Parallel Factorization of Boolean Polynomials*

Vadiraj Kulkarni[1], Pavel Emelyanov[2,3], Denis Ponomaryov[2,3], Madhava Krishna[1,4], Soumyendu Raha[1], and S K Nandy[1]

[1] Computer Aided Design Laboratory, Indian Institute of Science, Bangalore 560012
{vadirajk,madhava,raha,nandy}@iisc.ac.in
[2] Ershov Institute of Informatics Systems, Lavrentiev av. 6, 630090, Novosibirsk, Russia
[3] Novosibirsk State University, Pirogova st. 1, 630090, Novosibirsk, Russia
{emelyanov,ponom}@iis.nsk.su
[4] Morphing Machines Pvt. Ltd

Abstract. Polynomial factorization is a classical algorithmic problem in algebra, which has a wide range of applications. Of special interest is factorization over finite fields, among which the field of order two is probably the most important one due to the relationship to Boolean functions. In particular, factorization of Boolean polynomials corresponds to decomposition of Boolean functions given in the Algebraic Normal Form. It has been also shown that factorization provides a solution to decomposition of functions given in the full DNF (i.e., by a truth table), for positive DNFs, and for cartesian decomposition of relational datatables. These applications show the importance of developing fast and practical factorization algorithms. In the paper, we consider some recently proposed polynomial time factorization algorithms for Boolean polynomials and describe a parallel MIMD implementation thereof, which exploits both the task and data level parallelism. We report on an experimental evaluation, which has been conducted on logic circuit synthesis benchmarks and synthetic polynomials, and show that our implementation significantly improves the efficiency of factorization. Finally, we report on the performance benefits obtained from a parallel algorithm when executed on a massively parallel many core architecture (Redefine).

Keywords: Boolean Polynomials · Factorization · Reconfigurable Computing.

1 Introduction

Polynomial factorization is a classical algorithmic problem in algebra, [8], which has numerous important applications. An instance of this problem, which deserves a particular attention, is factorization of Boolean polynomials, i.e., multilinear polynomials over the finite field of order 2. A Boolean polynomial is one

* This work was supported by the grant of Russian Foundation for Basic Research No. 17-51-45125 and by the Ministry of Science and Education of the Russian Federation under the 5-100 Excellence Program.

of the well-known sum-of-product representations of Boolean functions known as Zhegalkine polynomials [14] in the mathematical logic or the Reed–Muller canonical form [10] in the circuit synthesis. The advantage of this form that has recently made it popular again is a more natural and compact representation of some classes of Boolean functions (e.g., arithmetical functions, coders/cyphers, etc.), a more natural mapping to some circuit technologies (FPGA–based and nanostructure–based electronics), and good testability properties.

Factorization of Boolean polynomials is a particular case of decomposition (so–called disjoint conjunctive or AND–decomposition) of Boolean functions. Indeed, in a Boolean polynomial each variable has degree at most 1, which makes the factors have disjoint variables: $F(X,Y) = F_1(X) \cdot F_2(Y)$, $X \cap Y = \varnothing$.

It has been recently shown [4, 5] that factorizaton of Boolean polynomials provides a solution to conjunctive decomposition of functions given in the full DNF (i.e., by a truth table) and for positive DNFs without the need of (inefficient) transformation between the representations. Besides, it provides a method for Cartesian decomposition of relational datatables [3, 6], i.e., finding tables such that their unordered Cartesian product gives the source table. We give some illustrating examples below.

Consider the following DNF

$$\varphi = (x \wedge u) \ \vee \ (x \wedge v) \ \vee \ (y \wedge u) \ \vee \ (y \wedge v) \vee (x \wedge u \wedge v)$$

It is equivalent to

$$\psi = (x \wedge u) \ \vee \ (x \wedge v) \ \vee \ (y \wedge u) \ \vee \ (y \wedge v)$$

since the last term in φ is redundant. One can see that

$$\psi \equiv (x \vee y) \wedge (u \vee v)$$

and the decomposition components $x \vee y$ and $u \vee v$ can be recovered from the factors of the polynomial

$$F_\psi = xu + xv + yu + yv = (x + y) \cdot (u + v)$$

constructed for ψ.

The following full DNF

$$\varphi = (x \wedge \neg y \wedge u \wedge \neg v) \vee (x \wedge \neg y \wedge \neg u \wedge v) \vee \\ \vee (\neg x \wedge y \wedge u \wedge \neg v) \vee (\neg x \wedge y \wedge \neg u \wedge v)$$

is equivalent to

$$(x \wedge \neg y) \vee (\neg x \wedge y) \bigwedge (u \wedge \neg v) \vee (\neg u \wedge v)$$

and the decomposition components of φ can be recovered from the factors of the polynomial

$$F_\varphi = x\bar{y}u\bar{v} + x\bar{y}\bar{u}v + \bar{x}yu\bar{v} + \bar{x}y\bar{u}v = (x\bar{y} + \bar{x}y) \cdot (u\bar{v} + \bar{u}v) \tag{1}$$

constructed for φ.

