lower levels of the
tree until later. Therefore, despite a better memory access pattern,
breadth-first has much more memory overhead than depth-first approaches,
which only need to maintain the current depth of the recursive calls in
memory.

PBF's approach combines the advantages of both breadth-first and
depth-first into a hybrid algorithm. PBF performs breadth-first sifting
up to a fixed memory threshold, and then it performs a context switch to begin a depth-first approach. This method explicitly limits the amount of physical memory needed, allowing the caches to exploit memory locality within a given context.

PBF implements parallelism by putting the idea of the context switch to further use. Because it provides the functionality to allow different sections of the operator tree to be worked on independently, it can permit different processors to take different contexts and work on them simultaneously. In fact, what occurs is that work for a given context is placed in two queues that fundamentally correspond to a Sift queue and an Apply queue. These queues also allow the multiple processors to balance their load by "stealing" work from another context's queue when one processor becomes idle.

5. Cilk and CilkBDD

The Cilk language is integral to CilkBDD's performance in a parallel processing environment. This section provides a short description of Cilk and explains why it is used to solve the BDD construction problem. A comparison is presented between CilkBDD use of the natural Cilk concept of work and work scheduling, which is based on function calls, and PBF's explicit work discretization and forced context switching. The section concludes with a description of other features of Cilk and how they are specifically beneficial to CilkBDD: Cilk provides functionality for reporting parallelism, a native locking mechanism, and semantics that result in simpler code structure.
Cilk is a language for multithreaded parallel programming. The Supercomputing Technologies Group developed it at the Massachusetts Institute of Technology’s Laboratory for Computer Science [9]. Cilk is based on ANSI C and is designed to allow the user to take any serial C program (the serial "elison" of the parallel program) and parallelize it by inserting a few Cilk constructs. The programmer’s task is to expose parallelism and reference locality, while allowing the Cilk runtime system to schedule the tasks to run efficiently on a given platform.

In particular, Cilk seems suited to the Binary Decision Diagram construction problem. While PBF exploits parallelism in the problem to simultaneously work on significant "chunks" of the operator tree, it requires a significant amount of software framework to achieve this parallelism. In particular, PBF goes to great lengths to minimize costly context switches, schedule work and implement work-stealing, implement message passing and locking between threads, and allocate memory in an intelligent and efficient manner. Cilk abstracts the client away from many of these issues, leaving programmers free to expose the inherent parallelism in the BDD problem.

Cilk provides functionality that PBF was forced to implement as part of its software package. One of PBF’s great strengths is its implementation of operator queues that allow work-stealing and thrash-minimizing context switching. Cilk implicitly discretizes work into calls declared as "Cilk functions", in this case Sift and Apply. Cilk thus abstracts the problem of implementing explicit work queues for both scheduling and stealing away from the user. Furthermore, Cilk guarantees that its task scheduling is nearly optimal. Whereas PBF explicitly defines evaluation threshold and forces context switches, Cilk
dynamically schedules work so the programmer does not need to worry about such issues.

An additional advantage of using Cilk is its predefined functionality. While it is necessary to use locking to avoid race conditions (since the hash table and caches are kept in shared memory and repeatedly accessed and modified), Cilk provides a native locking mechanism. Also significant is its ability to report the parallelism of a given task. As stated above, in BDD problems, parallelism is a critical performance metric. We can discover the inherent amount of parallelism in any BDD construction problem simply by asking the Cilk-version of the BDD algorithm to report that statistic. Then, theoretically, we know how many processors can be usefully applied to the problem.

The CilkBDD implementation thus becomes similar to the Sift and Apply algorithm described above in Section 2. Cilk requires that the procedures to be executed in parallel are defined as "cilk" procedures, but in fact, the only such procedures in CilkBDD are Sift and Apply. Modifying the original algorithm for parallel computing only requires putting Cilk locks around the caches and synchronizing the calls to Sift and Apply when we need the results from those calls to those functions. In order to minimize the amount of time spent waiting for locks to be released, each cache row and hash table bin has its own lock. The Cilk version of Sift, CilkSift, serves as an example of the modifications necessary to transform a normal C procedure into a Cilk procedure. Assuming we have a similarly modified procedure, CilkApply, CilkSift is functionally identical to Sift, except now it can operate in a parallel environment.
CilkSift (optree) → Node {
    n = opCacheSearch (operation (optree),
                         left_branch (optree),
                         right_branch (optree));
    if (n != NULL)
        return n;
    if (isOpNode (left_branch (optree)))
        left = spawn CilkSift (left_branch (optree));
    else
        left = left_branch (optree);
    if (isOpNode (right_branch (optree)))
        right = spawn CilkSift (right_branch (optree));
    else
        right = right_branch (optree);
    sync;
    n = spawn CilkApply (operation (optree),
                         left, right);
    sync;
    opCacheInsert (operation (optree),
                   left_branch (optree),
                   right_branch (optree));
}

