Fast Three-Valued Abstract Bit-Vector Arithmetic

Jan Onderka[1] and Stefan Ratschan[2]

2022-01-18

[1]Czech Technical University in Prague, Faculty of Information Technology, Prague, Czech Republic
[2]The Czech Academy of Sciences, Institute of Computer Science, Prague, Czech Republic

Introduction

- Three-valued abstraction = abstract bits can have value '0', '1', or 'X' (possibly 0, possibly 1)

Introduction

- Three-valued abstraction = abstract bits can have value '0', '1', or 'X' (possibly 0, possibly 1)
- How to generate a **single three-valued bit-vector operation output** for arithmetic operations quickly, with best results?
- For example, getting the best result $\hat{r}^{\mathrm{best}}$ for $\hat{a} + \hat{b}$:

$$\hat{a} = \text{``}\mathbf{X}0\text{''},\ \hat{b} = \text{``}11\text{''}$$

$$a = \mathbf{0}0_2, \quad b = 11_2, \quad r = \mathbf{01}1_2 = \mathbf{0}0_2 + 11_2$$

$$a = \mathbf{1}0_2, \quad b = 11_2, \quad r = \mathbf{10}1_2 = \mathbf{1}0_2 + 11_2$$

$$\hat{r}^{\mathrm{best}} = \text{``}\mathbf{XX}1\text{''}$$

Introduction

- Three-valued abstraction = abstract bits can have value '0', '1', or 'X' (possibly 0, possibly 1)
- How to generate a **single three-valued bit-vector operation output** for arithmetic operations quickly, with best results?
- For example, getting the best result $\hat{r}^{\text{best}}$ for $\hat{a} + \hat{b}$:

$$\hat{a} = \text{``}\mathbf{X}0\text{''},\ \hat{b} = \text{``}11\text{''}$$
$$a = \mathbf{0}0_2, \quad b = 11_2, \quad r = \mathbf{01}1_2 = \mathbf{0}0_2 + 11_2$$
$$a = \mathbf{1}0_2, \quad b = 11_2, \quad r = \mathbf{10}1_2 = \mathbf{1}0_2 + 11_2$$
$$\hat{r}^{\text{best}} = \text{``}\mathbf{XX}1\text{''}$$

- **Can the best result be obtained in polynomial time?**

Introduction

- Three-valued abstraction = abstract bits can have value '0', '1', or 'X' (possibly 0, possibly 1)
- How to generate a **single three-valued bit-vector operation output** for arithmetic operations quickly, with best results?
- For example, getting the best result $\hat{r}^{\text{best}}$ for $\hat{a} + \hat{b}$:

$$\hat{a} = \text{“}\mathbf{X}0\text{”},\ \hat{b} = \text{“}11\text{”}$$

$$a = \mathbf{0}0_2, \quad b = 11_2, \quad r = \mathbf{01}1_2 = \mathbf{0}0_2 + 11_2$$

$$a = \mathbf{1}0_2, \quad b = 11_2, \quad r = \mathbf{10}1_2 = \mathbf{1}0_2 + 11_2$$

$$\hat{r}^{\text{best}} = \text{“}\mathbf{XX}1\text{”}$$

- **Can the best result be obtained in polynomial time?**
- Let's go back to our motivation and formalization first...

Motivation

- Formally verifying conventional digital processor machine code: movement + bitwise logic + **wrap-around arithmetic** + branching

[3] T. Reinbacher, M. Horauer, and B. Schlich. "Using 3-valued memory representation for state space reduction in embedded assembly code model checking". In: *2009 12th International Symposium on Design and Diagnostics of Electronic Circuits Systems*. 2009, pp. 114–119. DOI: 10/dhg4ww.

[4] John Regehr and Alastair Reid. "HOIST: A System for Automatically Deriving Static Analyzers for Embedded Systems". In: *SIGOPS Oper. Syst. Rev.* 38.5 (Oct. 2004), pp. 133–143. ISSN: 0163-5980. DOI: 10/fjcx9w.

Motivation

- Formally verifying conventional digital processor machine code: movement + bitwise logic + **wrap-around arithmetic** + branching
- Explicit state space generation, trying to avoid exponential explosion $\rightarrow$ we want a single result

[3] T. Reinbacher, M. Horauer, and B. Schlich. "Using 3-valued memory representation for state space reduction in embedded assembly code model checking". In: *2009 12th International Symposium on Design and Diagnostics of Electronic Circuits Systems*. 2009, pp. 114–119. DOI: 10/dhg4ww.

[4] John Regehr and Alastair Reid. "HOIST: A System for Automatically Deriving Static Analyzers for Embedded Systems". In: *SIGOPS Oper. Syst. Rev.* 38.5 (Oct. 2004), pp. 133–143. ISSN: 0163-5980. DOI: 10/fjcx9w.