Finally, Cartesian decomposition of the following table

B	E	D	A	C
z	q	u	x	y
y	q	u	x	y
y	r	v	x	z
z	r	v	x	z
y	p	u	x	x
z	p	u	x	x

$=$

A	B
x	y
x	z

$\times$

C	D	E
x	u	p
y	u	q
z	v	r

can be obtained from the factors of the polynomial

$$\begin{aligned} &z_B \cdot q \cdot u \cdot x_A \cdot y_C + \; y_B \cdot q \cdot u \cdot x_A \cdot y_C + \\ &y_B \cdot r \cdot v \cdot x_A \cdot z_C + \; z_B \cdot r \cdot v \cdot x_A \cdot z_C + \\ &y_B \cdot p \cdot u \cdot x_A \cdot x_C + z_B \cdot p \cdot u \cdot x_A \cdot x_C = \\ &\qquad = (x_A \cdot y_B + x_A \cdot z_B) \cdot (q \cdot u \cdot y_C + r \cdot v \cdot z_C + p \cdot u \cdot x_C) \end{aligned}$$

constructed for the table's content.

Decomposition facilitates finding a more compact representation of Boolean functions and data tables, which is applied in the scope of the Logic Circuit Synthesis, self-organizing databases, and dependency mining, respectively. Due to the typically large inputs in these tasks, it is important to develop efficient and practical factorization algorithms for Boolean polynomials.

In [13], Shpilka and Volkovich showed a connection between polynomial factorization and identity testing. It follows from their results that a Boolean polynomial can be factored in time $O(l^3)$, where l is the size of the polynomial given as a symbol sequence. The approach employs multiplication of polynomials obtained from the input one, which is a costly operation in case of large inputs. In [4], Emelyanov and Ponomaryov proposed an alternative approach to factorization and showed that it can be done without explicit multiplication of Boolean polynomials. The approach has been further discussed in [7].

In this paper, we propose a parallel version of the decomposition algorithm from [4, 7]. In Section 2, we revisit the sequential factorization algorithm from these papers. In Section 3, we describe a parallel MIMD implementation of the algorithm and further in Section 4 we perform a quantitative analysis of the parallel algorithm versus the sequential one. Finally, in Section 5 we evaluate our algorithm on a massively parallel many core architecture (Redefine) and outline the results.

2 Background

In this section we reproduce the sequential algorithm from [4, 7] for the ease of exposition. Let us first introduce basic definitions and notations.

A polynomial $F \in \mathbb{F}_2[x_1, \ldots, x_n]$ is called *factorable* if $F = F_1 \cdot \ldots \cdot F_k$, where $k \geq 2$ and $F_1, \ldots, F_k$ are non-constant polynomials. The polynomials $F_1, \ldots, F_k$ are called *factors* of F. It is important to realize that since we consider multilinear polynomials (every variable can occur only in the power of ≤ 1), the factors are polynomials *over disjoint sets of variables*. In the following sections,

we assume that the polynomial F does not have *trivial divisors*, i.e., neither x, nor $x+1$ divides F. Clearly, trivial divisors can easily be recognized.

For a polynomial F, a variable x from the set of variables $Var(F)$ of F, and a value $a \in \{0,1\}$, we denote by $F_{x=a}$ the polynomial obtained from F by substituting x with a. $\frac{\partial F}{\partial x}$ denotes a *formal derivative* of F wrt x. Given a variable z, we write $z|F$ if z divides F, i.e., z is present in every monomial of F (note that this is equivalent to the condition $\frac{\partial F}{\partial z} = F_{z=1}$). Given a set of variables Σ and a monomial m, the *projection* of m onto Σ is 1 if m does not contain any variable from Σ, or is equal to the monomial obtained from m by removing all the variables not contained in Σ, otherwise. The *projection* of a polynomial F onto Σ, denoted by $F|_\Sigma$, is the polynomial obtained as the sum of monomials from the set S projected onto Σ, with duplicate monomials removed.

2.1 Factorization Algorithm

Algorithm 1 describes the sequential version of the factorization algorithm. As already mentioned, the factors of a Boolean polynomial have disjoint sets of variables. This property is employed in the algorithm, which tries to compute a variable partition. Once it is computed, the corresponding factors can be easily obtained as projections of the input polynomial onto the sets from the partition.

The algorithm chooses a variable randomly from the variable set of F. Assuming the polynomial F contains at least two variables the algorithm partitions the variable set of F into two sets with respect to the chosen variable:

- the first set Σ_{same} contains the selected variable and corresponds to an irreducible polynomial;
- the second set Σ_{other} corresponds to the second polynomial which can admit further factorization.

The factors of F, F_{same} and F_{other} are obtained as the projections of the input polynomial onto Σ_{same} and Σ_{other}, respectively.