As a practical advantage, CilkBDD code is much simpler than PBF. A major disadvantage of using PBF is that the code is quite complex, requiring many different packages and intricate data structures just to implement the parallelism framework. That framework does not exist as part of the CilkBDD code, for it is entirely contained within the language of Cilk. The new BDD code is more compact and is a straightforward implementation of the general Sift-and-Apply algorithm described above. The few Cilk constructs that are necessary are just simple extensions to C. The CilkBDD approach is to remove as many extraneous data structures that existed in PBF for the explicit performance of decreasing its running time. While this approach effects an overall performance hit when the metric is processing time, it removes a significant amount of overall system complexity from the BDD
algorithm. These simplifications allow us to utilize multiple processors approximately 10-33% more efficiently, but they result in a 2-4 times sacrifice of efficiency, and poorer scaling properties. Section 6 provides detailed, quantitative data on the results of simpler, Cilk-based programming.

6. Performance results

This section provides detailed statistics on the performance of CilkBDD. First, the parallelism performance metric used in this analysis is presented, as simply looking at the running time of the algorithm is inconclusive and does not adequately describe CilkBDD’s performance. The test cases and testing environment is described, and then three different tables of test data are included. The first table contains the results of the comparison between PBF and CilkBDD on a serial machine, which indicate that PBF runs faster and scales better, but does not exhibit nearly as much parallelism as CilkBDD. The second table displays results when both algorithms were executed on a computer using multiple processors, showing how CilkBDD uses a larger proportion of parallel processing power than PBF does. The third and final table contains data indicating how increasing the sizes of the caches and hash table up to a certain point can benefit the processing time, parallelism, and memory usage of CilkBDD. These tables will be analyzed in Section 7.

Ultimately, the measure of any algorithm is how it performs in practice. In the case of BDDs, there is no singular simple way to measure performance. It is of course critical to look at the actual CPU time used in creating any BDD. One can think of the parallelism of a problem, however, as a metric that measures the “potential” of an
algorithm. Cilk allows us to execute a program on a serial machine but still calculate the parallelism of the running task. This data is critical to the analysis of the CilkBDD algorithm and is included side-by-side with the running time of each trial in the performance data that will be presented. Because PBF is not written in Cilk, it does not provide statistics on parallelism. Because the PBF trials described in Section 5 reveal that the maximum speedup ever achieved by PBF was 4, however, we can calculate the projected processor utilization efficiency of PBF and compare it to CilkBDD. This efficiency, computed as the relative speedup divided by the number of processors, declines for PBF as the number of processors increase. On an 8-processor machine, PBF's reported speedup was 4, yielding a processor utilization efficiency of .50.

There are two different sets of performance data here. One set was compiled by running both PBF and CilkBDD on a single processor machine: A Sun Ultra 1 running SunOS 5.6, with 64 megabytes of memory and a 143-MHz processor. The parallel processing data was compiled on MIT's Pleiades Alpha 4100 Cluster. The experiments run used four Alpha 21164 466-MHz processors, with 512 megabytes of combined shared memory.

The serial data is presented first. CilkBDD does support limited "trace-reading" capabilities that allow it to import benchmark circuits for performance evaluation. Both PBF and CilkBDD provide more comprehensive and significant results on adder and multiplier circuits. (Actually, one of the most studied circuits in the ISCAS85 set is a multiplier circuit.) Adder circuits are excellent examples of small Boolean expressions that exhibit much redundancy. The size of the adder circuit grows linearly with the number of input bits it has, however,
and so it is quite limited in its ability to test the performance thresholds and parallel operation of either algorithm. Multiplier circuit representations grow exponentially with the number of input bits, and thus make for excellent performance test cases on both serial and parallel machines. In addition, due to the complexity of multiplier circuits, they exhibit redundancy that is distributed well throughout the circuit, so that large multipliers push the performance of the caches as well.

Serial Performance Data:
(All data has been averaged over multiple trials to reduce the impact of performance anomalies.)