Motivation

- Formally verifying conventional digital processor machine code: movement + bitwise logic + **wrap-around arithmetic** + branching
- Explicit state space generation, trying to avoid exponential explosion $\rightarrow$ we want a single result
- Focusing on forward direction, **no backtracking**

[3] T. Reinbacher, M. Horauer, and B. Schlich. "Using 3-valued memory representation for state space reduction in embedded assembly code model checking". In: *2009 12th International Symposium on Design and Diagnostics of Electronic Circuits Systems*. 2009, pp. 114–119. DOI: 10/dhg4ww.

[4] John Regehr and Alastair Reid. "HOIST: A System for Automatically Deriving Static Analyzers for Embedded Systems". In: *SIGOPS Oper. Syst. Rev.* 38.5 (Oct. 2004), pp. 133–143. ISSN: 0163-5980. DOI: 10/fjcx9w.

Motivation

- Formally verifying conventional digital processor machine code: movement + bitwise logic + **wrap-around arithmetic** + branching
- Explicit state space generation, trying to avoid exponential explosion $\rightarrow$ we want a single result
- Focusing on forward direction, **no backtracking**
- Movement + bitwise logic can be performed in linear time (standard Kleene three-valued logic)[3]

---

[3] T. Reinbacher, M. Horauer, and B. Schlich. "Using 3-valued memory representation for state space reduction in embedded assembly code model checking". In: *2009 12th International Symposium on Design and Diagnostics of Electronic Circuits Systems*. 2009, pp. 114–119. DOI: 10/dhg4ww.

[4] John Regehr and Alastair Reid. "HOIST: A System for Automatically Deriving Static Analyzers for Embedded Systems". In: *SIGOPS Oper. Syst. Rev.* 38.5 (Oct. 2004), pp. 133–143. ISSN: 0163-5980. DOI: 10/fjcx9w.

Motivation

- Formally verifying conventional digital processor machine code: movement + bitwise logic + **wrap-around arithmetic** + branching
- Explicit state space generation, trying to avoid exponential explosion $\rightarrow$ we want a single result
- Focusing on forward direction, **no backtracking**
- Movement + bitwise logic can be performed in linear time (standard Kleene three-valued logic)[3]
- Wrap-around arithmetic: operation results can be precomputed for 8-bit inputs (stored using BDDs), infeasible for larger inputs[4]

[3] T. Reinbacher, M. Horauer, and B. Schlich. "Using 3-valued memory representation for state space reduction in embedded assembly code model checking". In: *2009 12th International Symposium on Design and Diagnostics of Electronic Circuits Systems*. 2009, pp. 114–119. DOI: 10/dhg4ww.

[4] John Regehr and Alastair Reid. "HOIST: A System for Automatically Deriving Static Analyzers for Embedded Systems". In: *SIGOPS Oper. Syst. Rev.* 38.5 (Oct. 2004), pp. 133–143. ISSN: 0163-5980. DOI: 10/fjcx9w.

Formalization of three-valued bit-vectors

- Abstract bit values formalized as

$$\text{`0'} := \{0\}, \text{`1'} := \{1\}, \text{`X'} := \{0, 1\} \tag{1}$$

[5] Institute of Electrical and Electronics Engineers. "IEEE Standard Multivalue Logic System for VHDL Model Interoperability (Std_logic_1164)". In: *IEEE Std 1164-1993* (1993), pp. 1–24. DOI: 10/bhz6s9.

Formalization of three-valued bit-vectors

- Abstract bit values formalized as

$$\text{`0'} := \{0\}, \text{`1'} := \{1\}, \text{`X'} := \{0, 1\} \tag{1}$$

- Abstract bit-vectors: tuples of abstract bits, IEEE 1164 notation[5]: "XX10" $= (\text{`X'}, \text{`X'}, \text{`1'}, \text{`0'}) = (\{0,1\}, \{0,1\}, \{1\}, \{0\})$

[5] Institute of Electrical and Electronics Engineers. "IEEE Standard Multivalue Logic System for VHDL Model Interoperability (Std_logic_1164)". In: *IEEE Std 1164-1993* (1993), pp. 1–24. DOI: 10/bhz6s9.