In lines 1-3, we select an arbitrary variable x from the variable set of F and compute the polynomials A and B. A is the derivative of F wrt x and B is the polynomial obtained by setting x to zero in F. In lines 4-10, we loop through the variable set of F excluding x, calculate the polynomials C and D, and check if the product AD is equal to BC. C is the derivative of polynomial A and D is the derivative of polynomial B. To check whether AD is equal to BC we invoke the **IsEqual** procedure in line 6. We describe the **IsEqual** procedure in detail in the next subsection.

2.2 IsEqual Procedure

Algorithm 2 describes the sequential version of the IsEqual procedure.

Algorithm 1 Sequential Factorization Algorithm

Input Boolean polynomial to be factored F
Output F_{same} and F_{other} which are the factors of the input polynomial F

1: Take an arbitrary variable x occurring in F
2: Let $A = \frac{\partial F}{\partial x}, B = F_{x=0}$
3: Let $\Sigma_{same} = x, \Sigma_{other} = \emptyset, F_{same} = 0, F_{other} = 0$
4: **for each** $y \in var(F) \setminus \{x\}$ **do**
5: Let $C = \frac{\partial A}{\partial y}$, $D = \frac{\partial B}{\partial y}$
6: **if** IsEqual(A, D, B, C) **then**
7: $\Sigma_{other} = \Sigma_{other} \cup \{y\}$
8: **else**
9: $\Sigma_{same} = \Sigma_{same} \cup \{y\}$
10: **end if**
11: **end for**
12: If $\Sigma_{other} = \emptyset$ then F is non-factorable
13: Return polynomials F_{same} and F_{other} obtained as projections onto Σ_{same} and Σ_{other} respectively.

- The procedure takes input polynomials A, B, C, D and computes whether $AD = BC$ by employing recursion.
- Lines 1-2,7-16 implement the base cases when $AD = BC$ can be determined trivially.
- In Line 3-5, we check whether a variable z divides the polynomials A, B, C, D such that the condition in Line 4 holds. If this is not the case, then we can eliminate z from A, B, C, D and check if the products of the resulting polynomials are equal.
- In Lines 17-25, we recursively invoke **IsEqual** procedure on polynomials, whose sizes are smaller than the size of the original ones.

2.3 Scope for Parallelism

The crux of Algorithm 1 is the loop in Lines 4-11. We observe that the different iterations of the loop are independent of each other. Hence the loop exhibits thread level parallelism which can be exploited for performance gain. The conditional block inside the loop in Lines 6-10 can be used to exploit the task level parallelism between the multiple threads.
Multiple sections of Algorithm 2 are amenable for parallelization. Checking the divisibility of the polynomials A, B, C, D in Lines 3-6 of **IsEqual** procedure can be performed independently. In Lines 16-23, the recursive calls to **IsEqual** procedure are independent of each other and exhibit thread level parallelism.
In the next section we propose a parallel algorithm using the above observations.

Algorithm 2 Sequential IsEqual Procedure

Input Boolean polynomials A,B,C,D
Output TRUE if AD is equal to BC and FALSE otherwise.

1: If A=0 or D=0 then return (B=0 or C=0)
2: If B=0 or C=0 then return FALSE
3: **for each** z occurring in at least one of A,B,C,D **do**
4: **if** $z|A$ or $z|D$ xor $z|B$ or $z|C$ **then**
5: return FALSE
6: **end if**
7: Replace every $X \in \{A, B, C, D\}$ with $\frac{\partial X}{\partial z}$, provided $z|X$
8: **end for**
9: **if** A=1 and D=1 **then** return (B=1 and C=1)
10: **end if**
11: **if** B=1 and C=1 **then** return FALSE
12: **end if**
13: **if** A=1 and B=1 **then** return (D=C)
14: **end if**
15: **if** D=1 and C=1 **then** return (A=B)
16: **end if**
17: Pick a variable z
18: **if** not(IsEqual($A_{z=0}, D_{z=0}, B_{z=0}, C_{z=0}$)) **then** return FALSE
19: **end if**
20: **if** not(IsEqual($\frac{\partial A}{\partial z}, \frac{\partial D}{\partial z}, \frac{\partial B}{\partial z}, \frac{\partial C}{\partial z}$)) **then** return FALSE
21: **end if**
22: **if** IsEqual($\frac{\partial A}{\partial z}, B_{z=0}, A_{z=0}, \frac{\partial B}{\partial z}$) **then** return TRUE
23: **end if**
24: **if** IsEqual($\frac{\partial A}{\partial z}, C_{z=0}, A_{z=0}, \frac{\partial C}{\partial z}$) **then** return TRUE
25: **else** return FALSE
26: **end if**

3 Proposed Approach

3.1 Parallel Factorization Algorithm

Algorithm 3 describes the parallel version of the factorization algorithm. In Lines 1-3, we select an arbitrary variable x from the variable set of F and compute the polynomials A and B. In Lines 4-11, we perform multiple loop iterations independently in parallel by spawning multiple threads. Each thread will return two sets Σ^{tid}_{same} and Σ^{tid}_{other} specific to the scope of the thread designated by thread identifier *tid*. In Lines 12-13, the variable sets Σ_{same} and Σ_{other} are computed as the union of the thread specific instances, respectively. Note that Lines 12-13 perform barrier synchronization of all the parallel threads.

