

# Expression Equivalence Checking using Interval Analysis

Mohammad Ali Ghodrat

Tony Givargis

Technical Report CECS-05-06

July 20, 2005

Center for Embedded Computer Systems

University of California, Irvine

Irvine, CA 92697-3425, USA

(949) 824-8168

{mghodrat, givargis}@cecs.uci.edu

# Expression Equivalence Checking using Interval Analysis

Mohammad Ali Ghodrat

Tony Givargis

Technical Report CECS-05-06

July 20, 2005

Center for Embedded Computer Systems

University of California, Irvine

Irvine, CA 92697-3425, USA

(949) 824-8168

{mghodrat, givargis}@cecs.uci.edu

## 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 efforts to a sufficiently limited set of subspaces. Our results show that the proposed method is fast enough to be of use in practice.*

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Problem Definition</b>	<b>4</b>
<b>3</b>	<b>Domain Space Partitioning</b>	<b>5</b>
3.1	Computing Root-spaces . . . . .	6
3.2	Partitioning . . . . .	9
3.3	Evaluation . . . . .	10
3.4	Merging . . . . .	11
<b>4</b>	<b>Experiments</b>	<b>12</b>
4.1	Mediabench Examples . . . . .	12
4.2	Synthetic Examples . . . . .	14
<b>5</b>	<b>Conclusion</b>	<b>15</b>

## List of Figures

1	Partitioned Domain of $C : 2x_0 + x_1 + 4 > 0$ . . . . .	5
2	Space Partitioning Strategy . . . . .	6
3	Root-spaces of $2x_0 + x_1 + 4$ . . . . .	9
4	Partitioned Spaces for $2x_0 + x_1 + 4$ . . . . .	10
5	Evaluated Subspaces for $2x_0 + x_1 + 4 > 0$ . . . . .	11
6	Merged Spaces for $2x_0 + x_1 + 4 > 0$ . . . . .	12
7	Time vs. Number of Spaces - #Var.=4 . . . . .	15
8	Time vs. Number of Spaces - #Var.=5 . . . . .	16

# Expression Equivalence Checking using Interval Analysis

Mohammad Ali Ghodrat, Tony Givargis

Center for Embedded Computer Systems

University of California, Irvine

Irvine, CA 92697-3425, USA

{mghodrat,givargis}@cecs.uci.edu

<http://www.cecs.uci.edu>

## 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 efforts to a sufficiently limited set of subspaces. Our results show that the proposed method is fast enough to be of use in practice.

## 1 Introduction

Arithmetic expressions are the fundamental building blocks of hardware and software systems. In hardware, arithmetic expressions form the core of data-path 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 [11, 8]. 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 [7].

Efficiently solving the equivalence problem between two arithmetic expressions will have a profound impact in the areas of formal verification [10], complex code generation and technology mapping [6], resource scheduling [3], code transformation [4], synthesis technologies [17], and compiler techniques [1].

In this paper we propose a method for solving the expression equivalence problem using partial evaluation. In our method, we use interval analysis [18] to substantially prune the domain space of arithmetic expressions and limit the evaluation efforts to a limited set of subspaces. Our results show that the proposed method is fast enough to be of use in practice.

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 (BDD) [2] and its derivatives, namely ordered BDD (OBDD), ordered functional decision diagrams (OFDD), multi terminal BDD (MTBDD), binary moment diagram (BMD), edge-valued BDD (EVBDD), and multiplicative BMD (\*BMD). These approaches differ mainly in bit vs. 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 represents 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 (MTBDD and BMD) or edges (EVBDD, \*BMD and K\*BMD). A detailed survey of BDD and its derivatives can be found in [9].

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 dia-

grams can be applied to simple arithmetic expressions (e.g. datapath segments [12]), 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 (less-than, greater-than and equal) present in those regions.

