Equivalence Checking of Arithmetic Expressions using Fast Evaluation

Mohammad Ali Ghodrat

Tony Givargis

Technical Report CECS-05-07

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

Equivalence Checking of Arithmetic Expressions using Fast Evaluation

Mohammad Ali Ghodrat

Tony Givargis

Technical Report CECS-05-07

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, which consists of regular arithmetic operators (+, −, ×) and logical operators (and, or, not). 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.

Contents

List of Figures

Equivalence Checking of Arithmetic Expressions using Fast Evaluation

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, which consists of regular arithmetic operators ($+$, $-$, $\times$) and logical operators (*and*, *or*, *not*). 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.

1 Introduction and motivation

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 [18], compiler techniques [1], behavioral synthesis tools, reconfigurable computing methodologies, extensible processors, VLIW and multiple-processor-on-a-chip compilers.

In this paper we propose a method for solving the expression equivalence problem using partial evaluation. In our method, we use interval analysis [19] to substantially prune the domain space of arithmetic expressions and limit the evaluation effort to a limited set of subspaces. Our results show that the proposed method is fast enough to be of use in practice.

The high compute demand of media rich, networked, and mobile embedded devices, combined with low power consumption constraints, continue to push the limits of design at all levels of the hierarchy. For maximum performance and energy efficiency, an application-specific integrated circuit (ASIC) can often yield the ideal results. However, in the context of short time-to-market windows, support for multiple complex communication and computation standards, and ability to perform dynamic firmware updates, a design can not rely on ASIC solutions alone. In an effort to address these issues, processor vendors have introduced programmable platforms that combine complex application-specific kernels with one or more general-purpose processors (GPPs) on a single chip (a.k.a., system-on-a-chip (SOC)). Here, in some cases, the application-specific kernels are tightly embedded in the GPP's datapath (e.g., Xtensa/Tensilica) while in others kernels are integrated in a more traditional fashion (e.g., A7V05/Triscend).

One problem of interest is to realize the compute capability and efficiency of such SOCs by novel program synthesis techniques. In particular, given an SOC platform S defined as a set of resources $r_1, r_2...r_n$, resource conflict sets $C_1, C_2...C_m$, computation cost function $F(r_i)$, communication cost function $G(r_i, r_j)$, and an application program P, we seek to map (i.e., synthesize/compile) program P onto SOC S to achieve maximum performance and energy efficiency. Both resource r_i and program P are defined behaviorally in terms of a *high-level program*. A resource r_i may be a simple processor instruction (e.g., `add`) or a complex function (e.g., `crc-decode`) – furthermore, two resources r_i and r_j, may implement the same function. A conflict set $C_i = \{r_a, r_b...\}$ indicates that $r_a, r_b...$ can not be used in parallel. Functions F and G define the cost, in cycles, of using resource r_i or passing data from r_i to r_j (in case of data dependency).

At first glance, the above stated problem may appear familiar to the many problems in logic synthesis, RTL-synthesis, and behavioral synthesis. However, it differs from the these design methods in a fundamental way. While in logic/RTL/behavioral-synthesis the objective is to transform the design from a high-level of abstraction to lower and lower levels of abstractions (i.e., behavioral→RTL→logic) followed by binding to a small set of simple logic-gates, we seek to perform a mapping from a high-level program to another high-level program. Such program mapping involves non-trivial *algorithmic transformations*, *scheduling*, and extensive *design-space exploration*. Existing efforts so far have approached the problem in terms of pattern-matching and simple algorithm manipulations. Unfortunately, pattern matching techniques fail in most, including trivial, cases. Figure 1 shows a simple example where pattern matching does not work, although both codes implement the same behavior. The reason for this is that the two expression $x < 0$ and $x > 255$ can never be true at the same time, in other word $x < 0$ and $x > 255$ are mutual exclusive. This gives the need to find the mutual exclusion property between any two arithmetic and boolean expressions.

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.

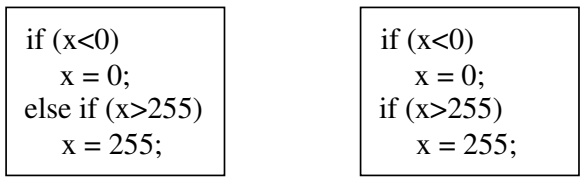

Figure 1: A simple example that shows pattern matching does not work always

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 the 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.

The remainder of this paper is organized as follows. In Section 2 we show the previous works. In Section 3, we formulate the problem of expression equivalence. In Section 4, we give our solution for this problem when we have only one simple arithmetic expression. In Section 5 we extend our solution for more complex arithmetic expressions which have boolean operators also. In Section 6 we present our experimental results. Finally, in Section 7, we give our conclusion.

2 Previous works

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 diagrams can be applied to simple arithmetic expressions (e.g. datapath segments [13]), 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. [21] 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. [14] have proposed *condition graphs*, a form of syntax pattern matching, to find mutual exclusion between two restricted boolean conditions. Further, Jian et al. [16, 17] have 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. [22] used a *branch labeling* method to find the mutual exclusion properties between two boolean expressions. Finally, Camposano [3], 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 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. [23] have proposed 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] have 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.