Formalization of three-valued bit-vectors

- Abstract bit values formalized as

$$`0' := \{0\}, `1' := \{1\}, `X' := \{0, 1\} \quad (1)$$

- Abstract bit-vectors: tuples of abstract bits, IEEE 1164 notation[5]: $``XX10" = (`X', `X', `1', `0') = (\{0, 1\}, \{0, 1\}, \{1\}, \{0\})$
- Concretization function for abstract bit vectors:

$$\gamma(\hat{a}) = \{a \mid \forall i \in \{0, \ldots, N-1\} \,.\, a_i \in \hat{a}_i\}. \quad (2)$$

- Example: $\gamma(``XX10") = \{0010_2, 0110_2, 1010_2, 1110_2\}$

[5] Institute of Electrical and Electronics Engineers. "IEEE Standard Multivalue Logic System for VHDL Model Interoperability (Std_logic_1164)". In: *IEEE Std 1164-1993* (1993), pp. 1–24. DOI: 10/bhz6s9.

Forward operation problem (simplified definitions)

- *Forward operation problem*: for a given binary operator $r : \mathbb{B}^N \times \mathbb{B}^N \to \mathbb{B}^M$, and abstract inputs $\hat{a}$, $\hat{b}$, find $\hat{r}$ such that

$$\forall a \in \gamma(\hat{a}) \; . \; b \in \gamma(\hat{b}) \; . \; \exists c \in \gamma(\hat{r}) \; . \; c = r(a, b) \tag{3}$$

Forward operation problem (simplified definitions)

- *Forward operation problem*: for a given binary operator $r : \mathbb{B}^N \times \mathbb{B}^N \to \mathbb{B}^M$, and abstract inputs $\hat{a}$, $\hat{b}$, find $\hat{r}$ such that

$$\forall a \in \gamma(\hat{a}) \; . \; b \in \gamma(\hat{b}) \; . \; \exists c \in \gamma(\hat{r}) \; . \; c = r(a, b) \tag{3}$$

- *Best abstract transformer*: minimizes $|\gamma(\hat{r})|$, naïve computation in $\Theta(2^{2N})$ time, example $\hat{a} + \hat{b}$ (just with actual possibilities):

$$\hat{a} = \text{“}\mathbf{X}0\text{”}, \; \hat{b} = \text{“}11\text{”}$$
$$a = \mathbf{0}0_2, \quad b = 11_2, \quad r = \mathbf{01}1_2 = \mathbf{0}0_2 + 11_2$$
$$a = \mathbf{1}0_2, \quad b = 11_2, \quad r = \mathbf{10}1_2 = \mathbf{1}0_2 + 11_2$$
$$\hat{r}^{\text{best}} = \text{“}\mathbf{XX}1\text{”}$$

Forward operation problem (simplified definitions)

- *Forward operation problem*: for a given binary operator $r : \mathbb{B}^N \times \mathbb{B}^N \to \mathbb{B}^M$, and abstract inputs $\hat{a}$, $\hat{b}$, find $\hat{r}$ such that

$$\forall a \in \gamma(\hat{a}) \,.\, b \in \gamma(\hat{b}) \,.\, \exists c \in \gamma(\hat{r}) \,.\, c = r(a, b) \tag{3}$$

- *Best abstract transformer*: minimizes $|\gamma(\hat{r})|$, naïve computation in $\Theta(2^{2N})$ time, example $\hat{a} + \hat{b}$ (just with actual possibilities):

$$\hat{a} = \text{``}\mathbf{X}0\text{''},\ \hat{b} = \text{``}11\text{''}$$

$$a = \mathbf{0}0_2, \quad b = 11_2, \quad r = \mathbf{01}1_2 = \mathbf{0}0_2 + 11_2$$

$$a = \mathbf{1}0_2, \quad b = 11_2, \quad r = \mathbf{10}1_2 = \mathbf{1}0_2 + 11_2$$

$$\hat{r}^{\text{best}} = \text{``}\mathbf{XX}1\text{''}$$

- Back to our question: **Can the best result be obtained in polynomial time?**

Our results

Assuming every input/output bit-vector fits in the machine word of a traditional processor performing verification,

Theoretical result 1: Fast abstract addition

The best abstract transformer of abstract bit-vector **addition** is computable **in linear time**.

Theoretical result 2: Fast abstract multiplication

The best abstract transformer of abstract bit-vector **multiplication** is computable in **worst-case quadratic time**.

Experimental evaluation results

Fast algorithms can be computed above $100\frac{\mathrm{kOps}}{\mathrm{s}}$ for $N = 32$, while naïve computation is practically infeasible for $N > 8$. Memory is a non-issue, only a small fixed amount of temporary variables is needed.