<table>
<thead>
<tr>
<th>Circuit Name</th>
<th>Algorithm</th>
<th>CPU Time (s)</th>
<th>Memory Usage (Mb)</th>
<th>Parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td>adder-2</td>
<td>CilkBDD</td>
<td>0.002</td>
<td>0.036</td>
<td>1.22</td>
</tr>
<tr>
<td>adder-2</td>
<td>PBF</td>
<td>0.010</td>
<td>0.250</td>
<td>n/a</td>
</tr>
<tr>
<td>adder-10</td>
<td>CilkBDD</td>
<td>0.006</td>
<td>0.055</td>
<td>1.62</td>
</tr>
<tr>
<td>adder-10</td>
<td>PBF</td>
<td>0.010</td>
<td>1.020</td>
<td>n/a</td>
</tr>
<tr>
<td>adder-50</td>
<td>CilkBDD</td>
<td>0.026</td>
<td>0.151</td>
<td>1.84</td>
</tr>
<tr>
<td>adder-50</td>
<td>PBF</td>
<td>0.050</td>
<td>4.440</td>
<td>n/a</td>
</tr>
<tr>
<td>adder-100</td>
<td>CilkBDD</td>
<td>0.051</td>
<td>0.271</td>
<td>1.90</td>
</tr>
<tr>
<td>adder-100</td>
<td>PBF</td>
<td>0.100</td>
<td>8.880</td>
<td>n/a</td>
</tr>
<tr>
<td>mult-2</td>
<td>CilkBDD</td>
<td>0.003</td>
<td>0.035</td>
<td>1.19</td>
</tr>
<tr>
<td>mult-2</td>
<td>PBF</td>
<td>0.010</td>
<td>0.380</td>
<td>n/a</td>
</tr>
<tr>
<td>mult-5</td>
<td>CilkBDD</td>
<td>0.065</td>
<td>0.274</td>
<td>10.34</td>
</tr>
<tr>
<td>mult-5</td>
<td>PBF</td>
<td>0.040</td>
<td>2.330</td>
<td>n/a</td>
</tr>
<tr>
<td>mult-6</td>
<td>CilkBDD</td>
<td>0.213</td>
<td>0.817</td>
<td>21.10</td>
</tr>
<tr>
<td>mult-6</td>
<td>PBF</td>
<td>0.070</td>
<td>3.600</td>
<td>n/a</td>
</tr>
<tr>
<td>mult-7</td>
<td>CilkBDD</td>
<td>0.686</td>
<td>2.567</td>
<td>34.31</td>
</tr>
<tr>
<td>mult-7</td>
<td>PBF</td>
<td>0.170</td>
<td>4.770</td>
<td>n/a</td>
</tr>
<tr>
<td>mult-8</td>
<td>CilkBDD</td>
<td>2.123</td>
<td>7.621</td>
<td>47.70</td>
</tr>
<tr>
<td>mult-8</td>
<td>PBF</td>
<td>0.450</td>
<td>5.480</td>
<td>n/a</td>
</tr>
</tbody>
</table>

Parallel Performance Data:
(algorithm circuit not implemented in parallel by PBF)