3 Problem Definition

An *arithmetic expression* is formed over the language (+, −, ×, 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 (=, ≠, <, ≤, >, ≥). 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 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 $[l_0, u_0] \times [l_1, u_1] \times ... \times [l_{n-1}, u_{n-1}]$. 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, ..., x_{n-1} is an n-dimensional space defined by the cartesian product $[min, max] \times [min, max] \times ... \times [min, max]$.

Given a *simple condition* C with integer-variables x_0, x_1, ..., 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 $s_1, s_2, ..., s_k$ 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,

[1] Typically, in a computer system, *min* and *max* values are determined by the width of the processor data-path.

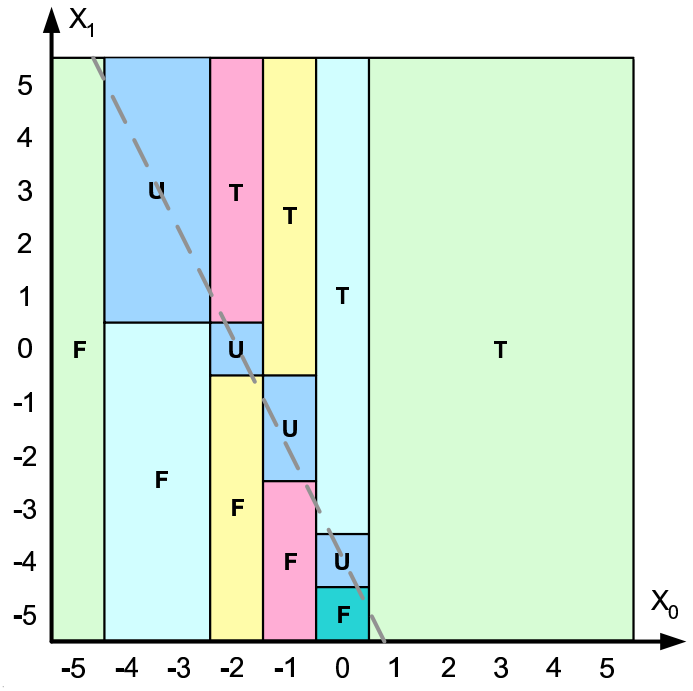

Figure 2: Partitioned Domain of $C : 2x_0 + x_1 + 4 > 0$

the domain of C is a 2-dimensional space defined by the cartesian product $[-5, 5] \times [-5, 5]$. Figure 2 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). We give our solution to the *domain space partitioning problem for a simple condition* in section 4.

A *complex condition* is either a *simple condition* or two *complex conditions* merged using *logical operators* (&&, ||, !). 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, ..., x_{n-1} is an n-dimensional space defined by the cartesian product $[min, max] \times [min, max] \times ... \times [min, max]$.

Similar to the *domain space partitioning problem for simple conditions*, given a *complex condition* C with integer-variables x_0, x_1, ..., 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 $s_1, s_2, ..., s_k$ 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-bit two's complement integers. We can construct the following *complex condition*:

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

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-bit 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 5.

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

4 Domain Space Partitioning for Simple Condition

Our overall domain space partitioning strategy is depicted in Figure 3. 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.

$$\sum_{i_0,i_1,...,i_{n-1}=0}^{D} c_{i_0,i_1,...,i_{n-1}} \times x_0^{i_0} \times x_1^{i_1} \times ... \times x_{n-1}^{i_{n-1}} \tag{1}$$

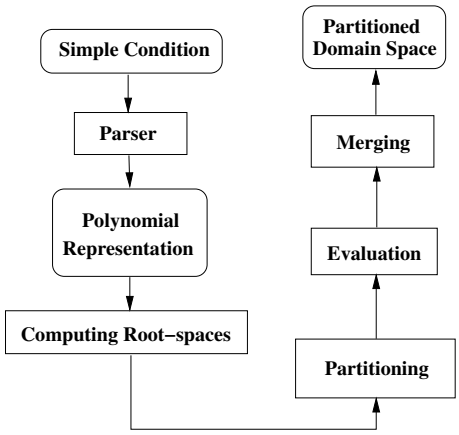

Figure 3: Space Partitioning Strategy

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

4.1 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* [19]. 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:

$$[a,b]+[c,d] = [a+c, b+d] \tag{2}$$

$$[a,b]-[c,d] = [a-d, b-c] \tag{3}$$

$$\begin{aligned}[a,b]\times[c,d] = [&\min(a\times c, a\times d, b\times c, b\times d),\\ &\max(a\times c, a\times d, b\times c, b\times d)]\end{aligned} \tag{4}$$

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

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

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 $S_0, S_2, ..., S_{k-1}$. If $S_0 \cup S_1 ... \cup S_{k-1} = S$, then S can not be minimized, thus, we add S to the output list of root-spaces R. If $S_0 \cup S_1 ... \cup S_{k-1} \subset S$, then we push $S_0, S_1, ..., S_{k-1}$ 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 [20]), implemented using interval

