Expression Equivalence Checking Using Interval Analysis

Mohammad Ali Ghodrat, *Student Member, IEEE*, Tony Givargis, *Member, IEEE*, and Alex Nicolau, *Member, IEEE*

***Abstract*—Arithmetic expressions are the fundamental building blocks of hardware and software systems. An important problem in computational theory is to decide if two arithmetic expressions are equivalent. However, the general problem of equivalence checking, in digital computers, belongs to the *NP Hard* class of problems. Moreover, existing general techniques for solving this decision problem are applicable to very simple expressions and impractical when applied to more complex expressions found in programs written in high-level languages. In this paper, we propose a method for solving the arithmetic expression equivalence problem using partial evaluation. In particular, our technique is specifically designed to solve the problem of equivalence checking of arithmetic expressions obtained from high-level language descriptions of hardware/software systems. In our method, we use interval analysis to substantially prune the domain space of arithmetic expressions and limit the evaluation effort to a sufficiently limited set of subspaces. Our results show that the proposed method is fast enough to be of use in practice.**

***Index Terms*—Expression equivalence, interval analysis, mutual exclusion.**

I. Introduction

ARITHMETIC expressions are the fundamental building blocks of hardware and software systems. In hardware, arithmetic expressions form the core of datapath designs. In software, arithmetic expressions form the core of basic blocks. A fundamental problem in computational theory is to decide if two expressions are equivalent [2], [3]. In hardware and software systems, expression equivalence is uniquely characterized by operating on finite precision integers. Furthermore, the general problem of equivalence checking, as related to hardware and software systems, belongs to the *NP Hard* class of problems [1].

Efficiently solving the equivalence problem between two arithmetic expressions will have a profound impact in the areas of formal verification [4], complex code generation and technology mapping [5], resource scheduling [6], code transformation [7], synthesis technologies [8], and compiler techniques [9].

In this paper, we propose a method for solving the expression equivalence problem using partial evaluation. In our method, we use interval analysis [10] to substantially prune the domain space of arithmetic expressions and limit the evaluation effort to a limited set of subspaces. We call this method domain space partitioning. Our results show that the proposed method is fast enough to be of use in practice.

Manuscript received July 1, 2005; revised January 6, 2006. This work was supported by the National Science Foundation under Award CCR-0205712.

The authors are with the Department of Computer Science and the Center for Embedded Computer Systems, University of California, Irvine, CA 92697-3425 USA (e-mail: mghodrat@uci.edu).

Digital Object Identifier 10.1109/TVLSI.2006.878471

As another application for domain space partitioning, we can consider *mutual exclusion*. Mutual exclusion is a special instance of the equivalence checking problem. Here, if E_1 and E_2 are two arithmetic expression, we say that E_1 and E_2 are mutually exclusive if the condition $E_1 = E_2$ is false for all values of E_1 and E_2. We say that E_1 and E_2 are not mutually exclusive if, for at least some point in the domain of E_1 or E_2, the expression $E_1 = E_2$ evaluates to true. This is indeed the problem of equivalence checking. If C_1 and C_2 are two conditional expressions (e.g., $x < 0$ and $x > 255$), we say that C_1 and C_2 are mutually exclusive if the condition $C_1 \&\& C_2$ evaluates to false for all points in the domain of C_1 and C_2.

We note that the domain space partitioning method described in this paper can help advance the state of the art in behavioral synthesis tools, reconfigurable computing methodologies, extensible processors, very long instruction word (VLIW), and multiple-processor-on-a-chip compilers, and high-level program validation and verification. As an example, we can take advantage of the fact that the Boolean value of a conditional expression in a program, determining the true/false execution paths, can be statically analyzed using the domain space partitioning method to determine cases when one or the other of the true/false paths are guaranteed to execute. Consequently, in such cases, code is generated to bypass the evaluation of the conditional expression. In instances when the bypass code is faster to evaluate than the conditional expression, a net performance gain is obtained.

The remainder of this paper is organized as follows. In Section II, we show previous related work. In Section III, we formulate the problem of expression equivalence. In Section IV, we give our solution for this problem when we have only one simple arithmetic expression. In Section V, we extend our solution for more complex arithmetic expressions which have Boolean operators also. In Section VI, we present our experimental results. Finally, in Section VII, we give our conclusion.

II. Previous Work

Most of the work on equivalence checking is done in the domain of formal verification. The most commonly used methods to do formal verification of circuits use binary decision diagrams (BDDs) [11] and their derivatives, namely ordered BDD (OBDD), ordered functional decision diagrams (OFDDs), multiterminal BDD (MTBDD), binary moment diagram (BMD), edge-valued BDD (EVBDD), multiplicative BMD (*BMD), and Taylor expansion diagrams (TEDs) [12]. These approaches differ mainly in bit- versus word-level scope and composition rules.

BDD, OBDD, and OFDD are bit-level decision diagrams, while the rest are word-level decision diagrams (bit-level decision diagrams represent Boolean functions $f : \{0,1\}^n \rightarrow \{0,1\}^m$, while word-level decision diagrams represent integer-valued functions $f : \{0,1\}^n \rightarrow Z$). These decision diagram-based approaches also differ in the type of decomposition rule used, specifically, Shannon (BDD, OBDD, and K*BMD), positive-Davio (OFDD and K*BMD), or negative-Davio (K*BMD). Among those decision diagrams that are word-level, a further difference is in the place where the integer weights are inserted, either in leaves (i.e., MTBDD and BMD) or edges (i.e., EVBDD, *BMD, and K*BMD). A detailed survey of BDD and its derivatives can be found in [13]. Finally, TEDs use Taylor series expansion for decomposing algebraic and Boolean expressions [12].

Another approach used in formal verification is using integer linear programming (ILP), where both arithmetic and Boolean operators are linearized to reach to an instance of ILP problem [14].

Due to exponential complexity, bit-level decision diagrams are only applicable to simple Boolean expressions and are not feasible when applied to arithmetic expressions. Word-level decision diagrams can be applied to simple arithmetic expressions (e.g., datapath segments [15]), however, they can only be used to determine the equivalence of arithmetic expressions. Conversely, our method, in addition to checking equivalence, can also partition the domain space into regions and define the arithmetic relations (e.g., less than, greater than, and equal to) present in those regions.

In related work, Wakabayashi *et al.* [16] have used the notion of a condition vector to find mutual exclusion between two Boolean conditions. Two conditional expressions are mutually exclusive if it can be shown that they can never be evaluated to true at the same time. Likewise, Juan *et al.* [17] have proposed condition graphs, a form of syntax pattern matching, to find mutual exclusion between two restricted Boolean conditions. Further, Li *et al.* [18], [19] have used a timed decision table (TDT) to find three possible types of mutual exclusion between a pair of conditional expressions, namely, structural, behavioral, and dataflow. Also, Xie *et al.* [20] used a branch labeling method to find the mutual exclusion properties between two Boolean expressions. Finally, Camposano [6], in his path-based scheduling technique, has proposed a method for determining mutual exclusion based on an exhaustive traversal of all paths in a control flow graph.

The problem of mutual exclusion between two Boolean conditions, as solved previously, is a special case of the problem solved in our study. The main limitation of existing works in this area is the restriction imposed on the grammar and the lack of support for mixed arithmetic and Boolean expressions. The problem solved in our work applies to general arithmetic expressions with arbitrary complexity.

Zhou *et al.* [21] have proposed a formal verification system, called conditional term rewriting on attribute syntax trees (ConTRAST), for verifying the equivalence between two differently synthesized datapaths. In their approach, they maintain attributes (e.g., real bounds) associated with each node of the syntax trees of the two datapaths and combine this with term rewriting to establish equivalence. Their approach differs from ours in that they focus on computation precision of real values as an element of comparison.

Cheung *et al.* [22] have used bit-slicing of BDDs to establish equivalence between two expressions. The main limitation of their approach is scalability, as representing general and arbitrary arithmetic expressions as a BDD is not feasible in terms of space and time requirements.

III. Problem Definition

An arithmetic expression is formed over the language ($+$, $-$, $\times$, integer-constant, integer-variable). A simple condition is in the form of $(expr_1\ ROP\ expr_2)$. Here, $expr_1$ and $expr_2$ are arithmetic expressions and ***ROP*** is a relational operator (e.g., $=, \neq, <, \leq, >, \geq$). Without loss of generality, we can assume all simple conditions to be of the form of $(expr\ ROP\ 0)$. This normalization is achieved by converting $(expr_1\ ROP\ expr_2)$ to $(expr_1 - expr_2\ ROP\ 0)$. Hence, $(expr\ ROP\ 0)$ is called a normalized simple condition. For the remainder of this paper, we refer to a normalized simple condition as a simple condition.

We define an n-dimensional space to be a box-shaped region defined by the Cartesian product $[l_0, u_0] \times [l_1, u_1] \times \cdots \times [l_{n-1}, u_{n-1}]$. In a simple condition, all integer-constants and integer-variables are assumed to be bounded between $\min$ and $\max$ values.[1] Hence, the domain of a simple condition C with n integer-variables $x_0, x_1, \ldots, x_{n-1}$ is an n-dimensional space defined by the Cartesian product $[\min, \max] \times [\min, \max] \times \cdots \times [\min, \max]$.

Given a simple condition C with integer-variables $x_0, x_1, \ldots, x_{n-1}$, the domain space partitioning problem for a simple condition is to partition the domain space of C into a minimal set of n-dimensional spaces $s_1, s_2, \ldots, s_k$ with each space s_i having one of true, false, or unknown truth values. If space s_i has a truth value of true, then C evaluates to true for every point in space s_i. If space s_i has a truth value of false, then C evaluates to false for every point in space s_i. If space s_i has a truth value of unknown, then C may evaluate to true for some points in space s_i and false for others.

For example, consider $C : 2 \times x_0 + x_1 + 4 > 0$. Let us assume $\min = -5$ and $\max = 5$. Therefore, the domain of C is a two-dimensional (2-D) space defined by the Cartesian product $[-5, 5] \times [-5, 5]$. Fig. 1 shows the partitioned domain space and the corresponding truth values for this example using our solution to the domain space partitioning problem.