Obtaining fast algorithms: pseudo-Boolean reinterpretation

- Reinterpret the concrete operation function r as a *pseudo-Boolean operation function* $h : \mathbb{B}^N \times \mathbb{B}^N \to \mathbb{N}_0$

Obtaining fast algorithms: pseudo-Boolean reinterpretation

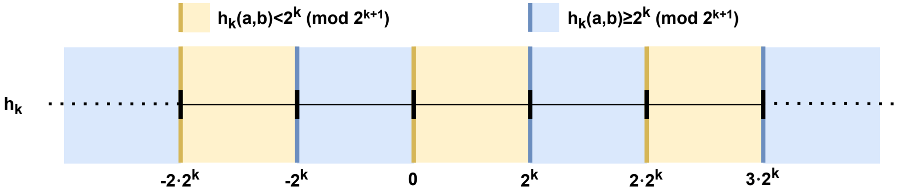
- Reinterpret the concrete operation function r as a *pseudo-Boolean operation function* $h : \mathbb{B}^N \times \mathbb{B}^N \to \mathbb{N}_0$
- Use new functions $(h_k)_{k=0}^{M-1}$, each congruent with h modulo 2^{k+1}

Obtaining fast algorithms: pseudo-Boolean reinterpretation

- Reinterpret the concrete operation function r as a *pseudo-Boolean operation function* $h : \mathbb{B}^N \times \mathbb{B}^N \to \mathbb{N}_0$
- Use new functions $(h_k)_{k=0}^{M-1}$, each congruent with h modulo 2^{k+1}
- Equivalent best abstract transformer formula:

$$\forall k \in \{0, \ldots, M-1\} \,.$$
$$(0 \in \hat{r}_k^{\text{best}} \Leftrightarrow \exists a \in \gamma(\hat{a}), b \in \gamma(\hat{b}) \,.\, (h_k(a,b) \bmod 2^{k+1}) < 2^k) \wedge \tag{4}$$
$$(1 \in \hat{r}_k^{\text{best}} \Leftrightarrow \exists a \in \gamma(\hat{a}), b \in \gamma(\hat{b}) \,.\, (h_k(a,b) \bmod 2^{k+1}) \geq 2^k)$$

- Visualisation of h_k inequalities for a single bit k:

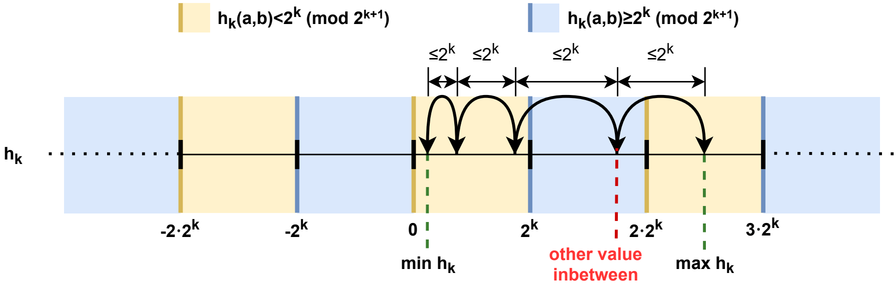


Modular extreme-finding technique

- *Step size* = absolute change of pseudo-Boolean function value when one bit is flipped

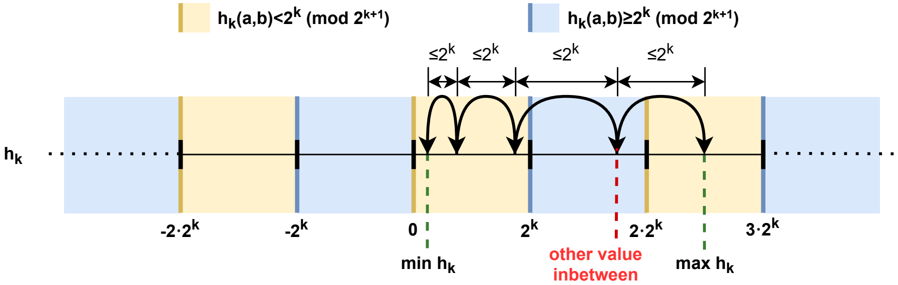
Modular extreme-finding technique

- *Step size* = absolute change of pseudo-Boolean function value when one bit is flipped
- Restrict h_k to step size at most 2^k, consider the previous modular inequalities



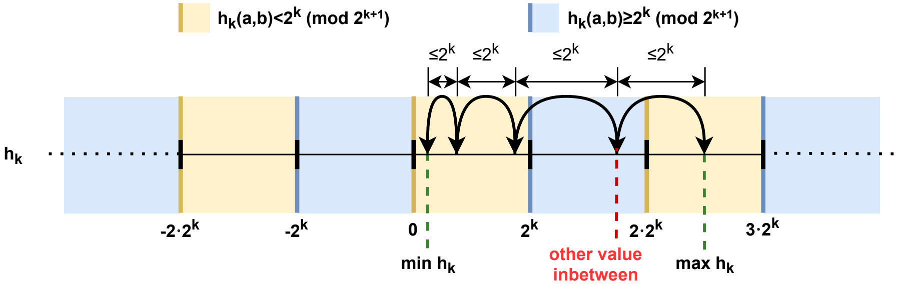
Modular extreme-finding technique

- *Step size* = absolute change of pseudo-Boolean function value when one bit is flipped
- Restrict h_k to step size at most 2^k, consider the previous modular inequalities
- Reaching minimum and maximum in the exact same area ⇒ only one holds ('0'/'1')
- Otherwise, both of them hold ('X')