<table>
<thead>
<tr>
<th>Algorithm/Circuit Name</th>
<th>Processors</th>
<th>Speedup over CPU Time</th>
<th>Memory Usage (Mb)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PBF/mult-2</td>
<td>1</td>
<td>1.00</td>
<td>0.77</td>
</tr>
<tr>
<td>PBF/mult-5</td>
<td>1</td>
<td>1.00</td>
<td>4.63</td>
</tr>
<tr>
<td>PBF/mult-6</td>
<td>1</td>
<td>1.00</td>
<td>7.74</td>
</tr>
<tr>
<td>PBF/mult-7</td>
<td>1</td>
<td>0.86</td>
<td>10.45</td>
</tr>
<tr>
<td>PBF/mult-8</td>
<td>1</td>
<td>0.80</td>
<td>11.55</td>
</tr>
<tr>
<td>PBF/mult-9</td>
<td>1</td>
<td>0.91</td>
<td>17.12</td>
</tr>
<tr>
<td>PBF/mult-10</td>
<td>1</td>
<td>0.95</td>
<td>20.80</td>
</tr>
<tr>
<td>PBF/mult-11</td>
<td>1</td>
<td>0.99</td>
<td>30.66</td>
</tr>
<tr>
<td>PBF/mult-12</td>
<td>1</td>
<td>1.00</td>
<td>54.66</td>
</tr>
<tr>
<td>Algorithm/Circuit Name</td>
<td>Processors</td>
<td>Speedup over CPU Time</td>
<td>Memory Usage (Mb)</td>
</tr>
<tr>
<td>------------------------</td>
<td>------------</td>
<td>-----------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>PBF/mult-2</td>
<td>2</td>
<td>1.00</td>
<td>0.91</td>
</tr>
<tr>
<td>PBF/mult-5</td>
<td>2</td>
<td>1.00</td>
<td>5.20</td>
</tr>
<tr>
<td>PBF/mult-6</td>
<td>2</td>
<td>1.00</td>
<td>7.83</td>
</tr>
<tr>
<td>PBF/mult-7</td>
<td>2</td>
<td>0.66</td>
<td>10.57</td>
</tr>
<tr>
<td>PBF/mult-8</td>
<td>2</td>
<td>0.80</td>
<td>11.67</td>
</tr>
<tr>
<td>PBF/mult-9</td>
<td>2</td>
<td>0.95</td>
<td>18.88</td>
</tr>
<tr>
<td>PBF/mult-10</td>
<td>2</td>
<td>1.18</td>
<td>22.93</td>
</tr>
<tr>
<td>PBF/mult-11</td>
<td>2</td>
<td>1.45</td>
<td>37.20</td>
</tr>
<tr>
<td>PBF/mult-12</td>
<td>2</td>
<td>1.42</td>
<td>66.23</td>
</tr>
<tr>
<td>PBF/mult-2</td>
<td>4</td>
<td>1.00</td>
<td>1.08</td>
</tr>
<tr>
<td>PBF/mult-5</td>
<td>4</td>
<td>1.00</td>
<td>5.23</td>
</tr>
<tr>
<td>PBF/mult-6</td>
<td>4</td>
<td>1.00</td>
<td>7.87</td>
</tr>
<tr>
<td>PBF/mult-7</td>
<td>4</td>
<td>0.66</td>
<td>10.62</td>
</tr>
<tr>
<td>PBF/mult-8</td>
<td>4</td>
<td>0.80</td>
<td>12.95</td>
</tr>
<tr>
<td>PBF/mult-9</td>
<td>4</td>
<td>0.95</td>
<td>20.80</td>
</tr>
<tr>
<td>PBF/mult-10</td>
<td>4</td>
<td>1.23</td>
<td>25.27</td>
</tr>
<tr>
<td>PBF/mult-11</td>
<td>4</td>
<td>1.66</td>
<td>45.10</td>
</tr>
<tr>
<td>PBF/mult-12</td>
<td>4</td>
<td>2.03</td>
<td>80.24</td>
</tr>
<tr>
<td>CilkBDD/mult-2</td>
<td>1</td>
<td>1.00</td>
<td>0.054</td>
</tr>
<tr>
<td>CilkBDD/mult-5</td>
<td>1</td>
<td>0.66</td>
<td>0.433</td>
</tr>
<tr>
<td>CilkBDD/mult-6</td>
<td>1</td>
<td>0.66</td>
<td>1.452</td>
</tr>
<tr>
<td>CilkBDD/mult-7</td>
<td>1</td>
<td>0.68</td>
<td>4.129</td>
</tr>
<tr>
<td>CilkBDD/mult-8</td>
<td>1</td>
<td>0.66</td>
<td>12.644</td>
</tr>
<tr>
<td>CilkBDD/mult-9</td>
<td>1</td>
<td>0.68</td>
<td>39.671</td>
</tr>
<tr>
<td>CilkBDD/mult-2</td>
<td>2</td>
<td>1.00</td>
<td>0.054</td>
</tr>
<tr>
<td>CilkBDD/mult-5</td>
<td>2</td>
<td>1.56</td>
<td>0.653</td>
</tr>
<tr>
<td>CilkBDD/mult-6</td>
<td>2</td>
<td>1.58</td>
<td>2.165</td>
</tr>
<tr>
<td>CilkBDD/mult-7</td>
<td>2</td>
<td>1.52</td>
<td>6.551</td>
</tr>
<tr>
<td>CilkBDD/mult-8</td>
<td>2</td>
<td>1.53</td>
<td>19.490</td>
</tr>
<tr>
<td>CilkBDD/mult-9</td>
<td>2</td>
<td>1.57</td>
<td>60.995</td>
</tr>
<tr>
<td>CilkBDD/mult-2</td>
<td>4</td>
<td>1.00</td>
<td>0.054</td>
</tr>
<tr>
<td>CilkBDD/mult-5</td>
<td>4</td>
<td>2.66</td>
<td>0.611</td>
</tr>
<tr>
<td>CilkBDD/mult-6</td>
<td>4</td>
<td>2.78</td>
<td>1.946</td>
</tr>
<tr>
<td>CilkBDD/mult-7</td>
<td>4</td>
<td>2.80</td>
<td>6.972</td>
</tr>
<tr>
<td>CilkBDD/mult-8</td>
<td>4</td>
<td>2.96</td>
<td>15.183</td>
</tr>
<tr>
<td>CilkBDD/mult-9</td>
<td>4</td>
<td>2.33</td>
<td>52.128</td>
</tr>
</tbody>
</table>

As discussed in Section 3.2.3, the size of the caches and hash table affects the performance of the algorithm quite heavily. CilkBDD allows the user to specify the size of the hash table, bddCache, and op cache in order to tune the performance for the specific circuit being constructed. The statistics below demonstrate how adjustment of the cache sizes can affect all three performance metrics used to analyze the algorithm.
Cache Analysis Data:
(Hash table and caches size values are actually the logarithm (base 2) of the size in bytes. The same example circuit is used for all analysis: a 7-bit multiplier.)