The problem of equivalence checking can be reduced to that of arithmetic expression evaluation which can be solved by an instance of the domain space partitioning problem. To determine if two expressions E_1 and E_2 are equivalent, we form the new expression $E_1 - E_2 = 0$. Next, we run the domain space partitioning algorithm on $E_1 - E_2 = 0$. If the output of the domain space partitioning contains a space with unknown truth value, we evaluate every point in that space to resolve the true/false values. Each point in turn becomes a new space. E_1 and E_2 are equivalent if and only if the resulting domain space contains no false space. We give our solution to the domain space partitioning problem for a simple condition in Section IV.

[1]Typically, in a computer system, $\min$ and $\max$ values are determined by the width of the processor datapath.

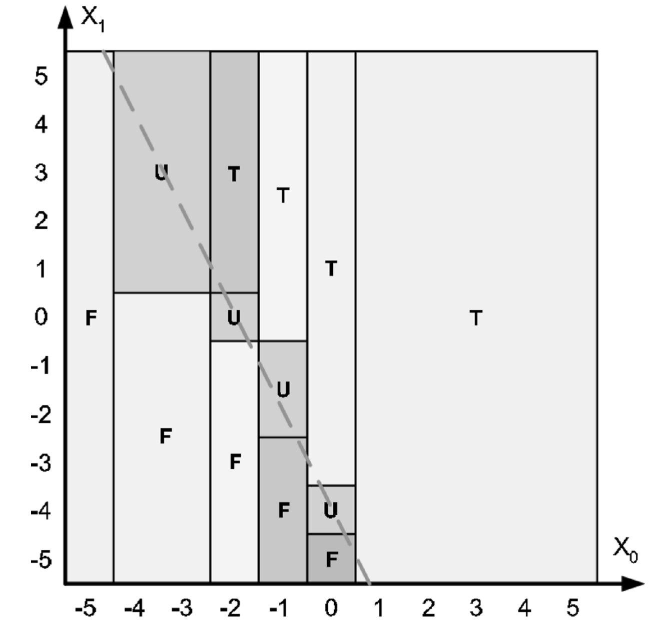

Fig. 1. Partitioned domain of $C : 2x_0 + x_1 + 4 > 0$.

A complex condition is either a simple condition or two complex conditions merged using logical operators (e.g., &&, ||, and !). Specifically, $!C$ computes the negation of the complex condition C; $(C_1\&\&C_2)$ computes logical-and of complex conditions C_1 and C_2; and $(C_1||C_2)$ computes logical-or of complex conditions C_1 and C_2.

The domain of a complex condition C with n integer-variables $x_0, x_1, \ldots, x_{n-1}$ is an n-dimensional space defined by the Cartesian product $[\min, \max] \times [\min, \max] \times \cdots \times [\min, \max]$.

Similar to the domain space partitioning problem for simple conditions, given a complex condition C with integer-variables $x_0, x_1, \ldots, x_{n-1}$, the domain space partitioning problem for complex conditions is to partition the domain space of C into a minimal set of n-dimensional spaces $s_1, s_2, \ldots, s_k$ with each space s_i having one of true, false, or unknown truth value. If space s_i has a truth value of true, then C evaluates to true for every point in space s_i. If space s_i has a truth value of false, then C evaluates to false for every point in space s_i. If space s_i has a truth value of unknown, then C may evaluate to true for some points in space s_i and false for others.

The general problem of equivalence checking between two expressions $expr_1$ and $expr_2$ with bounded variables[2] can be expressed in terms of the domain space partitioning problem for complex conditions. As an example, consider checking equivalence between $expr_1 = 2 \times x_0$ and $expr_2 = -x_1 - 4$. Further, let us assume x_0 and x_1 are 3-b two's complement integers. We can construct the following complex condition:

$$\begin{aligned}((2 \times x_0) - (-x_1 - 4) &= 0)\\ \&\&(x_0 + 4 &\geq 0)\\ \&\&(x_0 - 3 &\leq 0)\\ \&\&(x_1 + 4 &\geq 0)\\ \&\&(x_1 - 3 &\leq 0).\end{aligned}$$

[2]The ability to bound integer variables is necessary when considering hardware/software implementations.

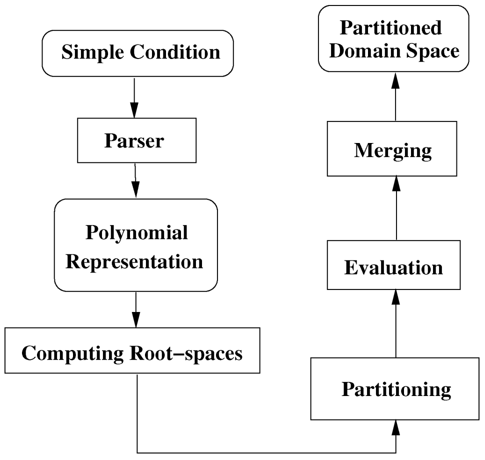

Fig. 2. Space partitioning strategy.

Here, $(2 \times x_0) - (-x_1 - 4) = 0$ evaluates to true, for values of x_0 and x_1 where $expr_1$ and $expr_2$ are equivalent. The remaining expressions (i.e., $x_0 + 4 \geq 0, x_0 - 3 \leq 0, x_1 + 4 \geq 0$, and $x_1 - 3 \leq 0$) evaluate to true when x_0 and x_1 are within the 3-b two's complement bounds. To establish equivalence, we solve the domain space partitioning problem and check that the entire region is marked as true. We give our solution to the domain space partitioning problem for a complex condition in Section V.

IV. Domain Space Partitioning for Simple Condition

A. Overview

Our overall domain space partitioning strategy is depicted in Fig. 2. On input, the arithmetic expression of the simple condition is parsed to obtain an equivalent polynomial representation. Then, we operate on the polynomial and obtain a set of minimally sized spaces (root-spaces) that contain the roots of that polynomial, as outlined in Section IV-C. Given the root-spaces for the polynomial, the entire domain of the polynomial can be partitioned into a number of disjoint spaces. This is accomplished by extending the boundaries of each root-space to the limits of the entire domain to establish the borders between the disjoint spaces (see Section IV-D). After partitioning the domain space, each disjoint space not overlapping with any of the root-spaces, can be evaluated separately. This is done by picking an arbitrary point in it and evaluating the simple condition (see Section IV-E). Finally, when two n-dimensional spaces have the same truth value and share $n - 1$ common borders, then these two spaces can be merged (see Section IV-F). This will result in the evaluated and partitioned domain space which is the output of the domain space partitioning problem.

We first show the main steps of our methodology on a simple example shown in Fig. 3. Details of each step are given in subsequent sections. In Fig. 3, an instance of the domain space partitioning problem for the simple condition $4x_0 + 5x_1 - 20 > 0$ is solved. Fig. 3(a) shows the domain space $\langle[0, 10], [0, 10]\rangle$ of the expression. Fig. 3(b) shows the resulting root-space $(\langle[0, 5], [0, 4]\rangle)$ after running the root-space computation step. In other words, there is at least one integer

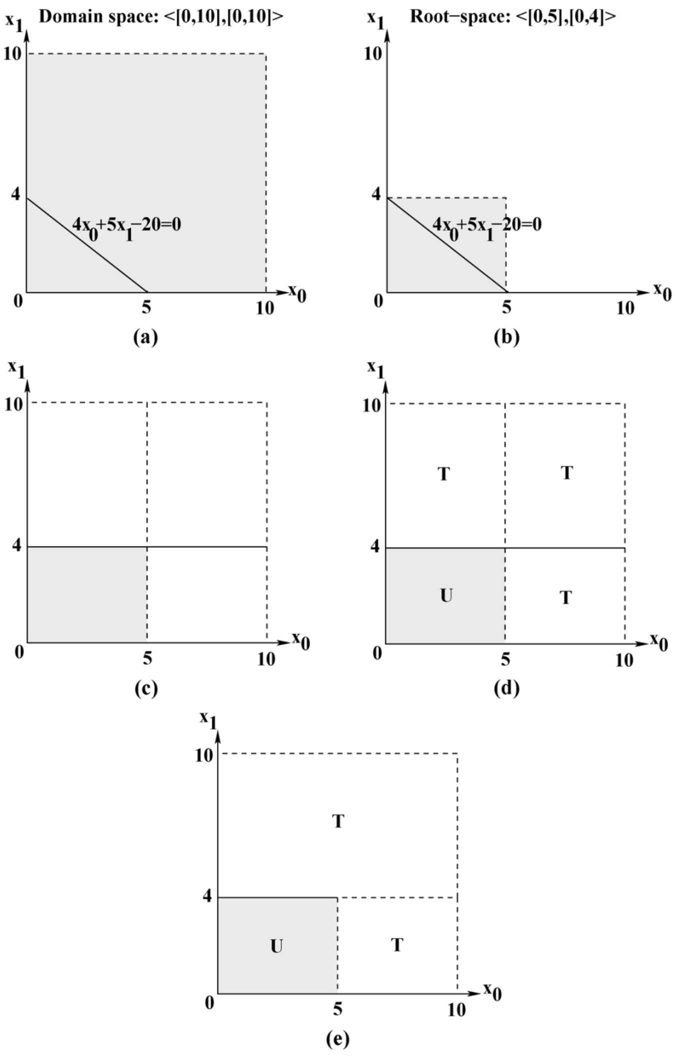

Fig. 3. Simple condition $4x_0 + 5x_1 - 20 > 0$. (a) Initial domain space. (b) Root-space computation. (c) Partitioning. (d) Evaluation. (e) Merging.

root for the expression $4x_0 + 5x_1 - 20$ in $\langle[0,5],[0,4]\rangle$ (e.g., $\langle x_0 = 5, x_1 = 0\rangle$). Fig. 3(c) shows the output of the partitioning step, i.e., the four spaces in the partitioned domain space. Fig. 3(d) shows the result of the evaluation step for each space in the partitioned domain space. For evaluation, one point in each space has been selected, condition $4x_0 + 5x_1 - 20 > 0$ is evaluated, and the resulting truth value is assigned to that space. Finally, Fig. 3(e) shows the result of the merging step on the partitioned and evaluated domain space. Here, the two upper spaces which are neighbors and both are true are merged and one larger true space is resulted.