Modular extreme-finding technique

- *Step size* = absolute change of pseudo-Boolean function value when one bit is flipped
- Restrict h_k to step size at most 2^k, consider the previous modular inequalities
- Reaching minimum and maximum in the exact same area ⇒ only one holds ('0'/'1')
- Otherwise, both of them hold ('X')
- **We now only need to find the extremes to get the best result!**



Fast abstract addition

- For addition,

$$h^+(a,b) = \left(\sum_{i=0}^{N-1} 2^i a_i\right) + \left(\sum_{j=0}^{N-1} 2^j b_j\right) \tag{5}$$

Fast abstract addition

- For addition,

$$h^{+}(a,b) = \left(\sum_{i=0}^{N-1} 2^i a_i\right) + \left(\sum_{j=0}^{N-1} 2^j b_j\right) \tag{5}$$

- To ensure each h_k is congruent with h and every step is at most 2^k,

$$h_k^{+}(a,b) \stackrel{\text{def}}{=} \left(\sum_{i=0}^{\mathbf{k}} 2^i a_i\right) + \left(\sum_{j=0}^{\mathbf{k}} 2^j b_j\right) \tag{6}$$

Fast abstract addition

- For addition,

$$h^{+}(a,b) = \left(\sum_{i=0}^{N-1} 2^i a_i\right) + \left(\sum_{j=0}^{N-1} 2^j b_j\right) \tag{5}$$

- To ensure each h_k is congruent with h and every step is at most 2^k,

$$h_k^{+}(a,b) \stackrel{\text{def}}{=} \left(\sum_{i=0}^{\mathbf{k}} 2^i a_i\right) + \left(\sum_{j=0}^{\mathbf{k}} 2^j b_j\right) \tag{6}$$

- Finding minimum and maximum in linear time is trivial

Fast abstract addition

- For addition,

$$h^{+}(a,b) = \left(\sum_{i=0}^{N-1} 2^i a_i\right) + \left(\sum_{j=0}^{N-1} 2^j b_j\right) \tag{5}$$

- To ensure each h_k is congruent with h and every step is at most 2^k,

$$h_k^{+}(a,b) \stackrel{\text{def}}{=} \left(\sum_{i=0}^{\mathbf{k}} 2^i a_i\right) + \left(\sum_{j=0}^{\mathbf{k}} 2^j b_j\right) \tag{6}$$

- Finding minimum and maximum in linear time is trivial
- **Directly leads to the fast algorithm**

Fast abstract addition

- For addition,

$$h^{+}(a,b) = \left(\sum_{i=0}^{N-1} 2^i a_i\right) + \left(\sum_{j=0}^{N-1} 2^j b_j\right) \tag{5}$$

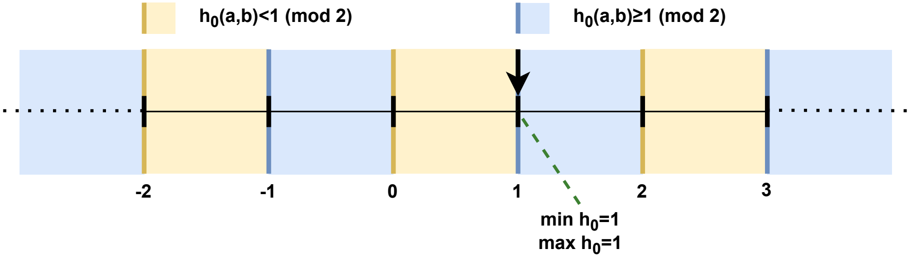
- To ensure each h_k is congruent with h and every step is at most 2^k,

$$h_k^{+}(a,b) \stackrel{\text{def}}{=} \left(\sum_{i=0}^{\mathbf{k}} 2^i a_i\right) + \left(\sum_{j=0}^{\mathbf{k}} 2^j b_j\right) \tag{6}$$

- Finding minimum and maximum in linear time is trivial
- **Directly leads to the fast algorithm**
- Similar for subtraction and summation with multiple independent operands