3.2 Parallel IsEqual Procedure

Algorithm 4 describes the parallel version of the IsEqual procedure. This algorithm takes as input four polynomials A, D, B, C and checks whether the product AD is equal to the product BC. Lines 1-2 and lines 14-21 describe the cases

Algorithm 3 Parallel Decomposition Algorithm

Input Boolean polynomial to be factored F
Output F_{same} and F_{other} which are the factors of the input polynomial F

1: Take an arbitrary variable x occurring in F
2: Let $A = \frac{\partial F}{\partial z}, B = F_{z=0}$
3: Let $\Sigma_{same} = x, \Sigma_{other} = \emptyset, F_{same} = 0, F_{other} = 0$
4: **for each** $y \in var(F) \setminus \{x\}$ **do in parallel**
5: Let $C = \frac{\partial A}{\partial y}$ $D = \frac{\partial B}{\partial y}$
6: **if** IsEqual(A, D, B, C) **then**
7: $\Sigma^{tid}_{other} = \Sigma^{tid}_{other} \cup \{y\}$
8: **else**
9: $\Sigma^{tid}_{same} = \Sigma^{tid}_{same} \cup \{y\}$
10: **end if**
11: **end for Wait for all the parallel threads to finish**
12: $\Sigma_{other} = \bigcup_{tid} \Sigma^{tid}_{other}$
13: $\Sigma_{same} = \bigcup_{tid} \Sigma^{tid}_{same}$
14: If $\Sigma_{other} = \emptyset$ then F is non-factorable; stop
15: Return polynomials F_{same} and F_{other} obtained as projections onto Σ_{same} and Σ_{other}, respectively.

when determining $AD = BC$ is trivial. In lines 3-9, we check whether a variable z divides the input polynomials A, D, B, C such that the condition in Line 5 holds. If this is not the case, we divide them by z to obtain the reduced polynomials. The above operations are performed for each variable independently in parallel by spawning multiple threads. In Line 8 each thread checks whether a variable z^{tid} (tid denotes the thread id) is a divisor of any of A, B, C, D. If z^{tid} divides any of A, B, C, D it computes the corresponding reduced polynomials $A^{tid}, D^{tid}, B^{tid}, C^{tid}$ obtained by dividing any of A, D, B, C by z^{tid}, respectively. In line 10 we wait for all the threads to finish. In Line 13 we take pairwise intersection of the corresponding monomials of thread specific polynomials $A^{tid}, D^{tid}, B^{tid}, C^{tid}$ to form polynomials which are free of trivial divisors. Intersection of two monomials m_1, m_2 is 1 if m_1, m_2 do not contain common variables and otherwise it is the monomial, which consists of the variables present in both m_1 and m_2. In Lines 23-27, we perform four recursive calls to the IsEqual function independently in parallel by spawning multiple threads. In Line 28-37, we wait for all the threads to finish and compare the outputs of each threads to form the final output. Note that lines 10 and 28 perform barrier synchronization of all the parallel threads.

4 Experiments and Results

Experimental evaluation of the sequential and parallel algorithms was made on Logic circuit synthesis benchmarks and synthetic Boolean polynomials.

Algorithm 4 Parallel IsEqual Function

Input Boolean polynomials A,B,C,D
Output TRUE if AD is equal to BC and FALSE otherwise.

If A =0 or D=0 then return (B=0 or C=0)
If B=0 or C=0 then return FALSE
for each z occurring in at least one of A,B,C,D **do in parallel**
    set $flag^{tid}$= True
    if $z|A$ or $z|D$ xor $z|B$ or $z|C$ **then**
        set $flag^{tid}$= FALSE
    end if
    Replace every $X^{tid} \in \{A, B, C, D\}$ with $\frac{\partial X^{tid}}{\partial z}$, provided $z|X^{tid}$
end for
Wait for all threads to finish
if $\bigwedge_{tid} flag^{tid} = FALSE$ **then** return FALSE
end if
$X = \bigcap_{tid} X^{tid}$, for $X \in \{A, B, C, D\}$
if A=1 and D=1 **then** return (B=1 and C=1)
end if
if B=1 and C=1 **then** return FALSE
end if
if A=1 and B=1 **then** return (D=C)
end if
if D=1 and C=1 **then** return (A=B)
end if
Pick a variable z
Do the next 4 lines in parallel
x = not(IsEqual($A_{z=0}, D_{z=0}, B_{z=0}, C_{z=0}$))
y = not(IsEqual($\frac{\partial A}{\partial z}$, $\frac{\partial D}{\partial z}$, $\frac{\partial B}{\partial z}$, $\frac{\partial C}{\partial z}$))
z = IsEqual($\frac{\partial A}{\partial z}, B_{z=0}, A_{z=0}, \frac{\partial B}{\partial z}$)
w = IsEqual($\frac{\partial A}{\partial z}, C_{z=0}, A_{z=0}, \frac{\partial C}{\partial z}$)
Wait for all threads to finish
if not(x) **then** return FALSE
end if
if not(y) **then** return FALSE
end if
if z **then** return TRUE
end if
if w **then** return TRUE
else return FALSE
end if