B. Parsing

Any arbitrary arithmetic expression can be rewritten as an n-variable polynomial with degree D using the general form shown in

$$\sum_{i_0,i_1,\ldots,i_{n-1}=0}^{D} c_{i_0,i_1,\ldots,i_{n-1}} \times x_0^{i_0} \times x_1^{i_1} \times \cdots \times x_{n-1}^{i_{n-1}}. \quad (1)$$

For example, the expression $2 \times x_0 + x_1 + 4$ of Fig. 1 can be rewritten as $2 \times {x_0}^1{x_1}^0 + {x_0}^0{x_1}^1 + 4 \times {x_0}^0{x_1}^0$ (zero coefficient terms not shown) with $n = 2$ and $D = 1$. We describe the remaining domain space partitioning steps in the following subsections.

C. Computing Root-Spaces

During this phase, we operate on an n-variable polynomial P and obtain a set of minimally sized spaces (root-spaces) that contain the roots of P, as outlined in Algorithm 1. We achieve this by finding the roots of P using interval analysis [10]. Let us first give an overview of the interval analysis technique.

A real interval of the form $[a, b]$ represents all possible values in the range a to b. The operations (i.e., $+$, $-$, $\times$, and $/$) can be defined on two real intervals $[a, b]$ and $[c, d]$ as

$$[a,b] + [c,d] = [a+c, b+d] \quad (2)$$
$$[a,b] - [c,d] = [a-d, b-c] \quad (3)$$
$$[a,b] \times [c,d] = [\min(a \times c, a \times d, b \times c, b \times d), \max(a \times c, a \times d, b \times c, b \times d)] \quad (4)$$
$$\frac{[a,b]}{[c,d]} = \begin{cases} [a,b] \times \left[\frac{1}{d}, \frac{1}{c}\right], & 0 \notin [c,d] \\ [-\infty, \infty], & 0 \in [c,d]. \end{cases} \quad (5)$$

Next, we describe our strategy (Algorithm 1) for computing the root-spaces. Algorithm 1 operates as follows:

Algorithm 1 Compute Root-spaces

```
1: Input: a $n$-variable polynomial $P$
2: Output: a set $R$ of minimally sized root-spaces
3: $R \leftarrow \emptyset$
4: $S \leftarrow \langle[\min,\max],\ldots,[\min,\max]\rangle$ {$|space| = n$}
5: $Q$.push($S$)
6: while $Q$.not_empty() do
7:     $S \leftarrow Q$.pop() {S is in the form of $\langle v_0, v_1, \ldots, v_{n-1}\rangle$}
8:     $changed \leftarrow 0$
9:     for all $x_i \in P$ do
10:        $P' \leftarrow$ convert $P$ to a polynomial with $x_i$ as the only variable and $x_j = v_j$ ($v_j \in S$)
11:        $roots \leftarrow P'$.solve()
12:        for all $r \in roots$ do
13:            if $r \neq v_i$ ($v_i \in S$) then
14:                $changed \leftarrow 1$
15:                $r \leftarrow r \cap v_i$ {Intersect new root with previous one}
16:                $Q$.push($\langle v_0, v_1, \ldots, r, \ldots, v_{n-1}\rangle$) {replace $v_i$ with $r$}
17:            end if
18:        end for
19:    end for
20:    if $changed = 0$ then
21:        $R \leftarrow R \cup \{S\}$
22:    end if
23: end while
```

```
24: for all $R_i \in R$ do
25:     $R_i \leftarrow$ convert $R_i$ to smallest bounding integer space
26: end for
```

1) **Initialization Phase (lines 3–5):** We start by creating a single root-space S that covers the entire domain of P. Specifically, S is an n-dimensional space with each dimension initialized to the interval $[\min, \max]$. Clearly, the roots of P (if any) are within S, however, S may not be minimally sized. To minimize S, we push S onto a queue Q to be processed by the iterative phase of the algorithm. In our running example $2 \times {x_0}^1 {x_1}^0 + {x_0}^0 {x_1}^1 + 4 \times {x_0}^0 {x_1}^0$ ($\min = -5$ and $\max = 5$), S is initialized to $\langle[-5,5],[-5,5]\rangle$.

2) **Iterative Phase (lines 6–23):** We pop a space S from the queue Q and split S into smaller spaces $S_0, S_2, \ldots, S_{k-1}$. If $S_0 \cup S_1 \cdots \cup S_{k-1} = S$, then S cannot be minimized, thus we add S to the output list of root-spaces R. If $S_0 \cup S_1 \cdots \cup S_{k-1} \subset S$, then we push $S_0, S_1, \ldots, S_{k-1}$ onto the queue Q and discard S. This process iterates until the queue Q is empty. This phase proceeds as follows:

a) As long as the queue Q is not empty, we pop a space S from the queue Q and clear a flag called *changed* (lines 7–8).

b) For each variable x_i in P, we compute a single variable polynomial P' by setting all variables x_j $(j \neq i)$ to the corresponding intervals $v_j \in S$. Next, we solve P' using any root finding algorithm (e.g., the Newton–Raphson Method [23]), implemented using interval analysis to obtain a set of one or more disjoint root-spaces (i.e., roots, line 9–11). In our running example, P' is computed twice during the run of the loop starting on line 9. In the first round, with x_0 as the variable, P' is $2 \times {x_0}^1 + [-5,5] + [4,4]$. Since P' is a polynomial of degree 1, we compute the root as $[-4.5, 0.5]$.

c) We compare each of $r_0, r_1, \ldots$ to the present value of x_i in space S, namely, v_i. If any root $r_0, r_1, \ldots$ is not equal to v_i, we create a new space and push it onto the queue Q for further processing. Moreover, we set the flag *changed* to signal that S should not be recorded in the output set R (lines 12–18). In our running example, root $r_0 = [-4.5, 0.5]$ is not equal to $v_0 = [-5,5]$, thus we create a new space $S_0 = \langle[-4.5,-0.5],[-5,5]\rangle$.

d) Once steps b) and c) are completed, if the flag *changed* is not set, S cannot be further minimized, thus we push it on the output set R (lines 20–22).

As an optimization, we use a method to help reach to shorter intervals for each root space computed in step 2 of our algorithm. Shorter interval helps in faster convergence for the algorithm. Specifically, if a root space $[lb, ub]$ contains 0 (i.e., $lb < 0 < ub$) we divide it into three intervals $[lb, -1]$, [0, 0] and $[1, ub]$. For example, in the running example, after computing the root for x_1, we reach to the interval $[-5,5]$. Then, we divide this interval into three disjoint intervals $[-5,-1]$, [0, 0], and [1, 5] to be pushed on the queue Q for processing during the following iteration of the algorithm. If $[lb, ub]$ does not contain 0, then we may divide it into three intervals $[lb, (ub+lb)/2 - 1]$, $[(ub+lb)/2, (ub+lb)/2]$, and $[(ub+lb)/2+1, ub]$. But this approach needs a terminating condition based on the size of the space.

TABLE I
ROOT-SPACES OF $2x_0 + x_1 + 4$

Final Real Results	Final Integer Results
$[0,0][-4,-4]$	$[0,0][-4,-4]$
$[-1.5,-1][-2,-1]$	$[-1,-1][-2,-1]$
$[-4.5,-2.5][1,5]$	$[-4,-3][1,5]$
$[-2,-2][0,0]$	$[-2,-2][0,0]$

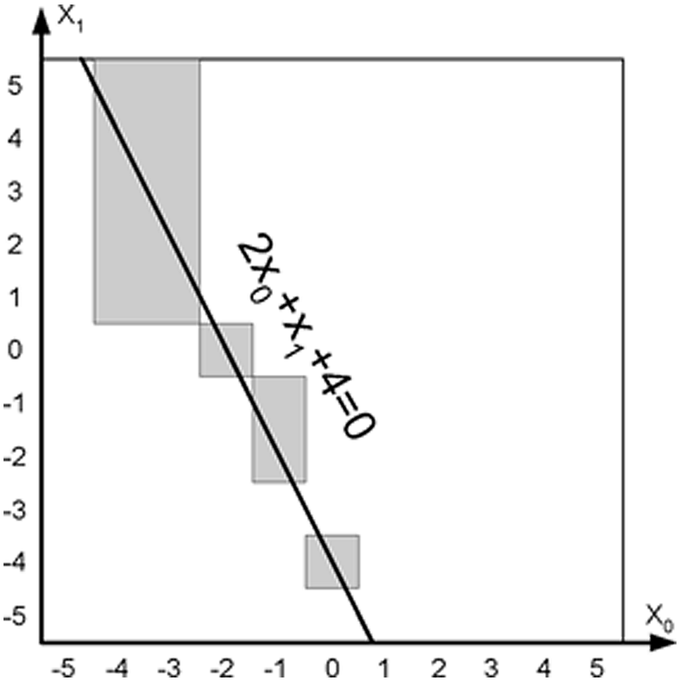

Fig. 4. Root-spaces of $2x_0 + x_1 + 4$.

3) **Quantization Phase (lines 24–26):** Finally, we convert each root-space in the output set R to the smallest bounding integer space. Table I gives the final output set R for our running example. This result is shown graphically in Fig. 4. All the shaded areas are the root-spaces, and, as shown in Fig. 4, the equation $2x_0 + x_1 + 4 = 0$ passes through all of them.

D. Partitioning

Given the root-spaces for an expression E_x (corresponding to a normalized simple condition E_x *ROP* 0), the entire domain of E_x can be partitioned into a number of disjoint spaces. This is accomplished by extending the boundaries of each root-space to the limits ($\min$ and $\max$) of the entire domain to establish the borders between the disjoint spaces. For our running example, the boundary points $\{0,-1,-4,-3,-2\}$ for x_0 and $\{-4,-2,-1,1,5,0\}$ for x_1 (see Table I) partition the entire domain space as shown in Fig. 5. In Fig. 5, the root-spaces are shown in shaded color.