Fast abstract addition: example 1/2

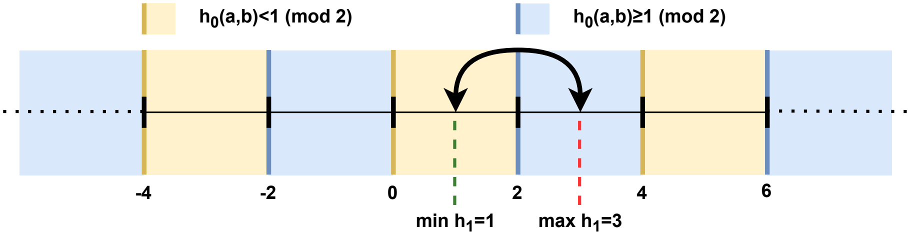
- Example "X0" + "11", k = 0:
  - $h_0^+ = \text{"0"} + \text{"1"}$
  - $\min h_0^+ = \mathbf{001}_2$
  - $\max h_0^+ = \mathbf{001}_2$
- Visualisation:



- $\lfloor \frac{\min h_0^+}{1} \rfloor = \lfloor \frac{\max h_0^+}{1} \rfloor \to \hat{r}_0^{\text{best}} = \{\lfloor \frac{\min h_0^+}{1} \rfloor \bmod 2\} = \text{'1'}$

Fast abstract addition: example 2/2

- Example "X0" + "11", k = 1:
  - $h_1^+ = \text{"X0"} + \text{"11"}$
  - $\min h_0^+ = \mathbf{001}_2$
  - $\max h_0^+ = \mathbf{011}_2$
- Visualisation:



- $\lfloor \frac{\min h_0^+}{2} \rfloor \neq \lfloor \frac{\max h_0^+}{2} \rfloor \rightarrow \hat{r}_1^{\text{best}} = \text{'X'}$

Fast abstract multiplication: first non-best approach

- First idea: performing multiplication via summation (long multiplication)
- Does not result in best abstract transformer
- Counterexample "11" · "X1":

$$
\begin{array}{lcccc}
 & (2^3) & (2^2) & (2^1) & (2^0) \\
 & & & 1 & 1 \\
\cdot & & & b_1 & 1 \\
\hline
 & (b_1) & (b_1) & b_1 & 1 \\
 & & b_1 & 1 & \\
\hline\hline
 & b_1 & \mathbf{2b_1} & 1 + b_1 & 1
\end{array}
$$

- Best result "X0X1", long multiplication produces "XXX1" due to the interaction of b_1 with itself
- $N = 8$: 15,9% results unnecessarily overapproximated

Fast abstract multiplication: finding h_k

- Multiplication pseudo-Boolean operation function:

$$h^*(a, b) = \sum_{i=0}^{N} \sum_{j=0}^{N-i} 2^{i+j} a_i b_j \tag{7}$$

Fast abstract multiplication: finding h_k

- Multiplication pseudo-Boolean operation function:

$$h^*(a,b) = \sum_{i=0}^{N} \sum_{j=0}^{N-i} 2^{i+j} a_i b_j \tag{7}$$

- Just removing summands divisible by 2^{k+1} **does not work** as step size is at most $2^{k+1} - 1$:

$$h_k(a,b) = \sum_{i=0}^{k} \sum_{j=0}^{k-i} 2^{i+j} a_i b_j \tag{8}$$

Fast abstract multiplication: finding h_k

- Multiplication pseudo-Boolean operation function:

$$h^*(a,b) = \sum_{i=0}^{N} \sum_{j=0}^{N-i} 2^{i+j} a_i b_j \tag{7}$$

- Just removing summands divisible by 2^{k+1} **does not work** as step size is at most $2^{k+1} - 1$:

$$h_k(a,b) = \sum_{i=0}^{k} \sum_{j=0}^{k-i} 2^{i+j} a_i b_j \tag{8}$$

- Flipping the sign of 2^k coefficients, the step size is at most 2^k:

$$h_k^*(a,b) \stackrel{\text{def}}{=} \left(-\sum_{i=0}^{k} 2^k a_i b_{k-i} \right) + \left(\sum_{i=0}^{k-1} \sum_{j=0}^{k-i-1} 2^{i+j} a_i b_j \right) \tag{9}$$

Fast abstract multiplication: finding extremes

- We have defined h_k^* as

$$h_k^*(a, b) = \left(-\sum_{i=0}^{k} 2^k a_i b_{k-i} \right) + \left(\sum_{i=0}^{k-1} \sum_{j=0}^{k-i-1} 2^{i+j} a_i b_j \right) \qquad (10)$$

- Now depends on values of variables forming summands $2^k a_i b_{k-i}$

Fast abstract multiplication: finding extremes

- We have defined h_k^* as

$$h_k^*(a,b) = \left(-\sum_{i=0}^{k} 2^k a_i b_{k-i} \right) + \left(\sum_{i=0}^{k-1} \sum_{j=0}^{k-i-1} 2^{i+j} a_i b_j \right) \qquad (10)$$