4.1 Logic Circuit Synthesis Benchmarks

We used ITC'99 [2], Iscas'85 [9], and n-bit ripple carry adder [12] benchmarks. RTL designs of the digital logic circuits were converted from Verilog to the full disjunctive normal form to obtain the corresponding Boolean polynomial. The sequential and parallel algorithms were evaluated on the obtained Boolean polynomials. Table 1 shows the execution time of sequential and parallel algorithms executed on Xeon processor running at 2.8 GHz with 4 threads averaged over 5 runs. One can observe a considerable performance speedup of the parallel algorithm over the sequential one.

Table 1. Results on Xeon processor at 2.8 GHz using 4 threads

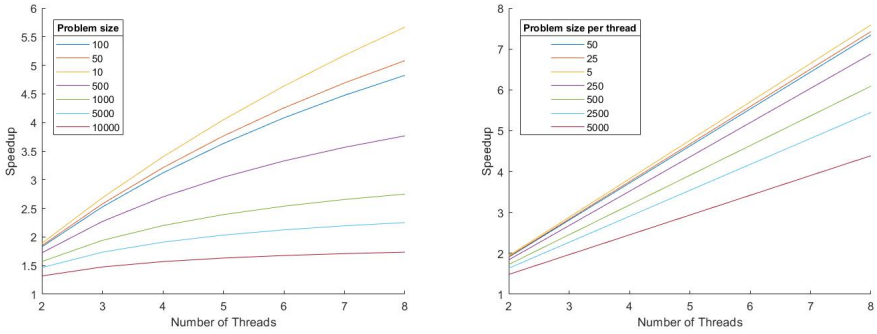
Benchmark	**Sequential**	**Multi-Threaded**	**Speedup**
ITC'99	4324(s)	1441(s)	3.01
Iscas'85	7181(s)	2633(s)	2.73
EPFL Adder	1381(s)	374(s)	3.69

4.2 Synthetic Polynomials

Synthetic polynomials of varying complexities were generated at random and sequential and parallel algorithms were evaluated on them. Table 2 shows execution times for the sequential and parallel algorithms executed on Xeon processor running at 2.8 GHz with 4 threads averaged over 5 runs. We observe that the execution time of both sequential and multithreaded algorithm increases drastically with the increase in the complexity of Boolean polynomials. We also observe that the speedup due to parallelization decreases with the increase in the complexity of Boolean polynomials.

Table 2. Execution time of factoring synthetic polynomials on Xeon processor at 2.8 GHz using 4 threads

Number of Monomials	**Sequential**	**Multi-Threaded**	**Speedup**
10	0.023(s)	0.0074(s)	3.12
50	16.29(s)	5.07(s)	3.21
100	103.5(s)	30.44(s)	3.4
500	483.6(s)	178.1(s)	2.7
1000	1165(s)	520.9(s)	2.2
5000	1430(s)	735.11(s)	1.91
10000	12614(s)	8034(s)	1.57



(a) (b)

Fig. 1. (a) Parallel speedup vs number of threads with fixed problem size
(b) Parallel speedup vs number of threads with fixed problem size per thread

4.3 Scaling Results

Figure 1a shows the speedup of the parallel decomposition algorithm over the sequential one wrt the number of threads. Here, the problem size is fixed to examine the strong scaling behaviour of the parallel decomposition algorithm. We observe that the parallel speedup is decreased as the size (complexity) of the problem increases. As the problem size increases, so does the call to the sequential bottleneck of the algorithm (simplification of Boolean polynomials), which causes the speedup to reduce.
Figure 1b shows the speedup of the parallel decomposition algorithm over the sequential algorithm wrt the number of threads. Here, the problem size per thread is fixed to examine the weak scaling behaviour of the parallel algorithm. The increase in the parallel speedup with the increase in the number of threads is less than the ideal linear speedup. This is due to the sequential bottlenecks in the decomposition algorithm (simplification of Boolean polynomials) and the communication bottleneck among multiple threads. Note that in these tests number of variables ranges from tens to two hundreds.

