

Efficient Incremental Dynamic Invariant Detection

Jeff Perkins and Michael Ernst

MIT CSAIL

Dynamic invariant detection

- Program analysis that generalizes over observed runtime values to hypothesize program properties
- The result is a set of likely invariants per program point
 - Entry to function `binary_search(int[] list, int val)`
 - `list` is sorted
 - `list` \neq null
 - `val` \in `list`
 - Exit from function `square(int a)`
 - `return = a · a`
 - Class `Stack`
 - `this.top = this.stack[this.top_stack-1]`
 - `this.stack[this.top_stack..] = null`

Uses of dynamic invariant detection

- Verifying safety properties [Vaziri 98] [Nimmer 02]
- Automatic theorem proving [Win 02]
- Identifying refactoring opportunities [Kataoka 01]
- Predicate abstraction [Dodoo 02]
- Generating test cases [Xie 03] [Gupta 03]
- Selecting and prioritizing test cases [Harder 03]
- Explaining test failures [Groce 03]
- Predicting incompatibilities in component upgrades [McCamant 03]
- Error detection [Raz 02] [Hangal 02] [Pytlik 03] [Mariani 04] [Brun 04]
- Error isolation [Xie 02] [Liblit 03]
- Choosing modalities [Lin 04]

Goals of this research

- Handle moderate to large programs
- Produce useful and expressive program properties
 - Rich set of derived variables
 - array references: $a[i]$, $a[i..]$, $a[..i]$
 - pre-state variables: at exit, $\text{orig}(x)$ stands for the value at entry
 - Rich invariant grammar
 - unary, binary, and ternary invariants
 - invariants over pointers, integers, floats, strings and arrays

Outline

- Approaches to invariant detection
 - Simple incremental algorithm
 - Simple incremental algorithm scales poorly
 - Many invariants are redundant
 - Multiple pass approach
 - Multi-pass scales poorly to large data sets
- Optimized incremental algorithm
- Complications
- Results

Simple incremental algorithm

- Hypothesize each invariant in the grammar
 - Over each set of variables
 - At each program point
- Check observed values for each variable (sample) at each invariant
 - Discard invariants that are falsified
- The remaining invariants are true over the sample data
- Examples
 - DIDUCE [Hangal 02] - checks 1 invariant over each variable
 - Carrot [Pytlik 03] - checks 2 unary and 4 binary invariants
 - Daikon version 1 [Ernst 99]

Simple incremental algorithm scales poorly

- Ternary derived variables (eg, $A[i..j]$)
 - V = the number of source program variables (at a program point)
 - $V_D = O(V^3)$
- Ternary invariants
 - $I = O(V_D^3) = O(V^9)$
- The number of possible invariants in modest test cases ranged from 460 million to 750 million

Many invariants are redundant

- Many invariants are implied by other invariants
- Examples
 - $(x = y) \wedge \text{odd}(x) \Rightarrow \text{odd}(y)$
 - $(x = 5) \wedge (y = 6) \Rightarrow (x < y)$
 - $(x < y) \Rightarrow (x \leq y)$
 - $(x \geq y)$ at class `Stack` \Rightarrow $(x \geq y)$ at method `Stack.top()`

Multiple pass approach

- Processes the input data multiple times
- Early passes check simple invariants
- Later passes check more complex invariants only if they are not redundant
 - Constants are checked first and removed
 - Equality is checked next. Only one member of an equal set need be checked in following passes
- The multi-pass approach doesn't create or check invariants implied by earlier passes (saving both time and space)
- Example: Daikon version 2

Multi-pass scales poorly to large data sets

- Even modest traces require gigabytes of space
- Possible solutions have drawbacks
 - May be too large to store in memory
 - File I/O is expensive and disks may be insufficient for larger traces
 - Running the target program multiple times is often not acceptable
 - Program has side effects
 - Program depends on its environment
 - Program uses expensive resources (such as human attention)
 - Program doesn't terminate

Outline

- Approaches to invariant detection
- Optimized incremental algorithm
 - Optimized incremental algorithm concept
 - Constants
 - Equality sets
 - Program point and variable hierarchy
 - Suppression
- Complications
- Results

Optimized incremental algorithm concept

- Same processing model as the simple incremental algorithm
- Redundant invariants are not instantiated or checked
 - Many invariants are implied by others
 - As long as the antecedents are true, the consequent need be neither instantiated nor checked
- An invariant must be created when its antecedent is falsified
 - $(x = y) \wedge \text{odd}(x) \Rightarrow \text{odd}(y)$
 - If a sample is seen where $x \neq y$, the $\text{odd}(y)$ invariant must be created
 - The new invariant must be true over all *past* samples (which are no longer available)
 - The new invariant must be checked over future samples