<table>
<thead>
<tr>
<th>Hash table size</th>
<th>Bdd cache size</th>
<th>Op cache size</th>
<th>CPU Time</th>
<th>Memory Usage</th>
<th>Parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>11</td>
<td>11</td>
<td>1.110</td>
<td>2.004</td>
<td>59.626</td>
</tr>
<tr>
<td>7</td>
<td>11</td>
<td>11</td>
<td>0.917</td>
<td>2.019</td>
<td>61.759</td>
</tr>
<tr>
<td>8</td>
<td>11</td>
<td>11</td>
<td>0.874</td>
<td>2.029</td>
<td>58.162</td>
</tr>
<tr>
<td>9</td>
<td>11</td>
<td>11</td>
<td>0.795</td>
<td>2.057</td>
<td>55.725</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>11</td>
<td>0.775</td>
<td>2.114</td>
<td>53.516</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>11</td>
<td>0.773</td>
<td>2.223</td>
<td>50.427</td>
</tr>
<tr>
<td>12</td>
<td>11</td>
<td>11</td>
<td>0.774</td>
<td>2.442</td>
<td>44.053</td>
</tr>
<tr>
<td>11</td>
<td>12</td>
<td>11</td>
<td>0.736</td>
<td>2.137</td>
<td>46.191</td>
</tr>
<tr>
<td>11</td>
<td>13</td>
<td>11</td>
<td>0.717</td>
<td>2.102</td>
<td>42.395</td>
</tr>
<tr>
<td>11</td>
<td>14</td>
<td>11</td>
<td>0.691</td>
<td>2.117</td>
<td>41.616</td>
</tr>
<tr>
<td>11</td>
<td>15</td>
<td>11</td>
<td>0.685</td>
<td>2.222</td>
<td>37.532</td>
</tr>
<tr>
<td>11</td>
<td>16</td>
<td>11</td>
<td>0.683</td>
<td>2.457</td>
<td>32.257</td>
</tr>
<tr>
<td>11</td>
<td>17</td>
<td>11</td>
<td>0.690</td>
<td>2.951</td>
<td>24.557</td>
</tr>
<tr>
<td>11</td>
<td>16</td>
<td>9</td>
<td>0.700</td>
<td>2.446</td>
<td>32.831</td>
</tr>
<tr>
<td>11</td>
<td>16</td>
<td>10</td>
<td>0.687</td>
<td>2.449</td>
<td>33.759</td>
</tr>
<tr>
<td>11</td>
<td>16</td>
<td>12</td>
<td>0.682</td>
<td>2.473</td>
<td>31.501</td>
</tr>
<tr>
<td>11</td>
<td>16</td>
<td>13</td>
<td>0.687</td>
<td>2.504</td>
<td>32.044</td>
</tr>
<tr>
<td>11</td>
<td>16</td>
<td>14</td>
<td>0.691</td>
<td>2.567</td>
<td>30.671</td>
</tr>
</tbody>
</table>

7. Analysis and conclusion

This final section provides an analysis of the results obtained in Section 6 and offers conclusions about the effectiveness of CilkBDD and PBF. Section 7.1 looks at the first two tables in Section 6 and provides explanations for the data obtained in both the serial and parallel processing cases. Section 7.2 presents suggestions for further improvements in future implementations of BDD construction algorithms. Finally, Section 7.3 offers conclusions about the effectiveness of CilkBDD and the implications of the experimental data and analysis.

7.1 Algorithm performance analysis

We first look at the serial algorithm data and provide explanations for the results. The serial data indicates that PBF has better performance statistics than CilkBDD and scales better with the size of the BDD being constructed, but CilkBDD exhibits impressive
parallelism numbers. An explanation is offered of how PBF's running
time-tailed data structures allow it to complete its work more quickly
than CilkBDD, but inhibit its overall parallelism. Next, an analysis of
the parallel algorithms is presented, based on the second table in
Section 6. The parallel statistics indicate that CilkBDD makes better
use of the available processors than PBF does. The parallel analysis
section looks closely at these numbers and postulates how future BDD
construction implementations might improve upon these results by
incorporating features of both algorithms. In particular, the simplicity
of CilkBDD allows for more parallelism, but PBF's complex data
structures might be necessary to take full advantage of this
parallelism. To more thoroughly understand how to incorporate
parallelism into BDD construction algorithms, more work is necessary.

Serial algorithm analysis

Improvements to CilkBDD's serial algorithm are a starting point
for improving the performance of the entire parallel system. One of the
main advantages of the serial algorithm is its simplicity. As a
straightforward implementation of the "sift-and-apply" BDD construction
algorithm, it is clear and concise. The only extraneous data structures
it uses are the two caches, which add little complexity and greatly
improve performance. PBF incorporates many auxiliary data structures
which increase the memory usage greatly, but which also reduce the
necessary computation time.