- Now depends on values of variables forming summands $2^k a_i b_{k-i}$
- At least two of them with both abstract bits 'X' (double-unknown k-th column pairs): **we have proven** that they imply $\hat{r}_k^{\mathrm{best}} = \text{'X'}$

Fast abstract multiplication: finding extremes

- We have defined h_k^* as

$$h_k^*(a,b) = \left(-\sum_{i=0}^{k} 2^k a_i b_{k-i} \right) + \left(\sum_{i=0}^{k-1} \sum_{j=0}^{k-i-1} 2^{i+j} a_i b_j \right) \qquad (10)$$

- Now depends on values of variables forming summands $2^k a_i b_{k-i}$
- At least two of them with both abstract bits 'X' (double-unknown k-th column pairs): **we have proven** that they imply $\hat{r}_k^{\mathrm{best}} = \text{'X'}$
- Otherwise, single-unknown k-th column pairs can be minimized/maximized

Fast abstract multiplication: finding extremes

- We have defined h_k^* as

$$h_k^*(a,b) = \left(-\sum_{i=0}^{k} 2^k a_i b_{k-i}\right) + \left(\sum_{i=0}^{k-1} \sum_{j=0}^{k-i-1} 2^{i+j} a_i b_j\right) \tag{10}$$

- Now depends on values of variables forming summands $2^k a_i b_{k-i}$
- At least two of them with both abstract bits 'X' (double-unknown k-th column pairs): **we have proven** that they imply $\hat{r}_k^{\mathrm{best}} = \text{‘X’}$
- Otherwise, single-unknown k-th column pairs can be minimized/maximized
- The one possibly remaining double-unknown k-th column pair with both abstract bits 'X' can be resolved as a special case afterwards

Fast abstract multiplication: finding extremes

- We have defined h_k^* as

$$h_k^*(a,b) = \left(-\sum_{i=0}^{k} 2^k a_i b_{k-i}\right) + \left(\sum_{i=0}^{k-1} \sum_{j=0}^{k-i-1} 2^{i+j} a_i b_j\right) \qquad (10)$$

- Now depends on values of variables forming summands $2^k a_i b_{k-i}$
- At least two of them with both abstract bits 'X' (double-unknown k-th column pairs): **we have proven** that they imply $\hat{r}_k^{\mathrm{best}} = \text{`X'}$
- Otherwise, single-unknown k-th column pairs can be minimized/maximized
- The one possibly remaining double-unknown k-th column pair with both abstract bits 'X' can be resolved as a special case afterwards
- **Best abstract transformer** with worst-case time complexity $\Theta(N^2)$
  - main problem: h_k^* cannot be computed with standard multiplication instruction

Experimental evaluation

- Our C++ implementation (conference artifact) available on figshare under CC0 licence

Experimental evaluation

- Our C++ implementation (conference artifact) available on figshare under CC0 licence
- Computationally verified equivalence of naïve and fast algorithms for $N \leq 9$

Experimental evaluation

- Our C++ implementation (conference artifact) available on figshare under CC0 licence
- Computationally verified equivalence of naïve and fast algorithms for $N \leq 9$
- Fast algorithms much faster for interesting $N \geq 8$

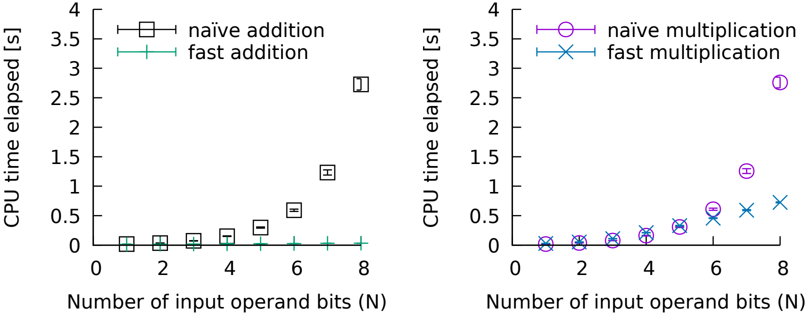


Figure 1: Measured computation times for 10^6 random abstract input combinations.