For each disjoint space s_i and s_i not overlapping with any of the root-spaces, it must be the case that evaluating the corresponding expression for any point in s_i will yield only positive results or only negative results, but not both (otherwise, s_i would contain a root and thus will have an overlap with one of

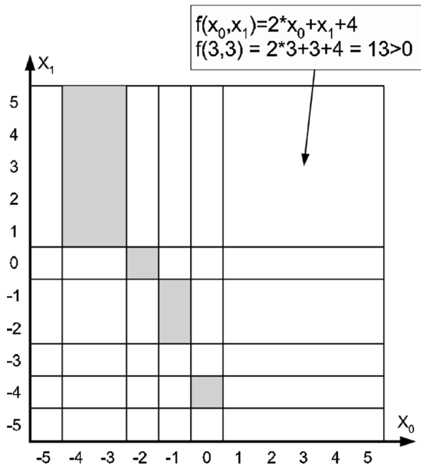

Fig. 5. Partitioned spaces for $2x_0 + x_1 + 4$.

the root-spaces). In Fig. 5, all spaces that are not shaded have this property. For example, the point (3, 3) in space $\langle[1,5][1,5]\rangle$ will make the expression $2x_0 + x_1 + 4$ positive. Furthermore, this is true for all of the points in space $\langle[1,5][1,5]\rangle$.

E. Evaluation

After partitioning the domain space, each disjoint space s_i, and s_i not overlapping with any of the root-spaces, can be evaluated separately. This is done by picking an arbitrary point in s_i and evaluating the simple condition C. This will yield either a true or a false result. Accordingly, space s_i can be marked as true or false. For a disjoint space s_j, and s_j overlapping with one of the root-spaces, such evaluation can not be performed, therefore, s_j must be marked as unknown. For example, evaluating $2x_0 + x_1 + 4 > 0$ with the arbitrary point (3, 3) in space $\langle[1,5],[1,5]\rangle$ yields a true value, thus, the entire space $\langle[1,5],[1,5]\rangle$ is marked as true (see Fig. 6). Conversely, evaluating $2x_0 + x_1 + 4 > 0$ with the arbitrary point $(-3,-2)$ in space $\langle[-4,-3],[-2,-1]\rangle$ yields a false value, thus, the entire space $\langle[-4,-3],[-2,-1]\rangle$ is marked as false (see Fig. 6).

F. Merging

When two n-dimensional spaces have the same truth value and share $n-1$ common borders, then these two spaces can be merged. For example, in Fig. 6, space $\langle[-1,-1],[0,0]\rangle$ and $\langle[-1,-1],[1,5]\rangle$ share the common border $[-1,-1]$ and thus can be merged into a single space $\langle[-1,-1],[0,5]\rangle$.

In our proposed technique (i.e., Fig. 2), the overall running time is bounded by the running time of the merging step. Given k disjoint n-dimensional spaces, a brute-force approach can be used to solve the merging problem. To do so, we take each pair of spaces (i.e., $O(k^2)$) and look for $n-1$ common borders (i.e., $O(n)$) for a total cost of $O(k^2 \times n)$. Here, in the worst case, one pair of spaces may be merged, reducing the total number of spaces to $k-1$. Then, the process repeats k times, until a single space remains. Thus, the total running time takes $O(k^3 \times n)$. The dimensionality n is the number of variables in the simple condition and is usually small (e.g., less than 8) for manually written programs. Hence, the effective running time of the brute-force merging algorithm is $O(k^3)$.

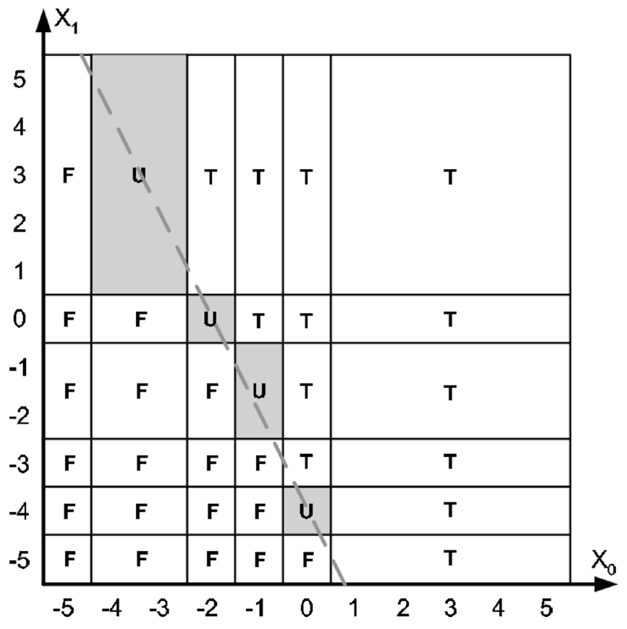

Fig. 6. Evaluated subspaces for $2x_0 + x_1 + 4 > 0$.

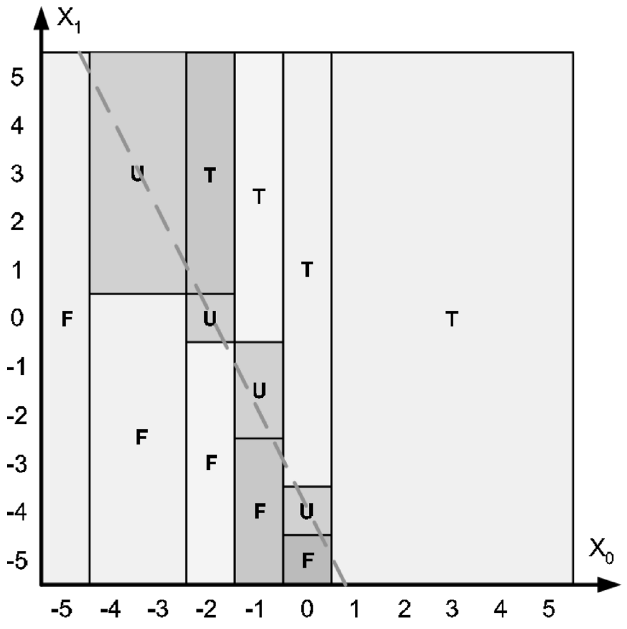

Fig. 7. Merged spaces for $2x_0 + x_1 + 4 > 0$.

Alternatively, we can use a divide-and-conquer heuristic to do this in $O(k^2)$. The idea is to subdivide the k disjoint sets into two equal clusters and recursively merge each cluster. In turn, each of these two clusters will be broken further until the size of the cluster is less than or equal to two. There are exactly $O(k/2) = O(k)$ such leaf clusters, and merging a leaf cluster takes $O(1)$, for a total of $O(k)$. The above procedure would, in the worst case, merge a single pair during each iteration, reducing the total number of clusters to $k-1$. Repeating, as long as some clusters have merged, would take $O(k)$ iterations. Thus, the final run time is bounded by $O(k^2)$.

Fig. 7 shows the result of merge operation on Fig. 6.

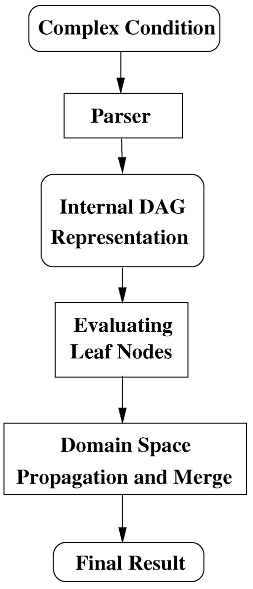
Complex Condition
Parser
Internal DAG Representation
Evaluating Leaf Nodes
Domain Space Propagation and Merge
Final Result

Fig. 8. Solution strategy for domain space partitioning for complex condition.

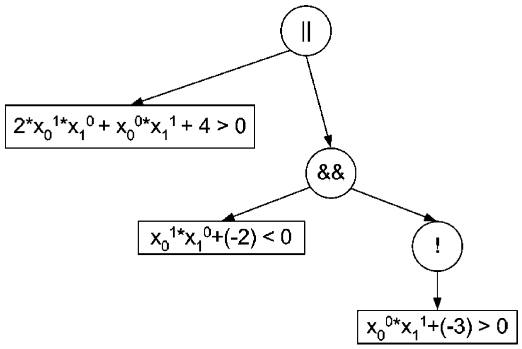

Fig. 9. DAG representation.

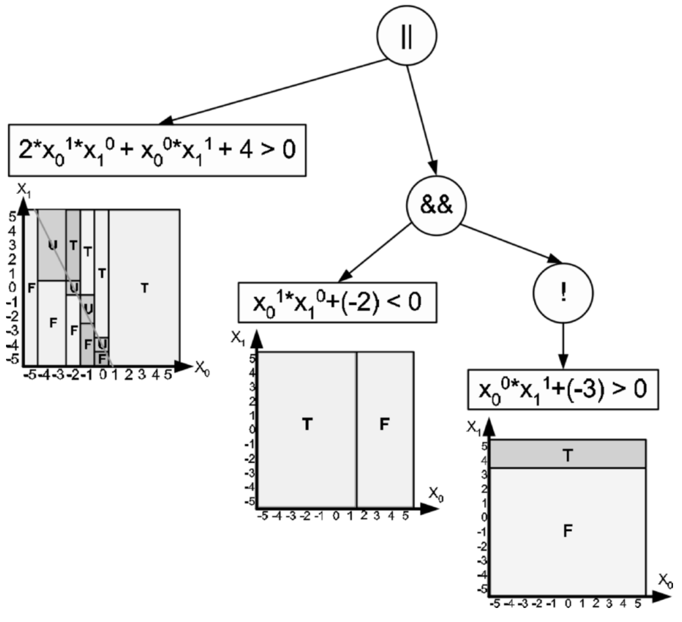

Fig. 10. Partitioned domain spaces for leaf nodes.