In related work, Wakabayashi et al. [20] 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. [13] proposed *condition graphs*, a form of syntax pattern matching, to find mutual exclusion between two restricted boolean conditions. Further, Jian et al. [15, 16] used *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. [21] used a *branch labeling* method to find the mutual exclusion properties between two boolean expressions. Finally, Camposano [3], in his path-based scheduling technique, proposed a method for determining mutual exclusion based on a 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 work. 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. [22] propose a formal verification system, called *conditional term rewriting on attribute syntax trees* (ConTRAST) for verifying the equivalence between two differently synthesized data-paths. In their approach, they maintain attributes (e.g., real bounds) associated with each node of the syntax trees of the two data-paths 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. [5] used bit-slicing of binary decision diagrams (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.

The remainder of this paper is organized as follows. In Section 2, we formulate the problem of expression equivalence. In Section 3, we give our interval analysis solution for this problem. In Section 4 we present our experimental results. Finally, in Section 5, we give our conclusion and future directions.

## 2 Problem Definition

An *arithmetic expression* is formed over the language  $(+, -, \times, \text{integer-constant}, \text{integer-variable})$ . A *simple condition* is in the form of  $(expr_1 \text{ ROP } expr_2)$ . Here,  $expr_1$  and  $expr_2$  are *arithmetic expressions* and  $\text{ROP}$  is a relational operator  $(=, \neq, <, \leq, >, \geq)$ . Without loss of generality we can assume all *simple conditions* to be of the form of  $(expr \text{ ROP } 0)$ . This normalization is achieved by converting  $(expr_1 \text{ ROP } expr_2)$  to  $(expr_1 - expr_2 \text{ ROP } 0)$ . Hence,  $(expr \text{ ROP } 0)$  is called a *normalized simple condition*. For the remainder of this work, 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  $[l_0, u_0] \times [l_1, u_1] \times \dots \times [l_{n-1}, u_{n-1}]$ . In a *simple condition*, all integer-constants and integer-variables are assumed to be bounded between *min* and *max* values<sup>1</sup>. Hence, the domain of a *simple condition*  $C$  with  $n$  integer-variables  $x_0, x_1, \dots, x_{n-1}$  is an  $n$ -dimensional space defined by the cartesian product  $[min, max] \times [min, max] \times \dots \times [min, max]$ .

Given a *simple condition*  $C$  with integer-variables  $x_0, x_1, \dots, x_{n-1}$ , the *domain space partitioning problem* is to partition the domain space of  $C$  into a minimal set of  $n$ -dimensional spaces  $s_1, s_2, \dots, s_k$  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.

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 2-dimensional space defined by the cartesian product  $[-5, 5] \times [-5, 5]$ . Figure 1

---

<sup>1</sup>Typically, in a computer system, *min* and *max* values are determined by the width of the processor data-path.



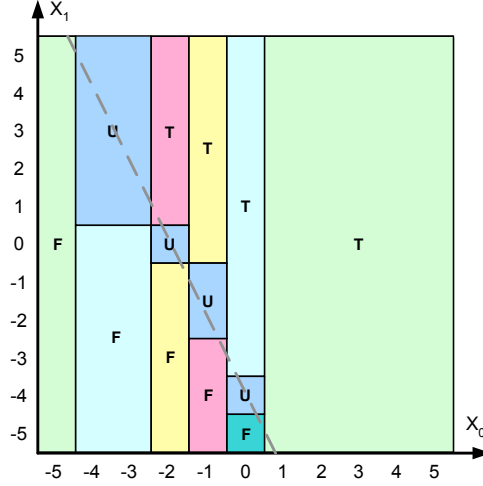


Figure 1: Partitioned Domain of  $C : 2x_0 + x_1 + 4 > 0$

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. Specifically, given two expressions  $E_1$  and  $E_2$ , by evaluating the condition  $E_1 - E_2 = 0$ , we can establish the equivalence of  $E_1$  and  $E_2$  (i.e.,  $E_1$  and  $E_2$  are not equivalent if the condition evaluates to false for a point in the domain of  $E_1$  and  $E_2$ ).