Experimental evaluation: fast algorithms

- Fast multiplication does not exhibit very noticeable quadratic behaviour for random inputs
- Fast addition extremely fast, fast multiplication still above $100\,\frac{\mathrm{kOps}}{\mathrm{s}}$ for $N = 32$

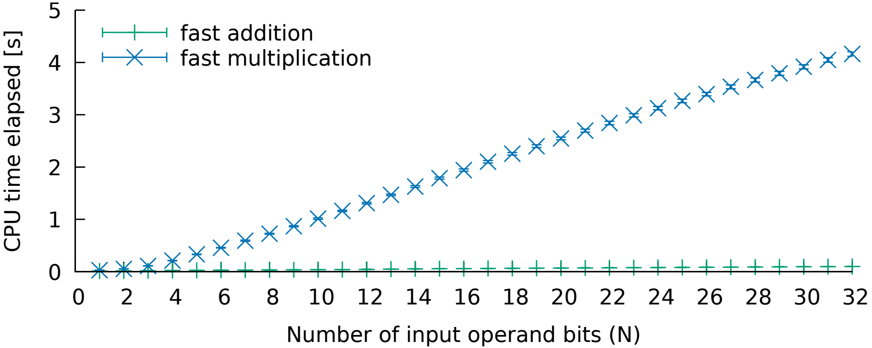


Figure 2: Measured computation time for 10^6 random abstract input combinations, fast algorithms only.

Experimental evaluation: dependence on the number of unknown bits

- Fast multiplication speed exhibits clear dependence
- Input combinations with no unknown bits are easier
- With many unknown bits, there is a high probability of multiple double-unknown k-th column pairs, implying $\hat{r}_k = \text{`X'}$

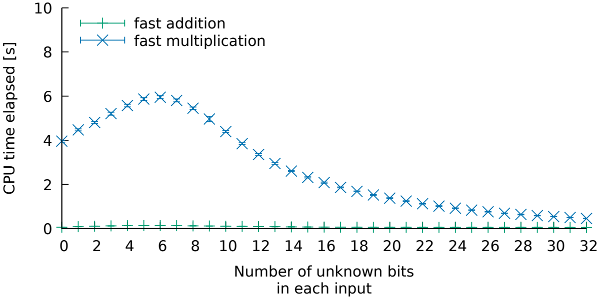


Figure 3: Measured computation times for 10^6 random abstract input combinations with fixed $N = 32$, while the number of unknown bits in each input varies.

Conclusion

- Generalized resolution of bitwise operations in three-valued logic to a **forward operation problem**

Conclusion

- Generalized resolution of bitwise operations in three-valued logic to a **forward operation problem**
- Devised a novel **modular extreme-finding technique** for best abstract transformer construction and proven its correctness

Conclusion

- Generalized resolution of bitwise operations in three-valued logic to a **forward operation problem**
- Devised a novel **modular extreme-finding technique** for best abstract transformer construction and proven its correctness
- Best abstract transformer algorithms found: $\Theta(N)$ addition, worst-case $\Theta(N^2)$ multiplication, implemented, working well

Conclusion

- Generalized resolution of bitwise operations in three-valued logic to a **forward operation problem**
- Devised a novel **modular extreme-finding technique** for best abstract transformer construction and proven its correctness
- Best abstract transformer algorithms found: $\Theta(N)$ addition, worst-case $\Theta(N^2)$ multiplication, implemented, working well
- Easily generalized to subtraction and general summation

Conclusion

- Generalized resolution of bitwise operations in three-valued logic to a **forward operation problem**
- Devised a novel **modular extreme-finding technique** for best abstract transformer construction and proven its correctness
- Best abstract transformer algorithms found: $\Theta(N)$ addition, worst-case $\Theta(N^2)$ multiplication, implemented, working well
- Easily generalized to subtraction and general summation
- Future work:
  - usage in actual model checker

Conclusion

- Generalized resolution of bitwise operations in three-valued logic to a **forward operation problem**
- Devised a novel **modular extreme-finding technique** for best abstract transformer construction and proven its correctness
- Best abstract transformer algorithms found: $\Theta(N)$ addition, worst-case $\Theta(N^2)$ multiplication, implemented, working well
- Easily generalized to subtraction and general summation
- Future work:
  - usage in actual model checker
  - division and remainder

Conclusion

- Generalized resolution of bitwise operations in three-valued logic to a **forward operation problem**
- Devised a novel **modular extreme-finding technique** for best abstract transformer construction and proven its correctness
- Best abstract transformer algorithms found: $\Theta(N)$ addition, worst-case $\Theta(N^2)$ multiplication, implemented, working well
- Easily generalized to subtraction and general summation
- Future work:
  - usage in actual model checker
  - division and remainder
  - operation fusing

Conclusion

- Generalized resolution of bitwise operations in three-valued logic to a **forward operation problem**
- Devised a novel **modular extreme-finding technique** for best abstract transformer construction and proven its correctness
- Best abstract transformer algorithms found: $\Theta(N)$ addition, worst-case $\Theta(N^2)$ multiplication, implemented, working well
- Easily generalized to subtraction and general summation
- Future work:
  - usage in actual model checker
  - division and remainder
  - operation fusing
  - **general operation problem**