Constants

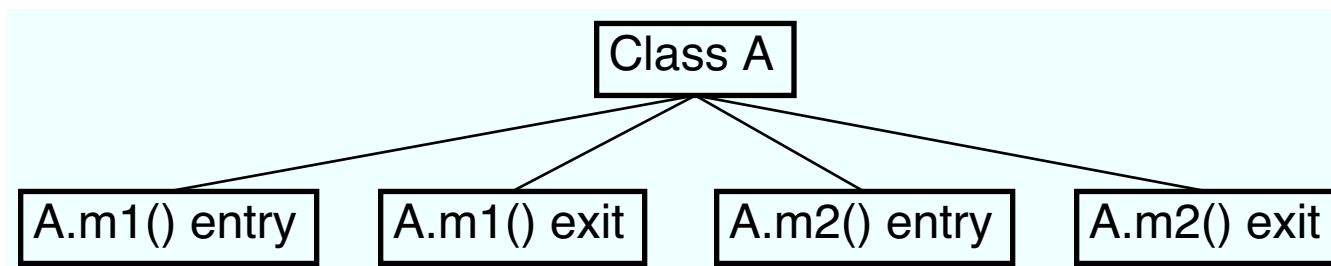
- Invariants over (only) constant variables are redundant
 - $(x = 5) \Rightarrow \text{odd}(x)$
 - $(x = 5) \wedge (y = 6) \Rightarrow x < y$
- All variables are initially constant
- Invariants are not instantiated *between* constants
- When ($\text{var} = \text{constant}$) is falsified
 - Invariants are instantiated between it and all remaining constants
 - Invariants which are not true over the constant values are discarded

Equality sets

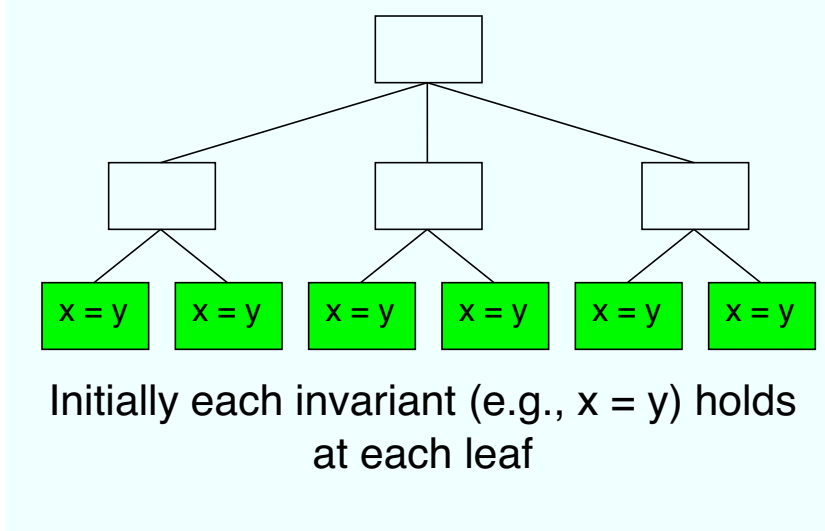
- If two or more variables are equal, any invariant true over one variable is true over all of them
 - $(x = y) \text{ and } f(x) \Rightarrow f(y)$
- Initially, all variables are placed in a single equality set
- One variable (the leader) represents the set
- Invariants are instantiated *only* between leaders
- When $(var1 = var2)$ is falsified
 - The set is split into two or more equality sets
 - Invariants over each old leader are copied to each new leader

Program point and variable hierarchy

- Relationship between program points

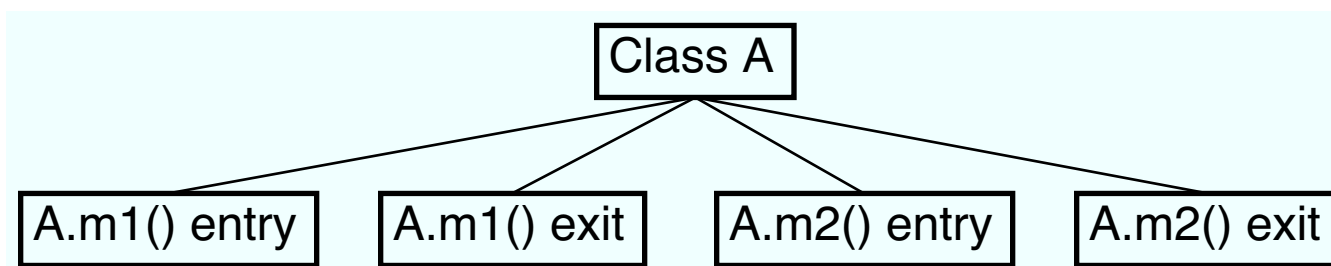


- Samples are only processed at the leaves of the hierarchy
- Invariants are created at the parent *iff* it is true at each child