Algorithm 1 Compute Root-spaces

Input: a n-variable polynomial P
Output: a set R of minimally sized root-spaces
$R \leftarrow \varnothing$
$S \leftarrow \langle [min, max], [min, max], ..., [min, max] \rangle$ $\{|space| = n\}$
Q.push(S)
while Q.not_empty() **do**
  $S \leftarrow Q$.pop()
  $changed \leftarrow 0$
  for all $x_i \in P$ **do**
    $P' \leftarrow$ convert P to a polynomial with x_i as the only variable and $x_j = v_j$ $(v_j \in S)$
    $roots \leftarrow P'$.solve()
    for all $r \in roots$ **do**
      if $r \neq v_i$ $(v_i \in S)$ **then**
        $changed \leftarrow 1$
        $r \leftarrow r \cap v_i$ {Intersect new root with previous one}
        Q.push($\langle v_0, v_1, ..., r, ..., v_{n-1} \rangle$) {replace v_i with r}
      end if
    end for
  end for
  if $changed = 0$ **then**
    $R \leftarrow R \cup \{S\}$
  end if
end while
for all $R_i \in R$ **do**
  $R_i \leftarrow$ convert R_i to smallest bounding integer space
end for

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 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 $r_0, r_1, ...$ to the present value of x_i in space S, namely v_i. If any root $r_0, r_1, ...$ 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 $r_0 = [-4.5, .5]$ 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 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.

3. **Quantization Phase (lines 24-26):** 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 4. All the shaded areas are the root-spaces, and as shown in Figure 4, the equation $2x_0 + x_1 + 4 = 0$ passes through all of them.

4.2 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

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

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

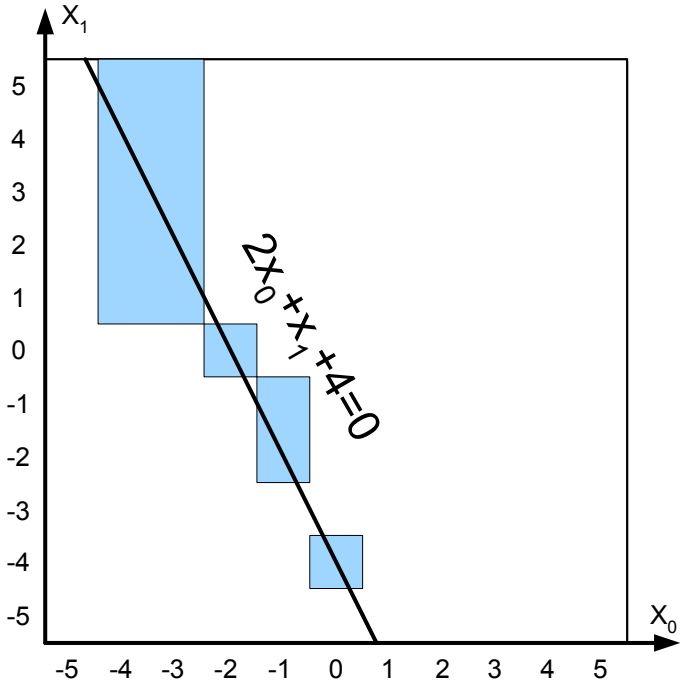

Figure 4: Root-spaces of $2x_0 + x_1 + 4$

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 5. In Figure 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 the root-spaces). In Figure 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.

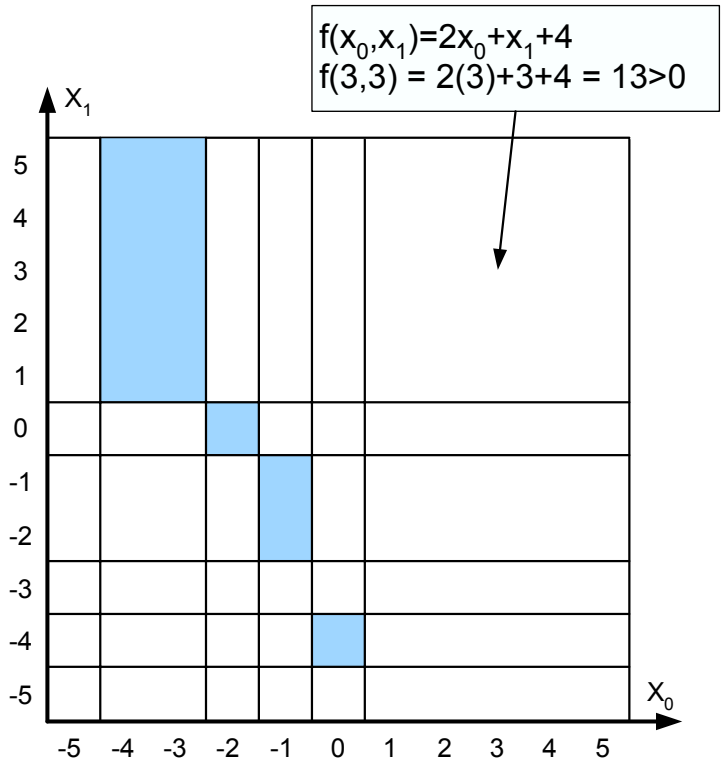

Figure 5: Partitioned Spaces for $2x_0 + x_1 + 4$

Furthermore, this is true for all the points in space $\langle[1,5][1,5]\rangle$.

4.3 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 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 Figure 6).