&&	T	F	U
T	T	F	U
F	F	F	F
U	U	F	U

!	
T	F
F	T
U	U

||	T	F	U
T	T	T	T
F	T	F	U
U	T	U	U

Fig. 11. Merge rules for operators &&, ||, and !.

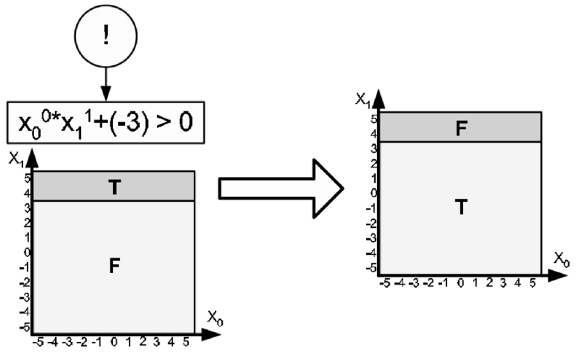

Fig. 12. Applying logical not operator (!) to leaf nodes.

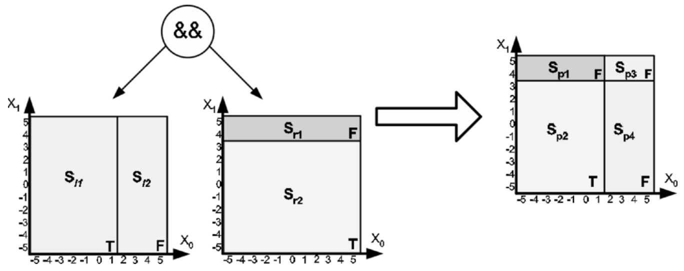

Fig. 13. Applying logical and operator (&&) to leaf nodes.

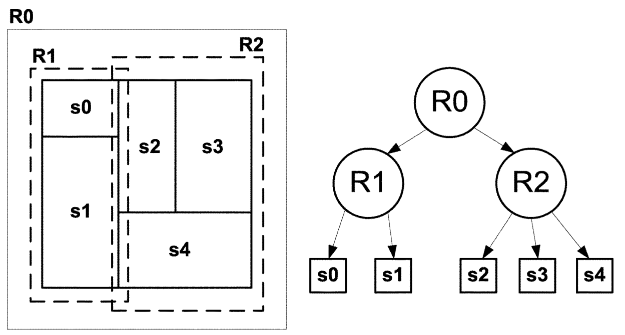

Fig. 14. Partitioned domain space representation using R-tree.

V. Domain Space Partitioning for Complex Condition

Our overall strategy for solving the domain space partitioning problem for complex conditions is depicted in Fig. 8. The steps involved include parsing, evaluating leaf nodes, and domain space propagation/merging. These steps will be described in detail in the following sections.

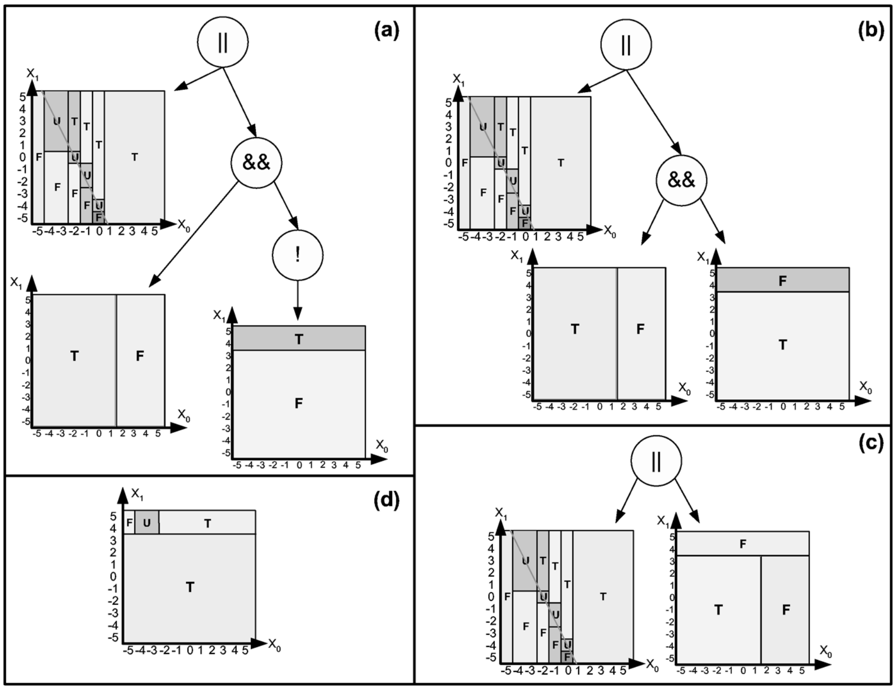

Fig. 15. Merging and propagation of spaces for Fig. 10. (a) Initial state./ (b) After applying ! operator. (c) After merging using &&. (d) After merging using ||.

TABLE II
OPERATION COMPLEXITY FOR MEDIABENCH APPLICATIONS

Benchmark	#Exp	Avg. #Var	Avg. #Arith	Avg. #Logic	Time (ms)
ADPCM	22	1.23	0.68	0.45	0.454545
EPIC	86	1.25	0.55	0.38	1.046510
G721	47	1.34	1.97	1.59	4.255320
GHSTSCR	14	3	1.71	2.07	3.571430
GSM	29	1.24	1.65	1.41	1.034480
JPEG	32	1.5	2.31	1.59	1.875000
MPG-DEC	11	1.54	2	1.36	0.909091
MPG-ENC	12	2.75	2.58	2.08	4.166670
PEGWIT	15	1.33	2.86	2.6	1.333330
PGP	14	1.92	2.42	3.35	5.000000
RASTA	15	2.11	2.33	2.33	3.333330

TABLE III
RESULTS FOR MEDIABENCH APPLICATIONS

Benchmark	True (%)	False (%)	Unknown (%)
ADPCM	23.8636	73.8636	2.27273
EPIC	54.3605	38.6628	6.97674
G721	25.0002	71.8082	3.19156
GHSTSCR	28.5714	53.1250	18.3036
GSM	13.7933	81.0343	5.17241
JPEG	15.6250	76.5625	7.81250
MPG-DEC	27.2727	63.6364	9.09090
MPG-ENC	23.6197	54.5747	21.8055
PEGWIT	9.72228	86.6666	3.61111
PGP	21.4286	78.5714	0.00000
RASTA	15.2778	81.9444	2.77778

A. Parsing

To capture a complex condition, we use a DAG representation with internal nodes of types (&&, ||, !) and leaf nodes of type simple conditions. As mentioned in Section III, the simple condition is captured as a multivariable polynomial *ROP 0*. As a running example, consider the complex condition $(2 \times x_0 + x_1 + 4 > 0)$ || ($(x_0 - 2 < 0)$ && !$(x_1 - 3 > 0)$) and its DAG representation shown in Fig. 9.

B. Evaluating Leaf Nodes

Each leaf node in the DAG representation is a simple condition and is evaluated as outlined in Section IV. Specifically, each leaf node in the DAG representation corresponds to one instance of the domain space partitioning problem for simple conditions. Fig. 10 shows the partitioned domain spaces for the leaf nodes of our running example.

C. Domain Space Propagation and Merging

After computing the partitioned domain spaces for leaf nodes, merging of these domain spaces is performed according to the rules listed in Fig. 11. These rules define how two sets of domain spaces are combined under the logical operators (i.e., &&, ||, and !).

For the logical *not* operator (!), the truth value of a space marked as *true* or *false* is inverted. A space marked as *unknown* is unchanged. Fig. 12 shows the DAG representation after applying logical not operator (!) to the $(x_1 - 3 > 0)$ leaf node.

For the logical *and* operator (&&), the merging is performed on those spaces that have an overlap region. Let us assume that L and R are two partitioned domain spaces. Let us further assume that $s_l \in L$ and $s_r \in R$ are two overlapping spaces in those domains. If space s_p is the overlapping space between s_l and s_r, then s_p will be added to the result of the logical *and*. The truth value of

TABLE IV
PARTIAL LIST OF SYNTHETIC SIMPLE CONDITION EXAMPLES

Simple Condition	#Spaces	Time (sec)
$(x0 + x1 + x2 == 100)$	441	0.05
$(x0 * x1 + x2 < 100)$	326	0.02
$(x0 * x0 + x1 * x1 * x2 < 100)$	298	0.02
$(x0 * x0 * x1 * x2 + x0 < 100)$	248	0.01
$(x0 * x0 * x1 * x2 == 100)$	114	0
$(x0 * x0 * x1 * x1 + x2 == 100)$	76	0.01
$(x0 + x1 + x2 + x3 == 100)$	7158	1.58
$((x0 * x0) + (x1 * x2) + x3 == 100)$	5341	1.34
$(x0 * x0 + x1 * x1 + x2 + x3 < 100)$	4597	2.35
$(x0 * x1 * x2 + x3 < 100)$	3209	0.74
$(x0 * x1 * x2 * x3 < 100)$	2036	0.21
$(x0 * x1 + x2 * x3 == 100)$	1296	0.16
$(x0 * x0 * x1 * x1 + x2 * x3 == 100)$	678	0.08
$(x0 * x0 * x1 * x1 * x2 * x3 == 100)$	345	0.05
$(x0 + x1 + x2 + x3 + x4 == 100)$	171975	95.58
$(x0 * x1 * x2 + x3 + x4 < 100)$	97802	47.99
$((x0 * x0 * x1 * x2) + x3 + x4 == 100)$	84499	42.14
$((x0 * x0) + (x1 * x1) + x2 + x3 + x4 == 100)$	63296	144.97
$((x0 * x0) + (x1 * x2 * x3 * x4) < 100)$	38456	10.72
$((x0 * x0) + (x1 * x2) + (x3 * x4) < 100)$	24057	10.02
$(x0 * x0 * x1) + (x2 * x3 * x4) < 100)$	10616	2.63
$(x0 * x0 * x1 * x1 * x2 * x3 * x4 < 100)$	6336	1.1
$((x0 * x0 * x1 * x1) + x2 + x3 + x4 < 100)$	3272	1.29

s_p is computed using the merge rules given in Fig. 11. This procedure is shown in Algorithm 2. Fig. 13 shows an example of the logical *and* merging of two partitioned domain spaces. In Fig. 13, two spaces s_{l1} and s_{r1} are overlapping and their overlap is space s_{p1}, with its truth value set to *false*. In the same way, the overlap of two spaces s_{l1} and s_{r2} is space s_{p2}, with its truth value set to *true*.