PBF's work queues allow it to postpone work until it is most
convenient to complete it. In contrast, when there is only one
processor, CilkBDD is essentially a depth-first algorithm, and thus
exhibits all of the deficiencies of depth-first algorithms in general, namely lack of memory locality. CilkBDD must first proceed down the entire tree, sifting out all of the operator nodes. Only then can it proceed back up the tree, applying the operators to the reduced nodes. Work queues would allow CilkBDD to "push" operator nodes down the tree without reducing their operands. We could begin from the top-down, pushing operator nodes and applying them, keeping pending Applies at the child nodes until we got to that location. This approach, while significantly more complex, would increase the locality of the node accesses, since on the way down the tree, once we process a node, we never need to return to it. Of course, using work queues might significantly reduce the parallelism, since they create bottlenecks by reducing how much work can be performed at once. Moreover, work queues require more locking to prevent multiple processors from altering the queues at the same time.

Both serial algorithms are similar in performance and memory usage until the circuit gets significantly large. At this point, PBF's performance overtakes CilkBDD's, largely due to its more intelligent allocation and management of memory. PBF implements a mark-and-sweep garbage collection algorithm, whereas CilkBDD does not implement garbage collection. Were CilkBDD to use less memory, it would be able to spend less time page swapping on large diagrams. Even when the hash tables and caches are set at the size that is optimal or performance, however, the majority of memory (> 66%) is taken up by bddNodes during the "sift-and-apply" process, not the hash tables or caches.

The way to reduce the memory used by the algorithm is to free unused nodes during construction, but this task is not trivial. The
CilkBDD algorithm requires that no redundant nodes exist in the final BDD, a property that the use of a node hash table enforces. One could implement garbage collection by keeping reference counts on nodes and freeing them as soon as they have no references to them. Such an approach would compromise the critical "no redundancy" property, however, as the defining characteristic of a node's uniqueness is its pointer value. Were we to free a node's record that was created by combining two other still "in use" nodes, we would have no way of recreating the reference to those node should we need them again. Because BDDs themselves are compact representations of functions, it is difficult to maintain node data that we can use to recreate the composition of a BDD when it becomes "live" again. Thus, the simplest approach, and the one most conducive to parallelization, is to keep a record of every BDD node ever created in the hash table.

**Parallel algorithm analysis**

In many cases of both PBF and CilkBDD, executions on parallel processors actually took longer to complete than their serial counterparts, presumably due to processor overhead. In some cases, CilkBDD's performance was quite slow, when it had to wait for cache locks to be released. The creation of CilkBDD was fundamentally an exercise in revealing how much potential parallelism exists in the BDD problem, however, and it succeeded at that. For both PBF and CilkBDD, in the serial executions, the recorded CPU Time and Elapsed Time were within ten percent of each other. By comparing the CPU Time and Elapsed Time numbers from the parallel executions, we arrive at the data in the "speedup" column of the table in Section 6. This number is a large indication of how much work the algorithm was able to accomplish in
parallel, and gives us a reasonable indication of the amount of parallelism in both PBF and CilkBDD, since PBF does not provide such data in its statistics output.

The parallel implementation of CilkBDD is far from optimal. One potential area of optimization is even finer-grain locking in the hash table and caches, so that we do not have to lock on every read from the tables. Another option could be to allow each processor to have its own caches, to minimize data races between processors trying to access those structures. Parallel computation can introduce problems into the CilkBDD algorithm for the case in which a processor begins work on a certain part of the operator tree. Another processor cannot directly access that part of the tree, but it could get a result from a Sift cache lookup that points it to the same piece of the tree that the first processor is working on. In this case, both processors may be trying to simultaneously Sift and Apply in the same part of the operator tree, wasting work, and possibly overwriting each other. A solution to this race condition is to implement locking on every node in the tree, but in practice, this approach takes up valuable memory and significantly increases the amount of time waiting for locks. In future implementations, cleaner partitioning of the operator tree between different processors should be investigated.

7.2 Other possible improvements

Despite its less superior CPU performance statistics, CilkBDD can be considered to be a significant improvement over PBF, due to its somewhat more conservative memory allocation and enormously increased parallelism. In fact, the results are quite inconclusive, but point to
many different approaches that future implementations should investigate. The data in Section 6 and the subsequent analysis illustrate the large amount of parallelism that is inherent to the BDD problem and is not currently being exploited. The fundamental question that remains is how to eliminate the slowdowns that prevent the actual parallel speedup from being equivalent to the parallelism of the problem. In this section, we discuss different aspects of CilkBDD and PBF that maybe limiting performance. First, we discuss propose theories on how to reduce the redundancy in order to achieve more parallelism in future implementations. Additionally, we present some experimental data that suggests the introduction of some of PBF's work reordering data structures will be beneficial to CilkBDD's performance. Finally, we discuss the specifics of the implementation of the caches and hash table. Then, the analysis turns to proposing and supporting the theory that a large amount of redundant work is being created by the parallel algorithm.