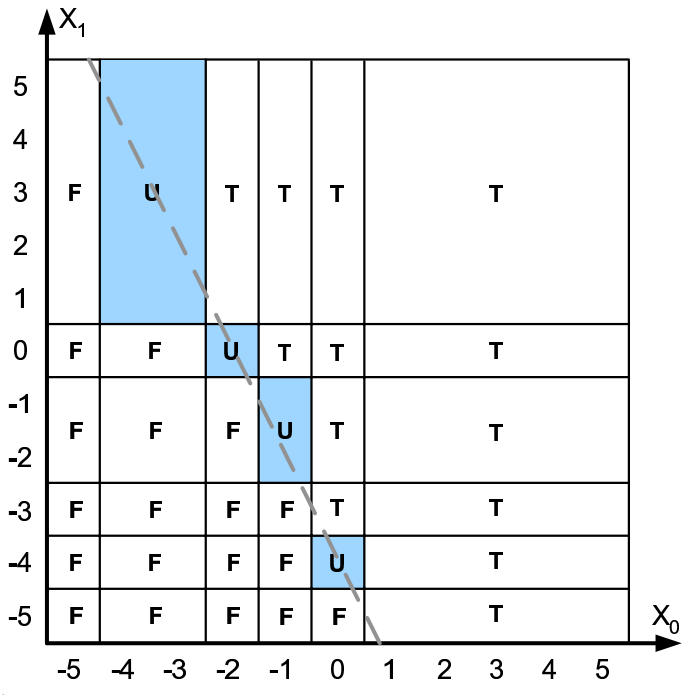

Figure 6: Evaluated Subspaces for $2x_0 + x_1 + 4 > 0$

4.4 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 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., Figure 3), 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 $O(k^3)$.

Alternatively, we can use a divide-and-conquer heuristic to do this in $O(k^2)$. The idea is to sub divide 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 7 shows the result of merge operation on Figure 6.

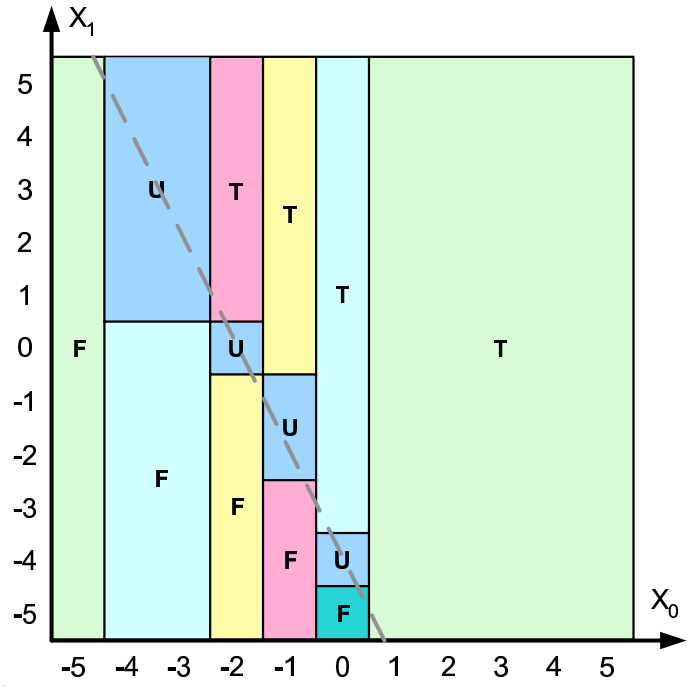

Figure 7: Merged Spaces for $2x_0 + x_1 + 4 > 0$

5 Domain Space Partitioning for Complex Condition

Our overall strategy for solving the *domain space partitioning problem for complex conditions* is depicted in Figure 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.

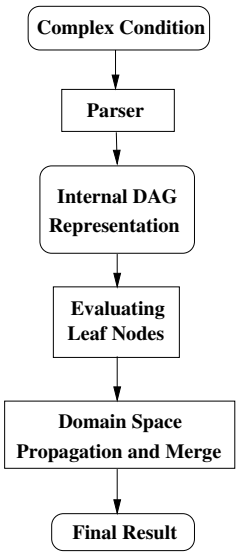

Figure 8: Solution Strategy for Domain Space Partitioning for Complex Condition

5.1 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 3, the *simple condition* is captured as a *multi-variable 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 Figure 9.

5.2 Evaluating Leaf Nodes

Each leaf node in the DAG representation is a *simple condition* and is evaluated as outlined in section 4. Specifically, each leaf node in the DAG representation corresponds to one instance of the *domain space partitioning problem for simple conditions*. Figure 10 shows the partitioned domain spaces for the leaf nodes of our running example.

5.3 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 Figure 11. These rules define how two sets of domain spaces are combined under the logical operators (&&, ||, !).