### 3 Domain Space Partitioning

Our overall domain space partitioning strategy is depicted in Figure 2. On input, the *arithmetic expression* of the *simple condition* is parsed to obtain an equivalent polynomial representation. Any arbitrary *arithmetic expression* can be rewritten as an  $n$ -variable polynomial with degree  $D$  using the general form shown in Equation 1.

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

For example, the expression  $2 \times x_0 + x_1 + 4$  of Figure 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

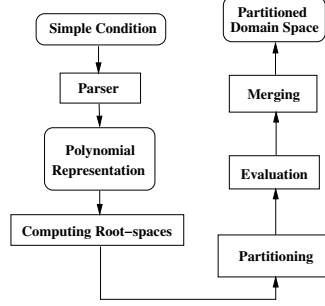


Figure 2: Space Partitioning Strategy

domain space partitioning steps in the following subsections.

### 3.1 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* [18]. 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  $(+, -, \times, /)$  can be defined on two real intervals  $[a, b]$  and  $[c, d]$  as shown below:

$$[a, b] + [c, d] = [a + c, b + d] \quad (2)$$

$$[a, b] - [c, d] = [a - d, b - c] \quad (3)$$

$$[a, b] \times [c, d] = [\min(a \times c, a \times d, b \times c, b \times d), \quad (4)$$

$$\max(a \times c, a \times d, b \times c, b \times d)]$$

$$[a, b]/[c, d] = \begin{cases} [a, b] \times [1/d, 1/c] & 0 \notin [c, d] \\ [-\infty, \infty] & 0 \in [c, d]. \end{cases} \quad (5)$$

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:  $S \leftarrow \langle [min, max], [min, max], \dots, [min, max] \rangle$   $\{|space| = n\}$ 
5:  $Q.push(S)$ 
6: while  $Q.not\_empty()$  do
7:    $S \leftarrow Q.pop()$ 
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:         $Q.push(\langle v_0, v_1, \dots, r, \dots, v_{n-1} \rangle)$  {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 6-22):** We pop a space  $S$  from the queue  $Q$  and split  $S$  into smaller spaces  $S_0, S_2, \dots, S_{k-1}$ . If  $S_0 \cup S_1 \dots \cup S_{k-1} = S$ , then  $S$  can not be minimized, thus, we add  $S$  to the output list of root-spaces  $R$ . If  $S_0 \cup S_1 \dots \cup S_{k-1} \subset S$ , then we push  $S_0, S_1, \dots, S_{k-1}$  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., Newton-Raphson Method [19]) 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 for 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  $r_0, r_1, \dots$  to the present value of  $x_i$  in space  $S$ , namely  $v_i$ . If any root  $r_0, r_1, \dots$  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 12-17). In our running example, root  $r_0 = [-4.5, .5]$  is not equal to  $v_0 = [-5, 5]$ , thus we create a new space  $S_0 = \langle [-4.5, -.5], [-5, 5] \rangle$ .
  - (d) Once steps (b) and (c) are completed, if the flag *changed* is not set,  $S$  can not be further minimized, thus we push it on the output set  $R$  (lines 19-21).

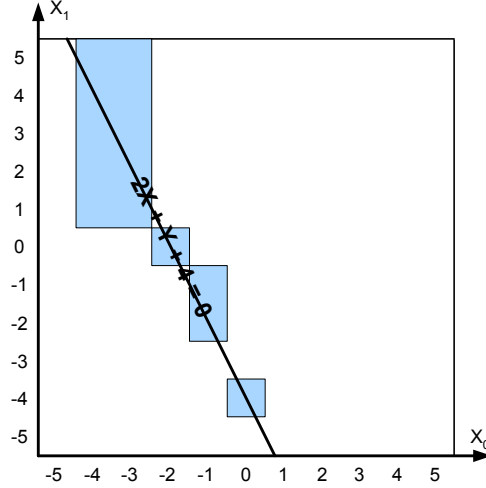


Figure 3: Root-spaces of  $2x_0 + x_1 + 4$

3. **Quantization Phase (lines 23-25):** Finally, we convert each root-space in the output set  $R$  to the smallest bounding integer space. Table 1 gives the final output set  $R$  for our running example. This result is shown graphically in Figure 3. All the shaded areas are the root-spaces, and as shown in Figure 3, the equation  $2x_0 + x_1 + 4 = 0$  passes through all of them.