What areas of the algorithm might be prohibiting CilkBDD from reaching its potential parallelism? One hypothesis is that the processors may be duplicating each other's work. This phenomenon can occur if one processor begins sifting out an operator node. Since we can have multiple references to an operator node in the operator tree, another processor could begin working on the same node before the first processor has finished. The first processor finishes, and puts its result in the opCache but the second processor continues to work on the same problem, as it had already checked the opCache when it began, and found no entry for the node. The second processor completes, and moves on to the next problem, but it has not done any useful work, even though it has been processing the whole time. Such situations would not be
reflected in the serial parallelism statistics, but when CilkBDD is run in parallel, it could actually be creating more work than in the serial case, thus mitigating the beneficial effects of the algorithm's measured parallelism.

To support the theory that CilkBDD is creating redundant work, we take measurements what procedures are being executed and when. First, we keep a count of the number of processors working on sifting out an operator node. When a call to Sift begins and does not return early due to an opCache hit, we increment the counter of the number of processors working on that node. The counter is decremented upon returning from the Sift call, but not before noting how many processors were working on the node. If the counter was greater than 1, we mark the call as being "useless". Otherwise, the work was useful. By tallying all of the "useless" Sift calls, we find that on 2 processors, in the mult-5 case, approximately 58% of the calls were useful, but that number steadily increases as we add more bits, until we reach approximately 80% usefulness at mult-8. Additionally, by using Cilk's provided work measurement statistics, we see that in the 2-processor cases, approximately 50% more work is being done overall. While this analysis certainly supports the theory that much redundant work is being performed, it is still not quite clear what exactly is going on. Why do the larger circuits seem to perform more useful work when in fact the data indicates that CilkBDD operators at approximately the same processor utilization efficiency on all test circuits?

To enhance our understanding of where exactly the redundant work is being created, we take even more statistical readings during the execution of CilkBDD. Perhaps the work is being wasted by Apply, and not
Sift, since in a typical execution, Apply is called 100 times more often than Sift. By measuring the amount of work done in calls to Apply, we discover that with an alarming amount of consistency, 10% of the total work done by CilkBDD is performed in Apply, no matter how many processors we are using. Additionally, by performing the same "usefulness" analysis on calls to Apply, we see that the percentage of useless calls is never above 4%. These statistics indicate that the redundancies are probably not created in Apply, which makes intuitive sense: Calls to Apply are performed at the bottom of the tree, and on much smaller nodes than Sift calls. Therefore, the likelihood that two processors are working on the same bddNode in Apply calls is much less.

The Apply analysis illustrates how the number of calls to a procedure does not necessarily indicate how much work is performed by those calls. We augment the "usefulness" analysis of the Sift calls to now take into account the size of the opNode the Sift call is working on. The size is simply the sum of the size of the opNode's two children, plus 1 for the parent node itself. This measure provides information about how much work the call to Sift must perform, since it eventually sifts through the entire size of the parent node. The new data reveals that regardless of the number of useful Sift calls, the percentage of total size of the nodes that are sifted by useful Sift calls is consistent. For the 2-processor case, approximately 55% of the size is operated on by useful Sifts, and that number lowers to 37% for the 4-processor case. When we have larger circuits, we have more calls to Sift, but the calls follow the structure of the tree. Thus, we have many more smaller calls at the base of the operator tree, which are less likely to overlap. We also have more large calls near the root of the
tree, however, and these are the calls that are more likely to be executed redundantly, thereby decreasing efficiency.

Now that we have likely identified the problem that is reducing CilkBDD’s empirical parallelism, what approaches can we take to combat it? One possible solution is to lock all operator nodes when they are being sifted. In practice, this approach greatly reduces the parallelism, as in many cases the processors cannot afford to wait for another processor to finish a large piece of work, and the work becomes serialized. Another approach is to randomize the order of calls to child processes. We can randomly alternate between sifting a node’s right child and left child first, hoping to minimize the chance that two processors will follow the same execution path. In practice, however, randomization does not yield a significant and consistent reduction in the number of useless operations performed. Because the redundant Sift calls are the large ones, local task scheduling within procedures is not able to avoid spawning redundant work.

The experiments indicate that a much more involved and global method of task scheduling is required to take full advantage of the parallelism. PBF implements such a method, using work queues and context switching to force the processors to work on a particular area of the operator tree at a particular time. This approach is worth investigating, but in PBF’s implementation, the queues themselves become parallelism bottlenecks and sources of complexity. Alternatively, CilkBDD’s “blind parallelism” approach, which just allows the multiple recursive calls to schedule the work, is too simple to prevent redundancy. Future approaches should seek a happy medium, a task
scheduling method that reduces or eliminates redundant calls while not causing a significant loss of parallelism.