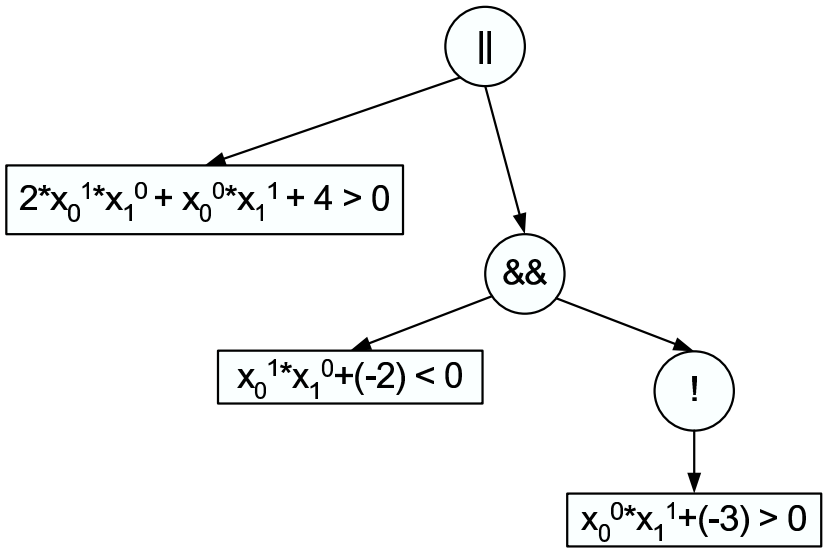

Figure 9: The DAG Representation

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. Figure 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 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 s_p is computed using the merge rules given in Figure 11. This procedure is shown in Algorithm 2. Figure 13 shows an example of the logical *and* merging of two partitioned domain spaces. In Figure 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, 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 [12] to make the search job faster. An R-tree as defined in [12] is a height-balanced tree suitable for handling spatial data in multidimensional spaces. Figure 14

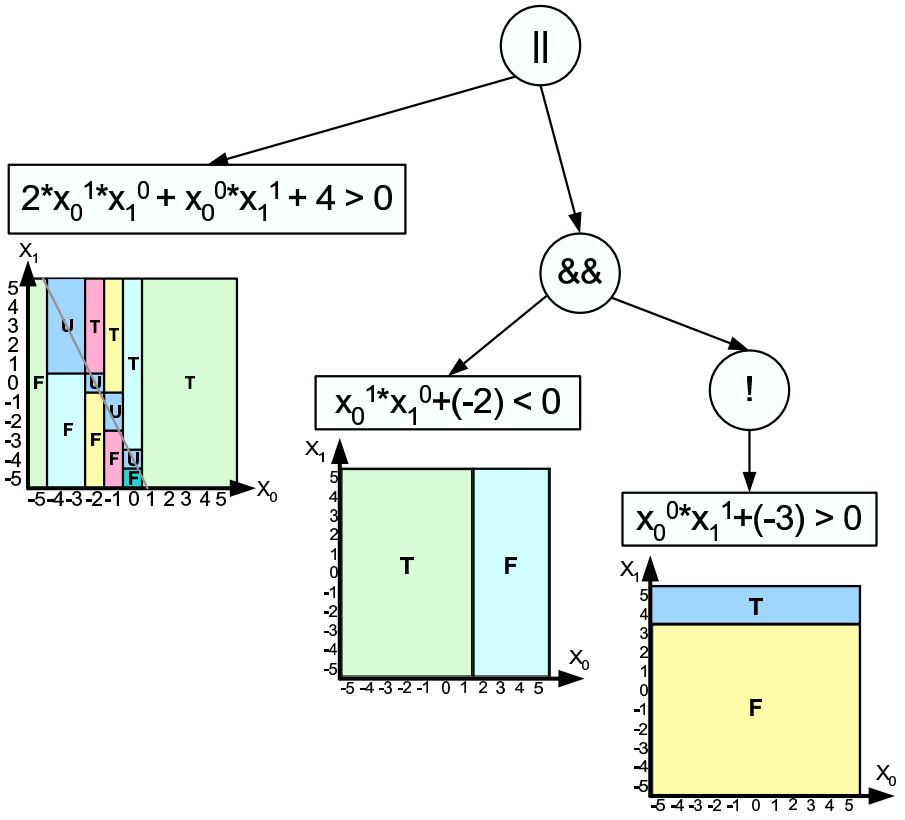

Figure 10: Partitioned Domain Spaces for Leaf Nodes

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.

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. Figure 15 shows the result of merging the spaces of Figure 10 in three steps. Figure 15(a) shows the initial state after evaluating the leaf nodes, Figure 15(b) shows the result after applying the ! operator and Figure 15(c) and Figure 15(d) show the result after merging using && and || operators.

Algorithm 2 Logical-AND Space Merging-Exhaustive Method

Input: Partitioned domain spaces S_l and S_r
Output: Merged domain space S_p
for all *spaces* $l \in S_l$ **do**
  for all *spaces* $r \in S_r$ **do**
    $p \leftarrow l \cap r$ {Compute the intersection of the two subspaces}
    if $(p \neq \phi)$ **then**
      $p.truth \leftarrow f_{mergerules}(l.truth, r.truth)$ {See Figure 11}
      $S_p.push(p)$
    end if
  end for
end for
$S_p.merge()$
return S_p

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

Input: Partitioned domain spaces S_l and S_r
Output: Merged domain space S_p
rT = make an R-tree using S_r
for all *spaces* $l \in S_l$ **do**
  overlappedRegion = rT.overlap(l)
  for all *spaces* $o \in overlappedRegion$ **do**
    $p \leftarrow l \cap o$ {Compute the intersection of the two subspaces}
    $p.truth \leftarrow f_{mergerules}(l.truth, o.truth)$ {See Figure 11}
    $S_p.push(p)$
  end for
end for
$S_p.merge()$
return S_p

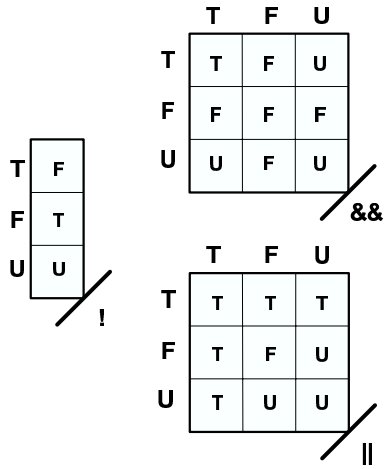

Figure 11: Merge Rules for Operators &&, ||, !