5 Implementation on Redefine

The REDEFINE architecture [1] comprises Compute Resources (CRs) connected through a Network-on-Chip (NoC) (see Figure 2a). REDEFINE is an application accelerator, which can be customized for a specific application domain through reconfiguration. Reconfiguration in REDEFINE can be performed primarily at two levels, viz. the level of aggregation of CRs to serve as processing cores for coarse grain multi-input, multi-output macro operations, and at the level of Custom Function Units (CFU) presented at the Hardware Abstraction Layer (HAL) as Instruction Extensions. Unlike traditional architectures, Instructions

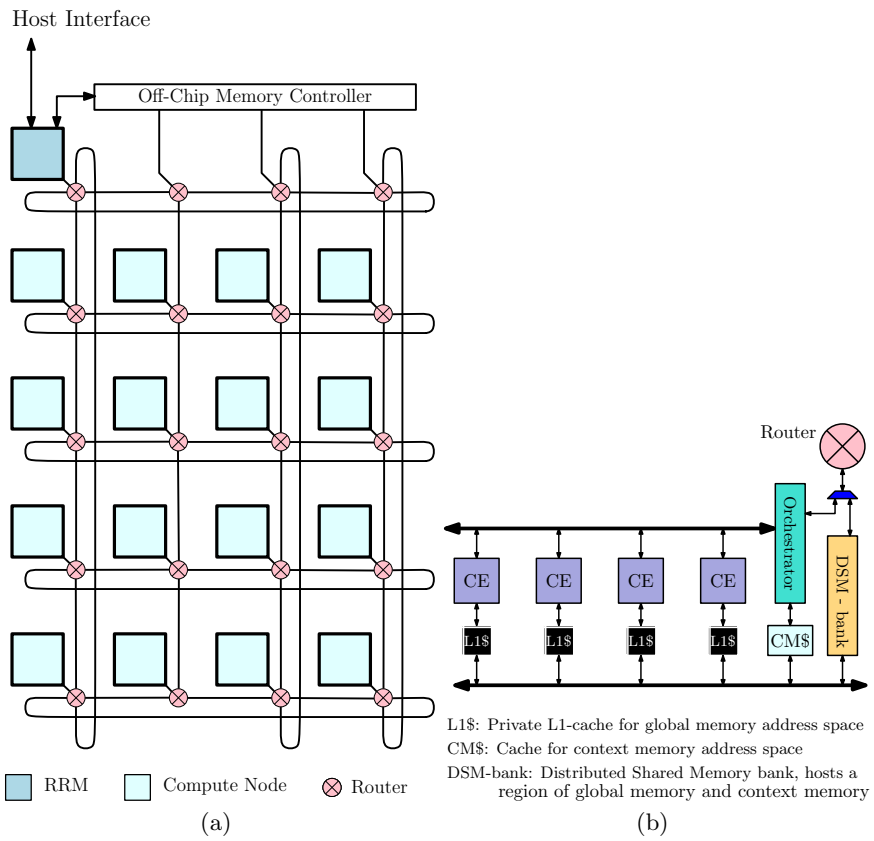


Fig. 2. (a) A 16 node REDEFINE comprising of a 4x4 toriodal mesh of routers and a redefine resource manager(RRM) for interfacing with the host
(b) Composition of a compute Node

Extensions in REDEFINE can be defined post-silicon. Post-silicon definition of Instruction Extensions in REDEFINE is a unique feature of REDEFINE that sets it aside from other commercial multicores by allowing customization of REDEFINE for different application domains.

REDEFINE execution model is inspired by the macro-dataflow model. In this model, an application is described as a hierarchical dataflow graph, as shown in Figure 3, in which the vertices are called hyperOps, and the edges represent explicit data transfer or execution order requirements among hyperOps. A hyperOp is a multiple-input and multiple-output (MIMO) macro operation. A hyperOp is ready for execution as soon as all its operands are available and all its execution order or synchronization dependencies are satisfied. Apart from the arithmetic, control, and memory load and store instructions, the REDEFINE execution model includes primitives for explicit data transfers and synchronization among hyperOps and primitives for adding new nodes (hyperOps)

and edges to the application graph during execution. Thus, the execution model supports dynamic (data-dependent) parallelism. The execution model follows non-preemptive scheduling of hyperOps; therefore cyclic dependencies are forbidden among hyperOps. The runtime unit named Orchestrator schedules ready hyperOps onto CRs. A CR comprises four Compute Elements (CEs). Each CE executes a hyperOp (see Figure 2b). All communications among hyperOps are unidirectional i.e., only producer hyperOp initiates and completes a communication. Thus with sufficient parallelism, all communications can overlap with computations. Compared to other hybrid dataflow/control-flow execution models, REDEFINE execution model simplifies the resource management and the memory model required to support arbitrary parallelism.

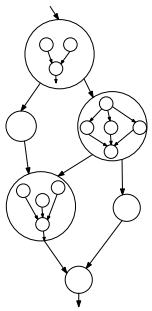


Fig. 3. Macro-dataflow execution model. An application described as a hierarchical dataflow graph, in which vertices represent hyperOps and edges represent explicit data transfer or execution order requirements between the connected hyperOps.