Algorithm 2 Logical-AND Space Merging-Exhaustive Method

```
1: **Input:** Partitioned domain spaces $S_l$ and $S_r$
2: **Output:** Merged domain space $S_p$
3: **for all** $spaces\ l \in S_l$ **do**
4:     **for all** $spaces\ r \in S_r$ **do**
5:         $p \leftarrow l \cap r$ {Compute the intersection of the two
           subspaces}
6:         **if** $(p \neq \phi)$ **then**
7:             $p.truth \leftarrow f_{mergerules}(l.truth, r.truth)$ {See
           **Fig. 11**}
8:             $S_p.push(p)$
9:         **end if**
10:    **end for**
11: **end for**
12: $S_p.merge()$
13: return $S_p$
```

Algorithm 2, with two nested *for* loops, has $O(N^2)$ running time. To improve on this algorithm, instead of comparing all the pairs of spaces in each domain space to see if they are overlapped or not, we use the R-tree data structure [24] to make the search job faster. An R-tree as defined in [24] is a height-balanced tree suitable for handling spatial data in multidimensional spaces. Fig. 14 shows a partitioned domain space and the way it is represented using the R-tree structure.

Algorithm 3 uses the R-tree data structure to make Algorithm 2 faster. Specifically, Algorithm 3 uses an R-tree representation of the domain spaces to efficiently find all overlapping regions. The running time of Algorithm 3 is $O(N \times log(N))$.

Finally, the logical *or* operator can be performed in a way similar to the logical *and* operator outlined above.

Algorithm 3 Logical-AND Space Merging-Using R-tree

```
1: **Input:** Partitioned domain spaces $S_l$ and $S_r$
2: **Output:** Merged domain space $S_p$
3: rT = make an R-tree using $S_r$
4: **for all** $spaces\ l \in S_l$ **do**
5:     overlappedRegion = rT.overlap(l)
6:     **for all** $spaces\ o \in overlappedRegion$ **do**
```

TABLE V
PARTIAL LIST OF SYNTHETIC COMPLEX CONDITION EXAMPLES

Complex Condition	#Spaces	Time (sec)
$(x0*x0*x1*x1*x2 == 100) \&\& (x0*x0*x1*x1*x2 == 200)$	1200	0.02
$(x0*x1*x2+x0 < 100) \&\& (x0*x1*x2+x0 < 200)$	2000	0.05
$(x0*x1+x0*x2 < 100) \| \| (x0*x1+x0*x2 < 200)$	4032	0.08
$(x0*x0*x1*x2+x0 == 100) \&\& (x0*x0*x1*x2+x0 == 200)$	4704	0.06
$(x0*x1+x2 < 100) \| \| (x0*x1+x2 < 200)$	5120	0.11
$(x0*x0+x1*x1+x2 == 100) \&\& (x0*x0+x1*x1+x2 == 200)$	5800	0.12
$(x0+x1+x2 < 100) \| \| (x0+x1+x2 < 200)$	6750	0.13
$(x0*x0*x1*x1*x2 == 100) \&\& (x0*x0*x1*x1*x2 == 200) \&\& (x0*x0*x1*x1*x2 == 300)$	1800	0.03
$(x0*x0*x1*x1+x2 < 100) \| \| (x0*x0*x1*x1+x2 < 200) \| \| (x0*x0*x1*x1+x2 < 300)$	2700	0.06
$(x0*x1*x2+x0 < 100) \&\& (x0*x1*x2+x0 < 200) \&\& (x0*x1*x2+x0 < 300)$	3000	0.09
$(x0*x1+x0*x2+x0 == 100) \&\& (x0*x1+x0*x2+x0 == 200) \&\& (x0*x1+x0*x2+x0 == 300)$	5082	0.11
$(x0*x0*x1*x2+x0 == 100) \&\& (x0*x0*x1*x2+x0 == 200) \&\& (x0*x0*x1*x2+x0 == 300)$	7056	0.12
$(x0*x0+x1*x1*x2 == 100) \&\& (x0*x0+x1*x1*x2 == 200) \&\& (x0*x0+x1*x1*x2 == 300)$	7200	0.14
$(x0*x1+x2 < 100) \| \| (x0*x1+x2 < 200) \| \| (x0*x1+x2 < 300)$	7680	0.19
$(x0*x0+x1*x1+x2 == 100) \&\& (x0*x0+x1*x1+x2 == 200) \&\& (x0*x0+x1*x1+x2 == 300)$	8360	0.18
$(x0+x1+x2 < 100) \| \| (x0+x1+x2 < 200) \| \| (x0+x1+x2 < 300)$	10125	0.22
$(x0*x1*x2*x3 == 100) \&\& (x0*x1*x2*x3 == 200)$	20000	0.38
$(x0*x0*x1*x1*x2+x3 == 100) \| \| (x0*x0*x1*x1*x2+x3 == 200)$	22000	0.56
$(x0*x0*x1*x1+x2*x3 == 100) \&\& (x0*x0*x1*x1+x2*x3 == 200)$	28800	0.73
$(x0*x1+x2*x3 == 100) \| \| (x0*x1+x2*x3 == 200)$	41472	1.4
$(x0*x0*x1+x1*x2*x3 == 100) \| \| (x0*x0*x1+x1*x2*x3 == 200)$	62208	1.74
$((x0*x0*x1*x2)+x3 == 100) \&\& ((x0*x0*x1*x2)+x3 == 200)$	92160	3.21
$((x0*x0*x1)+x2+x3 == 100) \| \| ((x0*x0*x1)+x2+x3 == 200)$	105300	3.54
$(x0*x0+x1*x1+x2*x3 == 100) \&\& (x0*x0+x1*x1+x2*x3 == 200)$	113680	3.57
$((x0*x0)+(x1*x2)+x3 == 100) \| \| ((x0*x0)+(x1*x2)+x3 == 200)$	149688	5.12
$(x0*x0+x1*x1+x2+x3 == 100) \| \| (x0*x0+x1*x1+x2+x3 == 200)$	173264	5.14
$(x0*x1+x2+x3 == 100) \&\& (x0*x1+x2+x3 == 200)$	180000	6.23
$(x0*x1*x2*x3 == 100) \&\& (x0*x1*x2*x3 == 200) \&\& (x0*x1*x2*x3 == 300)$	30000	0.75
$(x0*x0*x1*x1*x2+x3 == 100) \| \| (x0*x0*x1*x1*x2+x3 == 200) \| \| (x0*x0*x1*x1*x2+x3 == 300)$	33000	0.96
$(x0*x0*x1*x1+x2*x3 == 100) \&\& (x0*x0*x1*x1+x2*x3 == 200) \&\& (x0*x0*x1*x1+x2*x3 == 300)$	43200	1.17
$(x0*x1+x2*x3 == 100) \| \| (x0*x1+x2*x3 == 200) \| \| (x0*x1+x2*x3 == 300)$	62208	2.67
$(x0*x0*x1+x1*x2*x3 == 100) \| \| (x0*x0*x1+x1*x2*x3 == 200) \| \| (x0*x0*x1+x1*x2*x3 == 300)$	93312	3.16
$(x0*x1*x2+x3 == 100) \&\& (x0*x1*x2+x3 == 200) \&\& (x0*x1*x2+x3 == 300)$	122880	6.23
$((x0*x0*x1*x2)+x3 == 100) \| \| ((x0*x0*x1*x2)+x3 == 200) \| \| ((x0*x0*x1*x2)+x3 == 300)$	138240	6.42
$((x0*x0*x1)+x2+x3 == 100) \&\& ((x0*x0*x1)+x2+x3 == 200) \&\& ((x0*x0*x1)+x2+x3 == 300)$	157950	6.53
$(x0*x0+x1*x1+x2*x3 == 100) \| \| (x0*x0+x1*x1+x2*x3 == 200) \| \| (x0*x0+x1*x1+x2*x3 == 300)$	163856	6.58
$((x0*x0)+(x1*x2)+x3 == 100) \&\& ((x0*x0)+(x1*x2)+x3 == 200) \&\& ((x0*x0)+(x1*x2)+x3 == 300)$	220968	8.61
$(x0*x0+x1*x1+x2+x3 == 100) \| \| (x0*x0+x1*x1+x2+x3 == 200) \| \| (x0*x0+x1*x1+x2+x3 == 300)$	251664	8.68
$(x0*x1+x2+x3 == 100) \| \| (x0*x1+x2+x3 == 200) \| \| (x0*x1+x2+x3 == 300)$	270000	13.36

```
7:          $p \leftarrow l \cap o$ {Compute the intersection of the two subspaces}
8:          $p.truth \leftarrow f_{mergerules}(l.truth, o.truth)$ {See **Fig. 11**}
9:          $S_p.push(p)$
10:      **end for**
11: **end for**
12: $S_p.merge()$
13: return $S_p$
```

Using the *not* logical operator and the merge algorithms for logical operations *and* and *or*, the DAG representation is recursively merged in a bottom-up traversal. Fig. 15 shows the result of merging the spaces of Fig. 10 in three steps. Fig. 15(a) shows the initial state after evaluating the leaf nodes, Fig. 15(b) shows the result after applying the ! operator and Fig. 15(c) and (d) shows the result after merging using && and || operators.