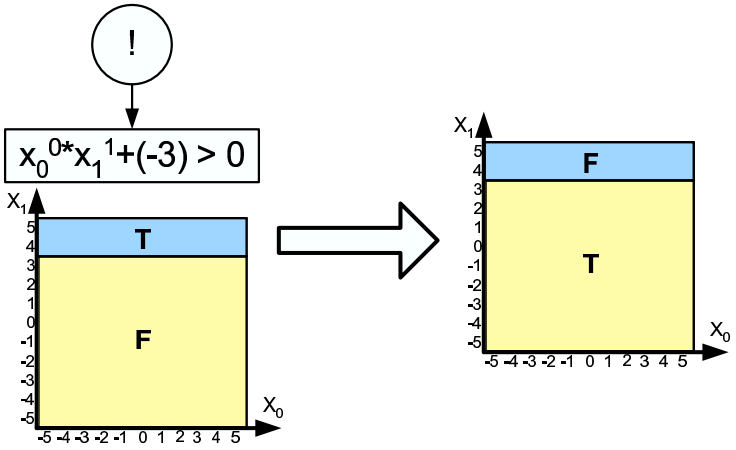

Figure 12: Applying Logical Not Operator (!) to Leaf Nodes

6 Experiments

We tested our tool, using two different approaches. In the first approach we picked some random *simple and complex conditions* from Mediabench [15] 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:

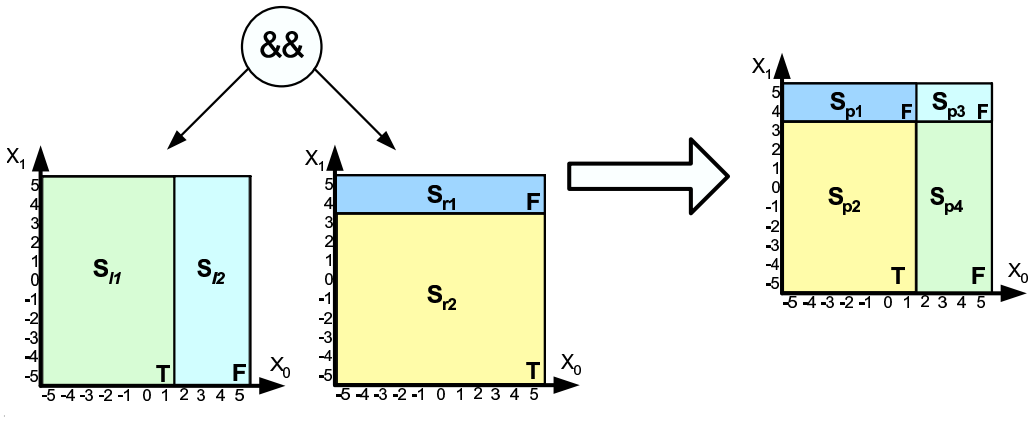

Figure 13: Applying Logical And Operator (&&) to Leaf Nodes

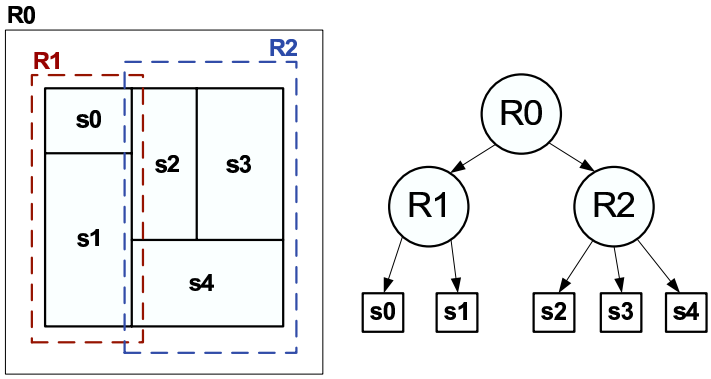

Figure 14: Partitioned Domain Space Representation Using R-tree

6.1 Mediabench Examples

In our first set of experiments, we randomly selected a number of *simple and complex conditions* from Mediabench applications [15]. Table 2 gives some basic statistics for the selected conditions, namely, the total number of *simple and complex conditions* (#Exp), average number of variables per condition (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 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 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.

Table 2: 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 3: 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

6.2 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 500 synthetic *single and complex conditions*, of those, a partial list is presented in Table 4 and Table 5. Table 4

Table 4: 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

and Table 5 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.

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

Figure 18 to Figure 21 show the CPU time for running our algorithm on those *complex condition* examples with 3 or 4 variables, 2 or 3 relational operators and 1 or 2 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.