Even though the experiments described strongly support the hypothesis that redundancy is heavily limiting performance, we must look at additional areas of CilkBDD that can be improved. The data in Section 6 points to the caches and hash tables as areas of the system that are critical to its efficiency. Due to their dramatic effect of their size on system performance, it is worthwhile to investigate if there are ways of further increasing the beneficial effects that the caches have on CilkBDD’s performance. One area of improvement could be the function that we use to map nodes to their location in the cache and hash table storage. If our function does not exhibit enough randomness, we can have many nodes fighting for the same entries in the cache table, and we must “boot” nodes unnecessarily. The current mapping function, for both the Apply and Sift caches, involves bit-shifting the right operand pointer four places to the right, and adding it to the pointer value of the left operand.

For the hash table, we used resolution by chaining, and we attempted to minimize the average list length for each hash bin. While it was easy to tune the performance such that the maximum average list length for any variable’s hash table was less than 2, if we used a different resolution mechanism, we might not have needed to make the hash tables so large in order to reduce the access time. Another approach, such as storing the nodes in a heap inside each bin, might increase the complexity, but could allow for more hash bin collisions, and thus reduce the overall size of the hash table. Still, the hash table itself, as compared to the caches, compromises a small portion of
the used memory, and the computational overhead involved with implementing a heap, or even an unbalanced binary tree at each hash bin, would most likely not be worth the effort.

By analyzing the cache hit rate statistics (accessible as an option from the command line) while modifying the mapping function, the optimal cache hit percentage for both caches was obtained by using a bit-shift of 4. For all multiplier circuits, the optimal hit rate for the apply cache was approximately 42%, and for the Sift cache it was 23%. Modifications to the number of bits being shifted, or incorporation of more operations on the pointers only had the impact of either requiring much larger caches to achieve the optimal hit rate, or sometimes never achieving that rate at all. It might be feasible to develop a mapping function that distributes the nodes more evenly about the cache, or it might be worthwhile to investigate using associativity in the caches in the future, but the data indicates that they function quite well as is.

How does the cache size affect the parallelism? Intuitively, it makes sense to allocate as much space as possible to the caches, since if they are larger, they are not forced to "boot" as many nodes. The data shows that at some point, however, enlargement of the cache actually becomes a detriment to performance. Large caches consume much memory, and if cache accesses actually require the memory to page, the entire system, including the parallelism, is affected. It is more beneficial to tune the cache size so that the maximum hit rate is achieved, and no larger. Additionally, the Sift cache is naturally smaller than the Apply cache (in the multiplier circuits, approximately 1/100 the size at optimality), but misses in the Sift cache lead to more
accesses to the apply cache. *Sift* cache hits are less frequent (23% vs. 42%), but since calls to *Sift* eventually must all be resolved in the leaves by calls to *Apply*, a *Sift* cache hit preempts many apply cache lookups. The *Sift* and *Apply* cache sizes must be tuned in conjunction with each other, as a reduction in one cache may be compensated for by an enlargement of the other. Still, the limitations of the multiplier circuit seem to be a 23%-*Sift* and 42%-*Apply* hit rate, independent of cache size, so if we reach both rates with minimal memory usage, there is no reason to make further adjustments.

### 7.3 Conclusion

The next step in the evolution of Binary Decision Diagrams is parallelization. CilkBDD, while not necessarily an optimal implementation of a parallel BDD algorithm, certainly demonstrates the enormous parallelism contained in BDD-construction problems. While PBF seems to operate approximately 2-3 times as fast as CilkBDD, its reported parallelism levels off around 4. In contrast, in serial trials, CilkBDD's parallelism seems to grow with the size of the problem, implying that the algorithm is not the limiting factor in the amount of available parallelism. In practice, using 4 processors, PBF still runs faster than CilkBDD, but it does not exhibit as much parallelism. Trials on 2 processors exhibit a 1-1.5 times speedup, and 1.5-2 times on 4 processors. For CilkBDD, 2-processor trials give around a 1.6 times speedup, and 4 processors yield a 2- to nearly 3 times speedup.

Despite the fact that they have been in use for over ten years now, there are still many areas in which BDD algorithms can be improved. Caches and hash tables provide considerable run-time improvements, but
their implementations are far from optimal as well. Our performance results are far from conclusive about what methods and algorithms are superior, but they do clearly indicate a necessity for increasing the utilization of available processors. Our subsequent analysis points towards the further investigation of specific methods for affecting such an increase. For BDD technology to continue to be useful on the larger types of circuits that are now necessary for industry to verify, existing algorithms must begin to exhibit increased parallelism. CilkBDD demonstrates that with intelligent application of data structures and algorithms like those seen in PBF, that parallelism can be exposed and exploited in the future.

References