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

Table 1: Root-spaces of  $2x_0 + x_1 + 4$

### 3.2 Partitioning

Given the root-spaces for an expression  $E_x$  (corresponding to a *normalized simple condition*  $E_x ROP0$ ), 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 1), partition the entire domain

space as shown in Figure 4. In Figure 4 the root-spaces are shown in shaded color.

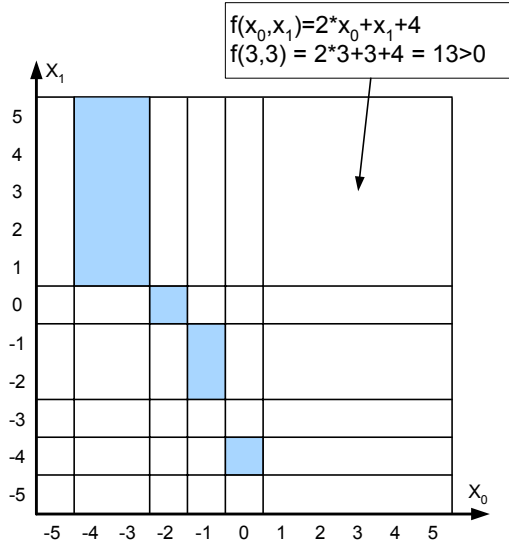


Figure 4: Partitioned Spaces for  $2x_0 + x_1 + 4$

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 the root-spaces). In Figure 4, 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 the points in space  $\langle [1, 5][1, 5] \rangle$ .

### 3.3 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 Figure 5). 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 Figure 5).

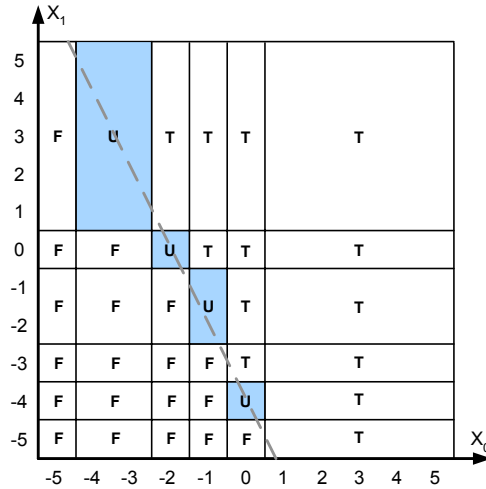


Figure 5: Evaluated Subspaces for  $2x_0 + x_1 + 4 > 0$

### 3.4 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 Figure 5, 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., Figure 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 a *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  $O(k^3)$ .

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  $O(k^2)$ .

Figure 6 shows the result of merge operation on Figure 5.

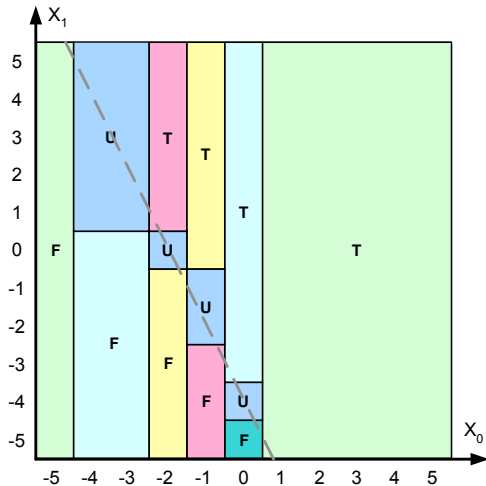


Figure 6: Merged Spaces for  $2x_0 + x_1 + 4 > 0$

## 4 Experiments

We tested our tool, using two different approaches. In the first approach we picked some random *simple conditions* from Mediabench [14] applications. In the second approach we evaluated our tool using some synthetic examples with more aggressive combination of supported arithmetic operators. The results of these two sets of experiments are in the following subsections:

### 4.1 Mediabench Examples

In our first set of experiments, we randomly selected a number of *simple conditions* from Mediabench applications [14]. Table 2 gives some basic statistics for the selected *simple conditions*, namely, the total number of *simple conditions* (#Exp), average number of variables per *simple condition* (Avg.