VI. EXPERIMENTS

We tested our tool, using two different approaches. In the first approach, we picked some random simple and complex conditions from Mediabench [25] applications. In the second approach we evaluated our tool using some synthetic examples with more aggressive combination of supported arithmetic and logical operators. The results of these two sets of experiments are in the following subsections.

A. Mediabench Examples

In our first set of experiments, we randomly selected a number of simple and complex conditions from Mediabench applications [25]. Table II gives some basic statistics for the selected conditions, namely, the total number of simple and complex conditions (#Exp), average number of variables per condition

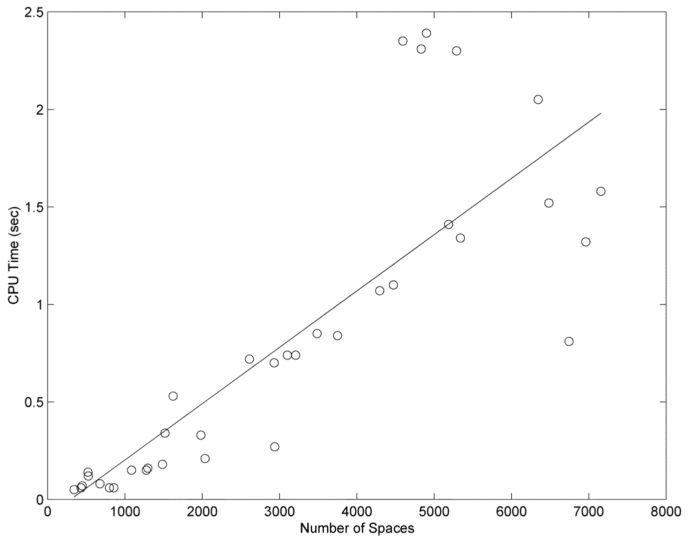

Fig. 16. Time versus number of spaces: $\#\mathrm{Var.} = 4$.

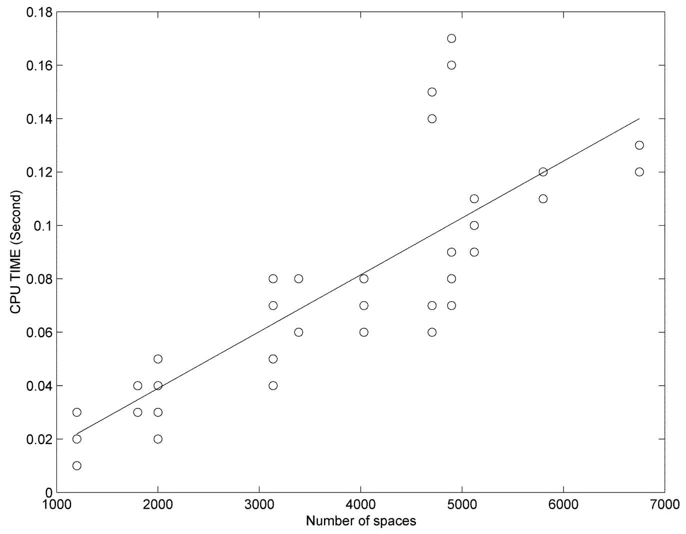

Fig. 18. Time versus number of spaces: $\#\mathrm{Var.} = 3$, $\#\mathrm{Rel\ Op} = 2$, and $\#\mathrm{Logic\ Op} = 1$.

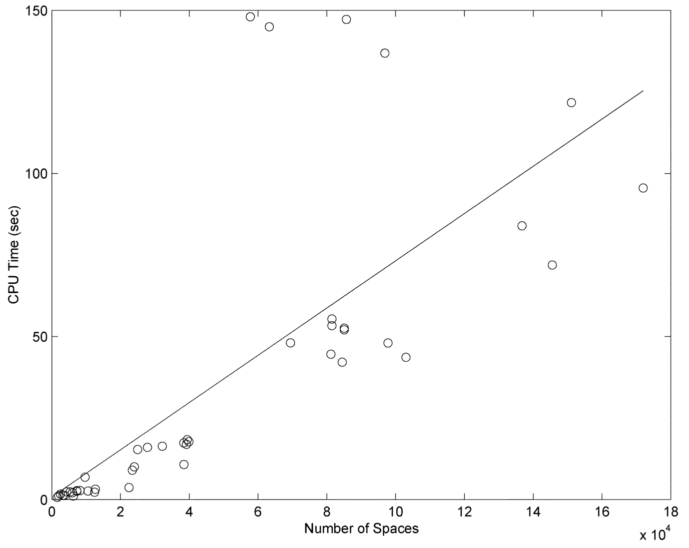

Fig. 17. Time versus number of spaces: $\#\mathrm{Var.} = 5$.

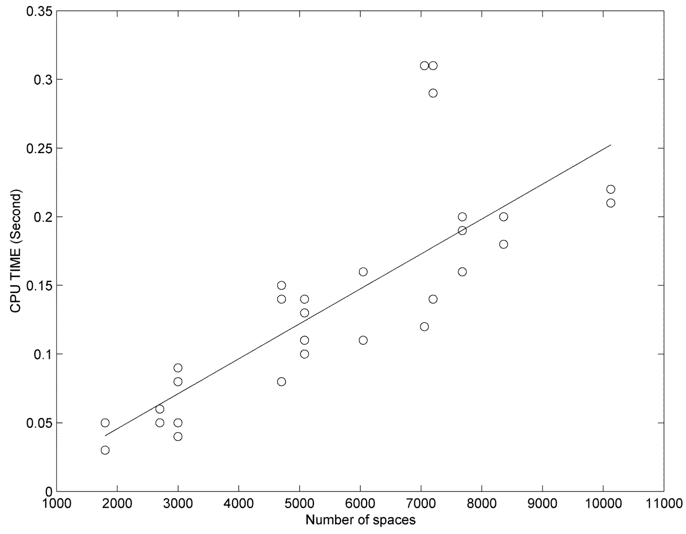

Fig. 19. Time versus number of spaces: $\#\mathrm{Var.} = 3$, $\#\mathrm{Rel\ Op} = 3$, and $\#\mathrm{Logic\ Op} = 2$.

(Avg. #Var), average number of arithmetic operations per condition (Avg. #Arith), average number of logical operations per condition (Avg. #Logic), and the average CPU time for evaluating a condition (Time).

Table III shows the ratio of truth values for Mediabench examples, as computed by our technique. On the average, about 92.7% of the whole domain of each condition is evaluated to *true* or *false* and about 7.30% is evaluated to *unknown*. Note that the portion of the domain space that is evaluated to *true* or *false* (i.e., 92.7%), represent the amount of pruning (with respect to evaluating the condition for all possible domain values) achieved by our algorithm. Conversely, the portion of the domain space that is evaluated to *unknown* (i.e., 7.30%) would require exhaustive evaluation to resolve the truth value of the condition. In cases where we have large *unknown* spaces, what can be done is as follows: if the space $[l_0, u_0][l_1, u_1]\ldots[l_n, u_n]$ is evaluated to *unknown*, we can divide it into 2^n smaller spaces by dividing each of the interval $[l_i, u_i]$ in space into two interval $[l_i, (l_i + u_i)/2]$ and $[(l_i + u_i)/2 + 1, u_i]$, and then apply the domain space partitioning algorithm for each resulting space separately.

B. Synthetic Examples

In our second set of experiments, we evaluated our tool using some synthetic examples with a more aggressive combination of supported arithmetic operators. We generated a total of 500 synthetic single and complex conditions; of those, a partial list is presented in Tables IV and V. Tables IV and V give some basic statistics for the synthetic simple and complex conditions, namely, the actual example (Single/Complex Condition), the generated number of unmerged spaces (#Spaces), and the CPU time for evaluating the synthetic single or complex condition (Time). In our strategy for generating these examples, we considered the number of variables ranging from 1 to 5, the number of arithmetic operations ($+$, $-$, $\times$) from 1 to 5, the number of relational operators from 2 to 3 and the number of logical operators from 1 to 2. The variables in expressions are 32-b integer.

Figs. 16 and 17 show the CPU time for running our algorithm on those simple condition examples with four or five variables.

Figs. 18–21 show the CPU time for running our algorithm on those complex condition examples with three or fpir vari-

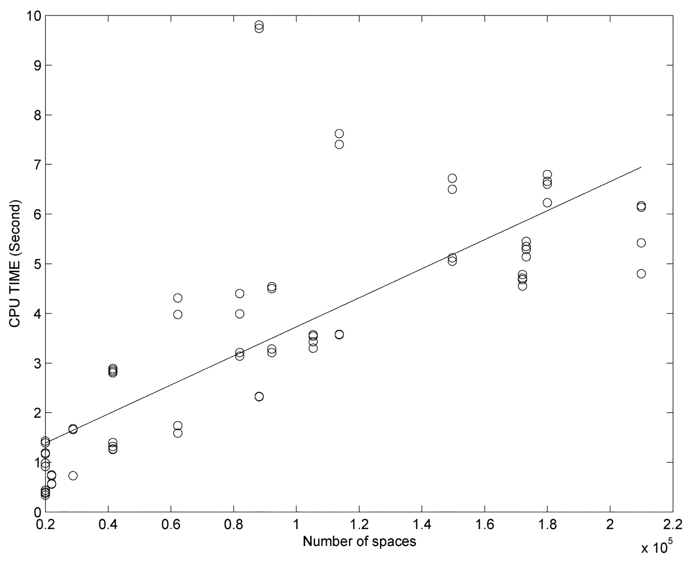

Fig. 20. Time versus number of spaces: $\#\mathrm{Var.} = 4$, $\#\mathrm{Rel\ Op} = 2$, and $\#\mathrm{Logic\ Op} = 1$.