7 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.

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.

[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 Nederlands, 2000.

[10] R. Drechsler. *Advanced formal verification.* Kluwer Academic Publisher, The Nederlands, 2004.

[11] J. Ferrante and C.W. Rackoff. The computational complexity of logical theories. *Lecture Notes in Mathematics*, 718, 1979.

[12] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, pages 47 – 57, 1984.

[13] S. Horeth and R. Drechsler. Formal verification of word-level specifications. In *Conference on Design, Automation and Test in Europe*, pages 52 – 58, 1999.

[14] 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.

[15] C. Lee, M. Potkonjak, and W.H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *International Symposium on Microarchitecture*, pages 330–335, 1997.

[16] 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.

[17] 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.

[18] G. De Micheli. *Synthesis and optimization of digital circuits.* McGraw Hill, Hightstown NJ, 1994.

[19] R.E. Moore. *Interval analysis*. Prentice-Hall, Englewood Cliffs, N. J., 1966.

[20] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical recipes in C*. Cambridge University Press, University of Cambridge, 1992.

[21] 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.

[22] 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.

[23] Z. Zhou and W. Burleson. Equivalence checking of datapaths based on canonical arithmetic expressions. In *Design Automation Conference*, pages 546 – 551, 1995.

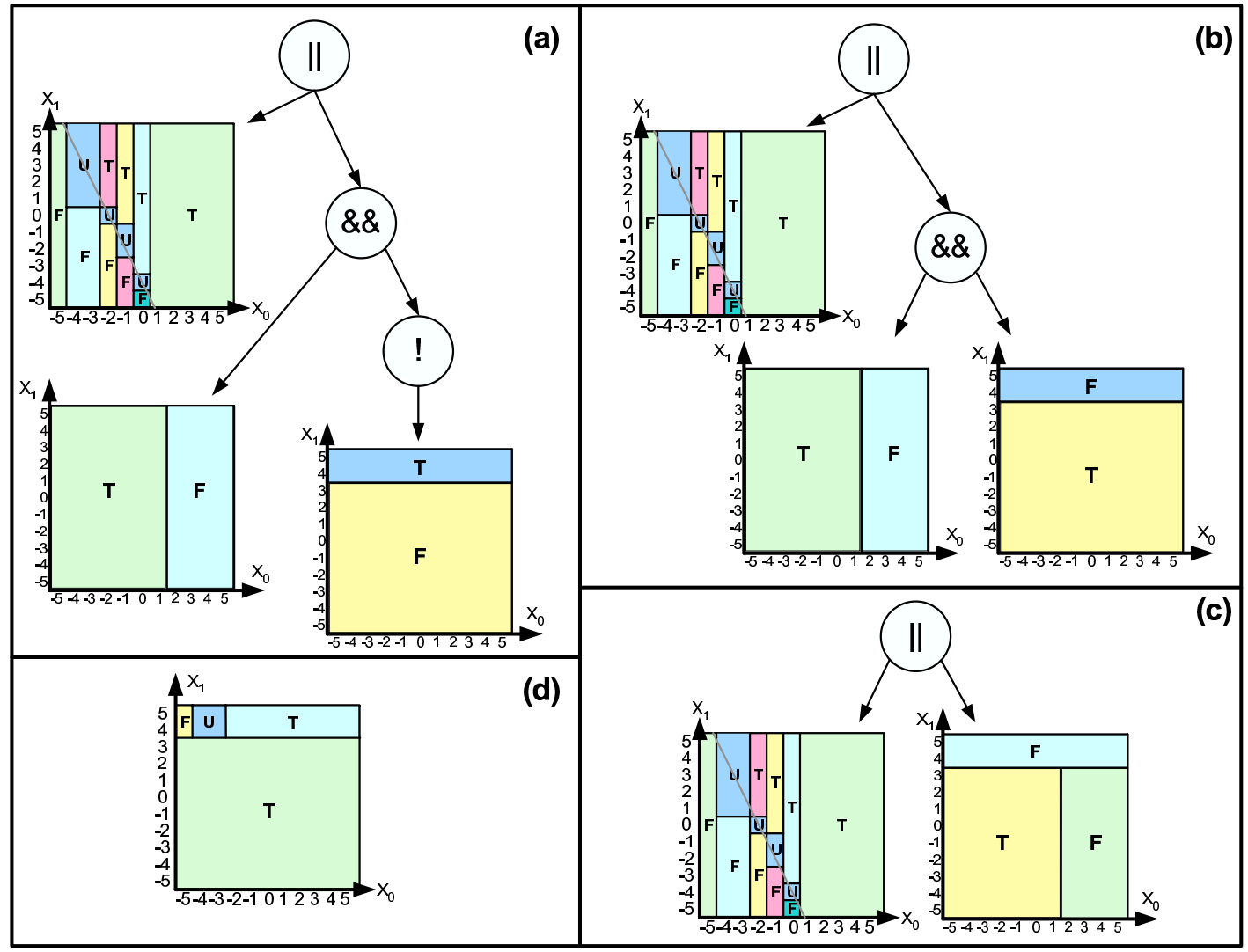

Figure 15: Merging and Propagation of Spaces for Figure 10: (a)- Initial State (b)- After Applying ! Operator (c)- After Merging Using && (d)- After Merging Using ||