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

Table 3 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 *simple 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 *simple 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 *simple condition*.

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

Table 2: Statistics for Mediabench Examples

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

Table 3: Results for Mediabench Examples

<i>Simple Condition</i>	#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

Table 4: Partial List of Synthetic Examples

## 4.2 Synthetic Examples

In our second set of experiments, we evaluated our tool using some synthetic examples with more aggressive combination of supported arithmetic operators. We generated a total of 250 synthetic *simple conditions*, of those, a partial list is presented in Table 4. Table 4 gives some basic statistics for the synthetic *simple conditions*, namely, the actual example (*Simple Condition*), the generated number of spaces (#Spaces), and the CPU time for evaluating the synthetic *simple condition* (Time). In our strategy for generating these examples, we considered the number of variables ranging from 1 to 5 and the number of arithmetic operations (+, −, ×) from 1 to 5.

Figure 7 and Figure 8 show the CPU time for running our algorithm on those examples with four or five variables. Our results show that the CPU time for running our algorithm is proportional to the number of spaces into which the domain of the *simple condition* that is being evaluated is

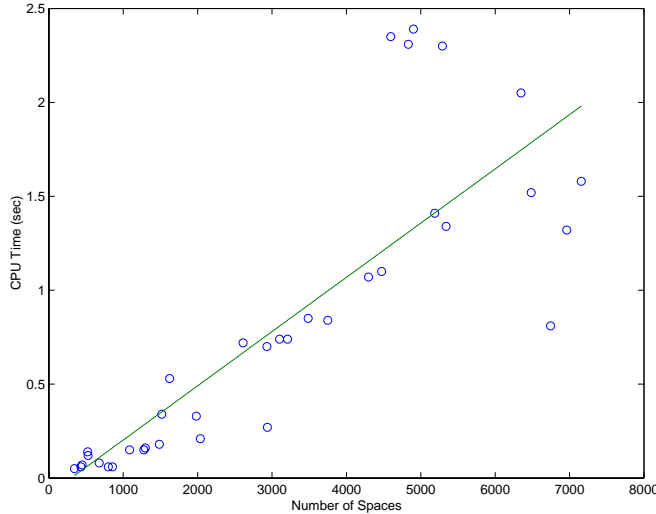


Figure 7: Time vs. Number of Spaces – #Var.=4

partitioned.

## 5 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 efforts to a sufficiently small number of minimally sized spaces within the domain of the expression. Our results show that the proposed method is fast enough to be of use in practice.

Our future work will focus on optimizing the various stages of the presented algorithm to further improve the running time, incorporate the division operator, and extend the technique to include complex conditions that are formed using logical operators (&&, ||, !).

## References

- [1] A. Aho, R. Sethi, and J. Ullman. *Compilers principles, techniques and tools*. Addison Wesley, Reading, Massachusetts, 1988.
- [2] S.B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, 27(6):509–516, 1978.