5.1 Decomposition Algorithm Using HyperOps

Algorithm 5 describes in pseudo-code the decomposition algorithm when written using "C with HyperOps". The code snippet, corresponding to Algorithm 5 is presented in the listing below. In the code snippet the terms `__CMAddr, __Sync, __kernel,__WriteCM, CMADDR` are REDEFINE specific annotations. Lines 2-8 and 12-13 of Algorithm 5 are the same as Lines 5-10 and 1-3 of Algorithm 1, respectively. In Lines 15-17 of Algorithm 5, for each variable y in the variable set of F (excluding x) we spawn HyperOps in parallel to calculate whether y belongs to Σ_{same} or Σ_{other}. In Lines 1-10, we define the HyperOp. It takes as input Boolean polynomials A, B and a variable y and adds y to Σ_{same} or Σ_{other}. In Lines 18-20, we wait for the all the HyperOps to finish and output F_{same} and F_{other}.

Listing 1.1 below shows the decomposition algorithm written in C with HyperOps.

The proposed algorithm with HyperOps was evaluated using REDEFINE emulator executed on Intel Xeon processor. Table 3 shows the execution time of the

Algorithm 5 Decomposition Algorithm using HyperOps

Input Boolean polynomial to be factored F
Output F_{same} and F_{other} which are the factors of the input polynomial F
Global variables $\Sigma_{same}, \Sigma_{other}$

Begin HyperOp
Inputs: A, B, variable y
Calculate $C = \frac{\partial A}{\partial y}$, $D = \frac{\partial B}{\partial y}$
if IsEqual(A,D,B,C) **then**
    $\Sigma_{other} = \Sigma_{other} \cup \{y\}$
else
    $\Sigma_{same} = \Sigma_{same} \cup \{y\}$
end if
Call Sync HyperOp
End HyperOp
Take an arbitrary variable x occurring in F
Let $A = \frac{\partial F}{\partial z}, B = F_{z=0}$
Let $\Sigma_{same} = x, \Sigma_{other} = \emptyset, F_{same} = 0, F_{other} = 0$
for each $y \in var(F) \setminus \{x\}$ **do in parallel**
    Spawn HyperOp with inputs A, B, y
end for
Wait for the Sync HyperOp to return
If $\Sigma_{other} = \emptyset$ then F is non-factorable; stop
Return polynomials F_{same} and F_{other} obtained as projections onto Σ_{same} and Σ_{other}, respectively.

Table 3. Parallel factoring of synthetic boolean polynomials using REDEFINE emulation running on Intel Xeon processor at 2.8 GHz

Number of Monomials	**Sequential (cpu cycles)**	**Multi-Threaded (cpu cycles)**	**Redefine (cpu cycles)**
30	17192×10^3	7896×10^3	6837×10^3
50	45612×10^3	14196×10^3	12320×10^3

decomposition algorithm executed on Redefine emulator on synthetic Boolean polynomials. The Redefine implementation has the lowest CPU cycles.

6 Conclusions and Future Work

In this paper, we have reviewed the factorization problem for Boolean polynomials. Factorization provides the basis for decomposition of Boolean functions in DNF and for decomposition of data tables. Hence, it is important to develop efficient factorization procedures. We have considered the approach from [4] for factoring Boolean polynomials and presented a MIMD implementation thereof, which exploits task and data level parallelism to achieve better performance. Evaluation of the sequential and parallel algorithms on logic circuit synthesis benchmarks and synthetic Boolean polynomials showed a considerable speedup