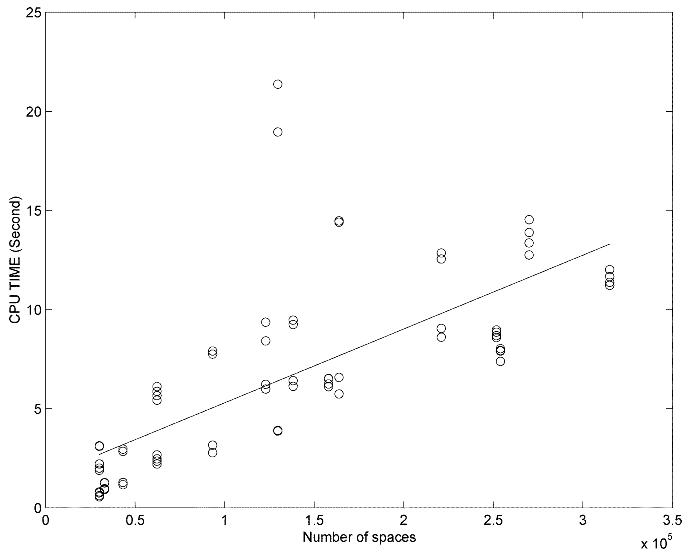

Fig. 21. Time versus number of spaces: $\#\mathrm{Var.} = 4$, $\#\mathrm{Rel\ Op} = 3$, and $\#\mathrm{Logic\ Op} = 2$.

ables, two or three relational operators, and one or two logical operators. Our results show that the CPU time for running our algorithm is proportional to the number of spaces into which the domain of the condition that is being evaluated is partitioned.

The number of spaces depends on the complexity of the arithmetic expression and the number of variables in it. Fig. 22 shows the dependency between the number of variables and the number of spaces for four equations $x_0 + x_1 = 100$, $x_0 + x_1 + x_2 = 100$, $x_0 + x_1 + x_2 + x_3 = 100$, and $x_0 + x_1 + x_2 + x_3 + x_4 = 100$. As the number of variables increases, the number of spaces increases exponentially and so does the time for running the domain space partitioning algorithm. Our experiments show that our heuristic can be applied on those kind of arithmetic equations which have at most eight number of variables, and this is fair enough for typical expressions found in software/hardware designs.

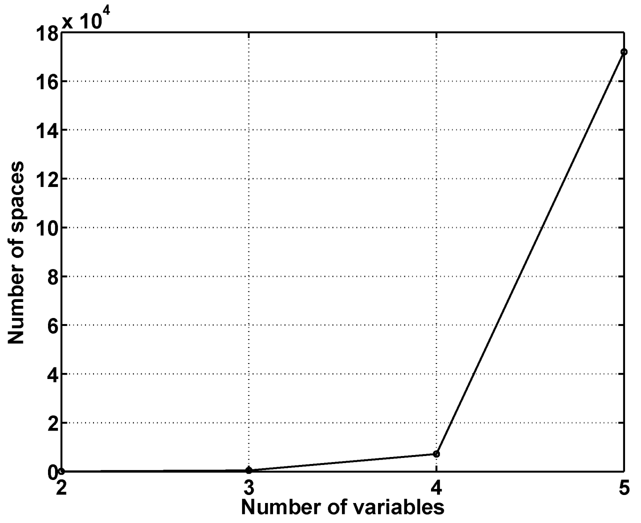

Fig. 22. Number of variables versus number of spaces.

VII. Conclusion

In this paper, we have proposed a method for solving the expression equivalence problem using partial evaluation. In our method, we used interval analysis to substantially prune the domain space of arithmetic expressions (and conditional expressions) and limited the evaluation effort to a sufficiently small number of minimally sized spaces within the domain of the expression. Then, we extend the technique to incorporate arbitrary use of logic operators *and*, *or*, and *not* within arithmetic expressions. Our results show that the proposed method is fast enough to be of use in practice.

For future work, we plan to first add the division operator to our grammar. For this, we can use the notion of rational polynomials (P/Q) and extend interval arithmetic accordingly by applying domain space partitioning method to P and Q individually and merging the spaces. Next, we can easily extend our grammar to include the shift operators ($\ll$, $\gg$), because both of these can be implemented using multiplication and division. Finally, we plan to consider bit-wise logical operators perhaps by considering mapping of these operators to the previously defined arithmetic operators.

References

[1] N. Dershowitz, "Rewrite systems," in *Handbook of Theoretical Computer Science*. Dordrecht, The Netherlands: Elsevier, 1990.
[2] J. Ferrante and C. Rackoff, "The computational complexity of logical theories," in *Lecture Notes in Mathematics*. New York: Springer-Verlag, 1979, vol. 718.
[3] P. Downey, R. Sethi, and R. Tarjan, "Variations on the common subexpression problem," *J. ACM*, vol. 27, no. 4, pp. 758–771, 1980.
[4] R. Drechsler, *Advanced Formal Verification*. Boston, MA, The Nederlands: Kluwer, 2004.
[5] E. Clarke, K. McMillan, X. Zhao, M. Fujita, and J. Yang, "Spectral transforms for large boolean functions with applications to technology mapping," in *Proc. Design Autom. Conf.*, Jun. 1993, pp. 54–60.
[6] R. Camposano, "Path-based scheduling for synthesis," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 10, no. 1, pp. 85–93, Jan. 1991.
[7] V. Chaiyakul, D. Gajski, and L. Ramachandran, "High-level transformations for minimizing syntactic variances," in *Proc. Design Autom. Conf.*, Jun. 1993, pp. 413–418.
[8] G. D. Micheli, *Synthesis and Optimization of Digital Circuits*. New York: McGraw-Hill, 1994.

[9] A. Aho, R. Sethi, and J. Ullman, *Compilers Principles, Techniques and Tools*. Reading, MA: Addison-Wesley, 1988.

[10] R. Moore, *Interval Analysis*. Englewood Cliffs, NJ: Prentice-Hall, 1966.

[11] S. Akers, "Binary decision diagrams," *IEEE Trans. Comput.*, vol. C-27, no. 6, pp. 509–516, Jun. 1978.

[12] Z. Z. M. Ciesielski, P. Kalla, and B. Rouzeyre, "Taylor expansion diagrams: A compact, canonical representation with applications to symbolic verification," in *Proc. Conf. Design, Autom. Test Europe*, 2002, pp. 285–289.

[13] R. Drechsler, *Formal Verification of Circuits*. Dordrecht, The Nederlands: Kluwer, 2000.

[14] R. Brinkmann and R. Drechsler, "Rtl-datapath verification using integer linear programming," in *Proc. Conf. Asia South Pacific Design Autom./VLSI Design*, 2002, pp. 741–746.

[15] S. Horeth and R. Drechsler, "Formal verification of word-level specifications," in *Proc. Conf. Design, Autom. Test Europe*, 1999, pp. 52–58.

[16] K. Wakabayashi and T. Yoshimura, "A resource sharing and control synthesis method for conditional branches," in *Proc. Int. Conf. Comput.-Aided Design*, 1989, pp. 62–65.

[17] H. Juan, V. Chaiyakul, and D. D. Gajski, "Condition graphs for high-quality behavioral synthesis," in *Proc. Int. Conf. Comput.-Aided Design*, 1994, pp. 170–174.

[18] J. Li and R. Gupta, "An algorithm to determine mutually exclusive operations in behavioral descriptions," in *Proc. Conf. Design, Autom. Test Europe*, 1998, pp. 457–465.

[19] ——, "Hdl pre-synthesis optimizations using a tabular model," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 8, no. 4, pp. 369–387, Apr. 2000.

[20] Y. Xie and W. Wolf, "Allocation and scheduling of conditional task graph in hardware/software co-synthesis," in *Proc. Conf. Design, Autom. Test Europe*, 2001, pp. 620–625.

[21] Z. Zhou and W. Burleson, "Equivalence checking of datapaths based on canonical arithmetic expressions," in *Proc. Design Autom. Conf.*, 1995, pp. 546–551.

[22] N. Cheung, S. Parameswaran, J. Henkel, and J. Chan, "Mince: Matching instructions using combinational equivalence for extensible processor," in *Proc. Conf. Design, Autom. Test Europe*, 2004, pp. 1020–1025.

[23] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C*. Cambridge, U.K.: Cambridge Univ. Press, 1992.

[24] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *Proc. ACM SIGMOD Int. Conf. Management of Data*, 1984, pp. 47–57.

[25] C. Lee, M. Potkonjak, and W. Mangione-Smith, "Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems," in *Proc. Int. Symp. Microarchitecture*, 1997, pp. 330–335.

Mohammad Ali Ghodrat (S'04) received the B.S. degree from Amirkabir University of Technology, Tehran, Iran, in 1996, and the M.S. degree from the University of Tehran, Tehran, Iran, in 1999, both in computer engineering. He is currently working toward the Ph.D. degree at the School of Computer Science and Information, University of California, Irvine.

He is a member of the Center for Embedded Computer System, University of California, Irvine. From 1999 to 2003, he was an ASIC Design Engineer. His research interests are algorithmic transformations for embedded software, compiler for multiprocessor systems on chip, and instruction generation/selection for ASIP.

Tony Givargis (M'99) received the B.S. and Ph.D. degrees from the University of California, Riverside, in 1997 and 2001, respectively.

He is an Assistant Professor with the Department of Computer Science and a member of the Center for Embedded Computer Systems, University of California, Irvine. He has published over 45 conference and journal articles on the topic of embedded system design, with a specific emphasis on embedded software design. He is also the coauthor of the book *Embedded System Design: A Unified Hardware/Software Introduction* (Wiley, 2001).

Alex Nicolau (M'92) received the Ph.D. degree in computer science from Yale University, New Haven, CT, in 1984.

He is a Professor with the Center for Embedded Computer Systems, University of California, Irvine, with academic appointments in the Information and Computer Science and Electrical Engineering Departments. His current research interests include embedded systems, computer architecture, and optimizing compilers. He is the coauthor of two books, *Memory Issues in Embedded Systems-on-Chip: Optimizations and Exploration* (Kluwer, 1999) and *Memory Architecture Exploration for Programmable Embedded Systems* (Kluwer, 2002), and ten edited volumes. He has published over 200 journal and conference papers in the above areas. He serves as Editor-in-Chief of the *International Journal of Parallel Programming* and has served on the program and steering committees of numerous conferences in the field, notably ICS, PACT, and Micro.