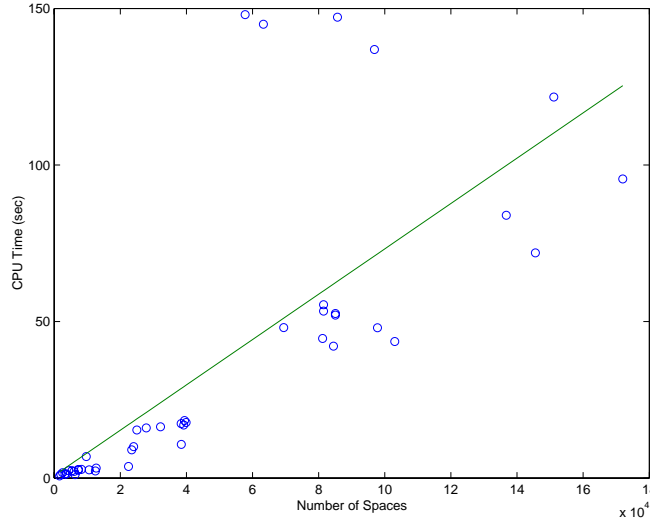


Figure 8: Time vs. Number of Spaces – #Var.=5

- [3] R. Camposano. Path-based scheduling for synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 10(1):85–93, 1991.
- [4] V. Chaiyakul, D. Gajski, and L. Ramachandran. High-level transformations for minimizing syntactic variances. In *Design Automation Conference*, June 1993.
- [5] N. Cheung, S. Parameswaran, J. Henkel, and J. Chan. Mince: matching instructions using combinational equivalence for extensible processor. In *Conference on Design, Automation and Test in Europe*, pages 1020–1025, 2004.
- [6] E.M. Clarke, K.L. McMillan, X. Zhao, M. Fujita, and J. Yang. Spectral transforms for large boolean functions with applications to technology mapping. In *Design Automation Conference*, June 1993.
- [7] N. Dershowitz. Rewrite systems. *Handbook of Theoretical Computer Science, Elsevier Science Publishers*, 1990.
- [8] P.J. Downey, R. Sethi, and R.E. Tarjan. Variations on the common subexpression problem. *Journal of the ACM*, 27(4):758–771, 1980.

- [9] R. Drechsler. *Formal verification of circuits*. Kluwer Academic Publishers, The Netherlands, 2000.
- [10] R. Drechsler. *Advanced formal verification*. Kluwer Academic Publisher, The Netherlands, 2004.
- [11] J. Ferrante and C.W. Rackoff. The computational complexity of logical theories. *Lecture Notes in Mathematics*, 718, 1979.
- [12] S. Horeth and R. Drechsler. Formal verification of word-level specifications. In *Conference on Design, Automation and Test in Europe*, pages 52 – 58, 1999.
- [13] H.P. Juan, V. Chaiyakul, and D. D. Gajski. Condition graphs for high-quality behavioral synthesis. In *International Conference on Computer-Aided Design*, pages 170–174, 1994.
- [14] C. Lee, M. Potkonjak, and W.H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *International Symposium on Microarchitecture*, pages 330–335, 1997.
- [15] J. Li and R.K. Gupta. An algorithm to determine mutually exclusive operations in behavioral descriptions. In *Conference on Design, Automation and Test in Europe*, pages 457 – 465, 1998.
- [16] J. Li and R.K. Gupta. Hdl pre-synthesis optimizations using a tabular model. *IEEE Transactions on Very Large Scale Integration Systems*, 8(4):369–387, 2000.
- [17] G. De Micheli. *Synthesis and optimization of digital circuits*. McGraw Hill, Hightstown NJ, 1994.
- [18] R.E. Moore. *Interval analysis*. Prentice-Hall, Englewood Cliffs, N. J., 1966.
- [19] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical recipes in C*. Cambridge University Press, University of Cambridge, 1992.
- [20] K. Wakabayashi and T. Yoshimura. A resource sharing and control synthesis method for conditional branches. In *International Conference on Computer-Aided Design*, pages 62–65, 1989.

- [21] Y. Xie and W. Wolf. Allocation and scheduling of conditional task graph in hardware/software co-synthesis. In *Conference on Design, Automation and Test in Europe*, pages 620–625, 2001.
- [22] Z. Zhou and W. Burch. Equivalence checking of datapaths based on canonical arithmetic expressions. In *Design Automation Conference*, pages 546 – 551, 1995.