Table 5: 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

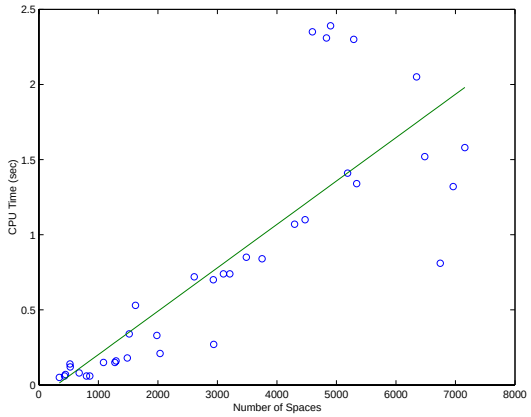

Figure 16: Time vs. Number of Spaces – #Var.=4

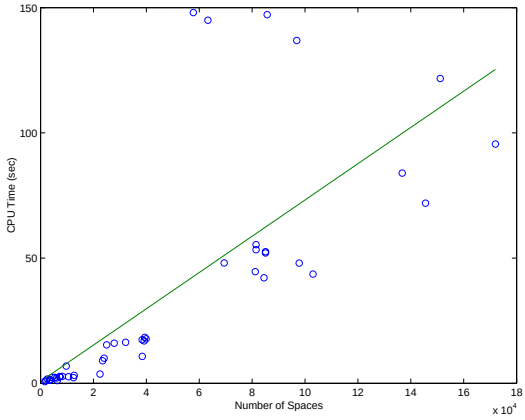

Figure 17: Time vs. Number of Spaces – #Var.=5

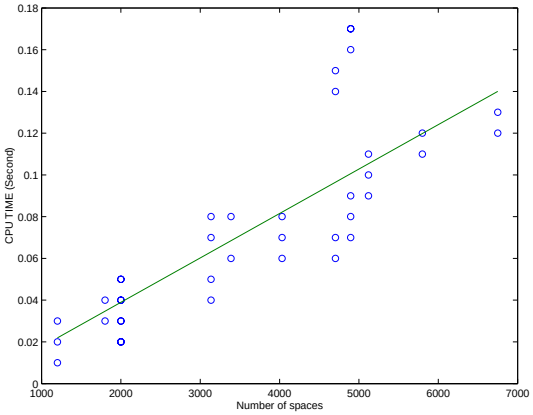

Figure 18: Time vs. Number of Spaces – #Var.=3, #Rel Op=2, #Logic Op=1

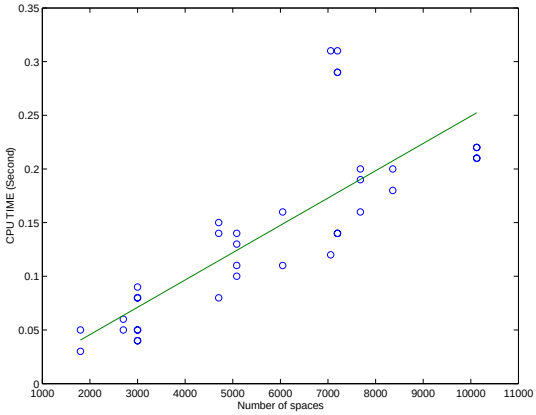

Figure 19: Time vs. Number of Spaces – #Var.=3, #Rel Op=3, #Logic Op=2

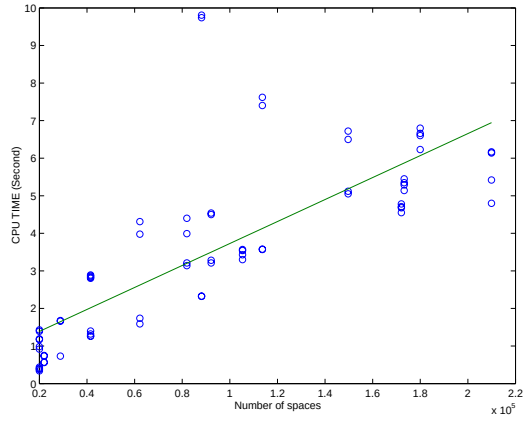

Figure 20: Time vs. Number of Spaces – #Var.=4, #Rel Op=2, #Logic Op=1

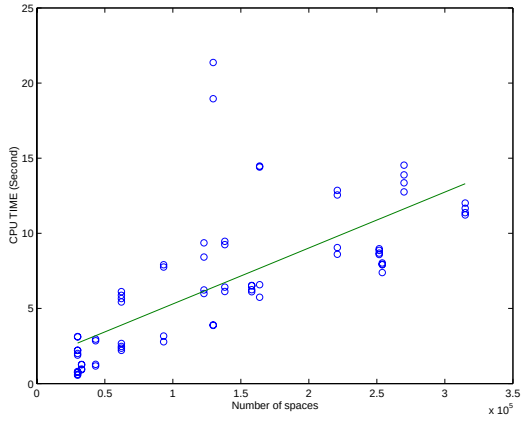

Figure 21: Time vs. Number of Spaces – #Var.=4, #Rel Op=3, #Logic Op=2