obtained by parallelization. The implementation of the parallel algorithm on a REDEFINE emulator outlined the performance benefits under execution on a massively parallel many core architecture. REDEFINE execution model is based on data flow principles and hence, the need for explicit barrier synchronization is obviated. This results in better performance of MIMD applications (Ex:Boolean factorization) on the REDEFINE architecture. In the future work we are going to benchmark the proposed parallel algorithm on REDEFINE hardware. We also plan to use REDEFINE for an efficient hardware implementation of Boolean functions given as Boolean polynomials and DNFs in order to efficiently implement decomposition algorithms for these representations. Finally, we are going to use these implementations for non-disjoint decomposition of DNFs [11] and data tables [3], which is based on massive computation of disjoint decompositions as a subtask.

```
__hyperOp__ void decompose(__CMAddr selfId , __Op32 a, __Op32 b,__Op32 p_s,__Op32 p_o,
      __Op32 m,__Op32 n, __Op32 i, __Op32 consumerFrId){

   int *A   = a.ptr;
   int *B   = b.ptr;
   int *partition_same = p_s.ptr;
   int *partition_other = p_o.ptr;
   int I = i.i32;
   int n = n.i32;
   int m = m.i32;
   int I=0,J = 0;
   int *C, *D;

   __CMAddr confrId = consumerFrId.cmAddr;
  for (I = 0; I<n; I++){
       *(partition_same+J) = 0;
       *(partition_other+J) = 0;
    }

  for (I = 0; I<m; I++){
    for(J=0;J<n;j++){
       *(C+I*columns+J) = 0;
       *(D+I*columns+J) = 0;
     }
   }
   derivative(A,B,C,i);
   derviative(A,B,D,i);
   if(IsEqual(A,D,B,C)){
    *(partition_other+i) =1;
   }
   else{
        *(partition_same+i) =1;
   }
   __Sync( confrId , -1);
}


__kernel int decompose_start (int *A, int *B,int *partition_same, int *
     partition_other, int N){

    int i = 0, j = 0;
    static int counter = 0;
    __CMAddr decomposeFr;
    __CMAddr syncFr = __CreateInst(&smd_Sync);
    __WriteCM( CMADDR(syncFr, 15), N-1);

  for (i = 1; i<N; i++){
    decomposeFr = __CreateInst(&smd_decompose);
    __WriteCM( CMADDR(decomposeFr, 0), (void*)(A));
    __WriteCM( CMADDR(decomposeFr, 1), (void*)(B));
    __WriteCM( CMADDR(decomposeFr, 2), (void*)(partition_same));
    __WriteCM( CMADDR(decomposeFr, 3), (void*)(partition_other));
    __WriteCM( CMADDR(decomposeFr, 4), M);
    __WriteCM( CMADDR(decomposeFr, 5), N);
    __WriteCM( CMADDR(decomposeFr, 6), i);
    __WriteCM( CMADDR(decomposeFr, 7), CMADDR(syncFr,15));
    }
    return 0;
}
```

Listing 1.1. Snippet of decomposition algorithm using Hyperops

References

1. Redefine - reconfigurable silicon core description. http://morphing.in/redefine, accessed: 2018-12-07
2. Corno, F., Reorda, M., Squillero, G.: Rt-level itc'99 benchmarks and first atpg results. Design Test of Computers, IEEE **17**(3), 44–53 (Jul 2000). https://doi.org/10.1109/54.867894
3. Emelyanov, P.: On two kinds of dataset decomposition. In: Proceedings of the 18^{th} International Conference on Computational Science (ICCS 2018), Part II. Lecture Notes in Computer Science, vol. 10861, pp. 171–183. Springer (2018)
4. Emelyanov, P., Ponomaryov, D.: Algorithmic issues of AND-decomposition of boolean formulas. Programming and Computer Software (2015)
5. Emelyanov, P., Ponomaryov, D.: On tractability of disjoint AND–decomposition of boolean formulas. In: Proceedings of the PSI 2014: 9^{th} Ershov Informatics Conference. Lecture Notes in Computer Science, vol. 8974, pp. 92–101. Springer (2015)
6. Emelyanov, P., Ponomaryov, D.: Cartesian decomposition in data analysis. In: Siberian Symposium on Data Science and Engineering (SSDSE) (2017)
7. Emelyanov, P., Ponomaryov, D.: On a polytime factorization algorithm for multilinear polynomials over f2. In: Gerdt, V.P., Koepf, W., Seiler, W.M., Vorozhtsov, E.V. (eds.) Computer Algebra in Scientific Computing - 20th International Workshop, CASC 2018, Lille, France, September 17-21, 2018, Proceedings. Lecture Notes in Computer Science, vol. 11077, pp. 164–176. Springer (2018). https://doi.org/10.1007/978-3-319-99639-4, https://doi.org/10.1007/978-3-319-99639-4_11
8. von zur Gathen, J., Gerhard, J.: Modern Computer Algebra. Cambridge University Press, New York, NY, USA, Third edn. (2013)
9. Hansen, M.C., Yalcin, H., Hayes, J.P.: Unveiling the iscas-85 benchmarks: A case study in reverse engineering. IEEE Des. Test **16**(3), 72–80 (Jul 1999). https://doi.org/10.1109/54.785838, https://doi.org/10.1109/54.785838
10. Muller, D.E.: Application of Boolean algebra to switching circuit design and to error detection. IRE Transactions on Electronic Computers **EC-3**, 6–12 (1954)
11. Ponomaryov, D.: A polynomial time delta-decomposition algorithm for positive dnfs. In: Proceedings of the 14^{th} International Computer Science Symposium in Russia (CSR). Lecture Notes in Computer Science, vol. 11532. Springer (2019)
12. Schmidt, J., Fišer, P.: A prudent approach to benchmark collection
13. Shpilka, A., Volkovich, I.: On the relation between polynomial identity testing and finding variable disjoint factors. In: Abramsky, S., Gavoille, C., Kirchner, C., Meyer auf der Heide, F., Spirakis, P.G. (eds.) Automata, Languages and Programming (2010)
14. Zhegalkin, I.: Arithmetization of symbolic logics. Sbornik Mathematics **35**(1), 311–377 (1928), in Russian.