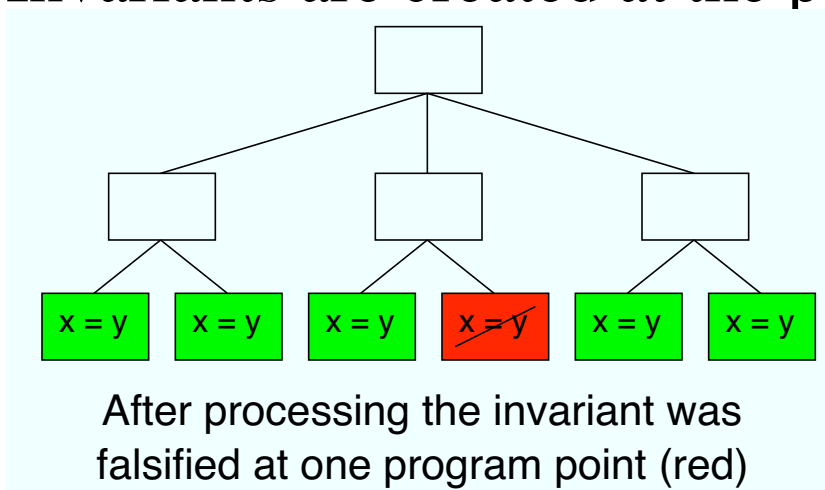


Program point and variable hierarchy

- Relationship between program points

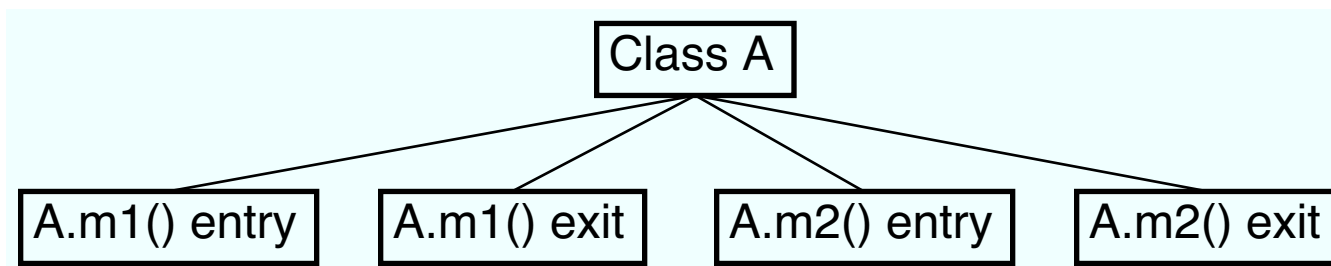


- Samples are only processed at the leaves of the hierarchy
- Invariants are created at the parent *iff* it is true at each child

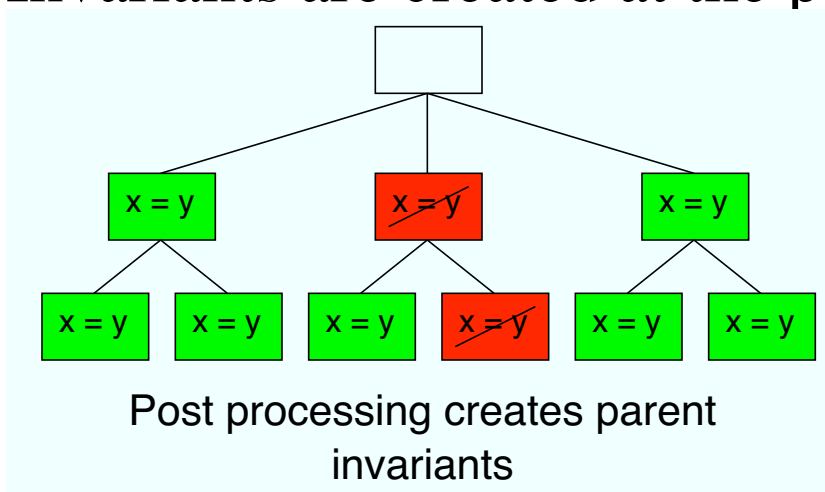


Program point and variable hierarchy

- Relationship between program points

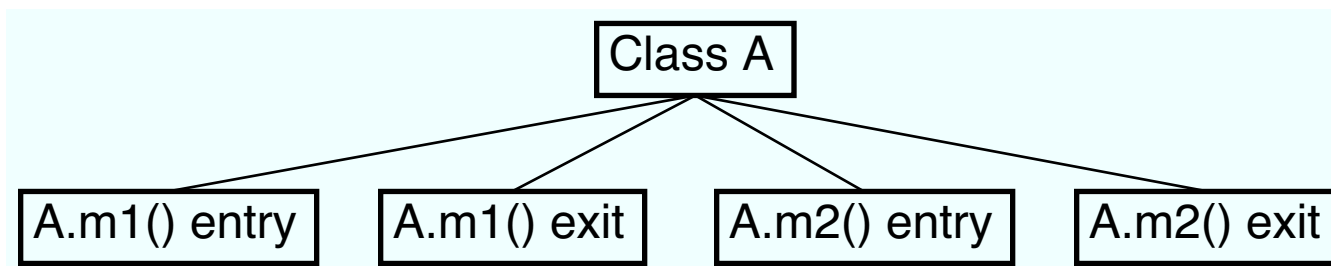


- Samples are only processed at the leaves of the hierarchy
- Invariants are created at the parent *iff* it is true at each child

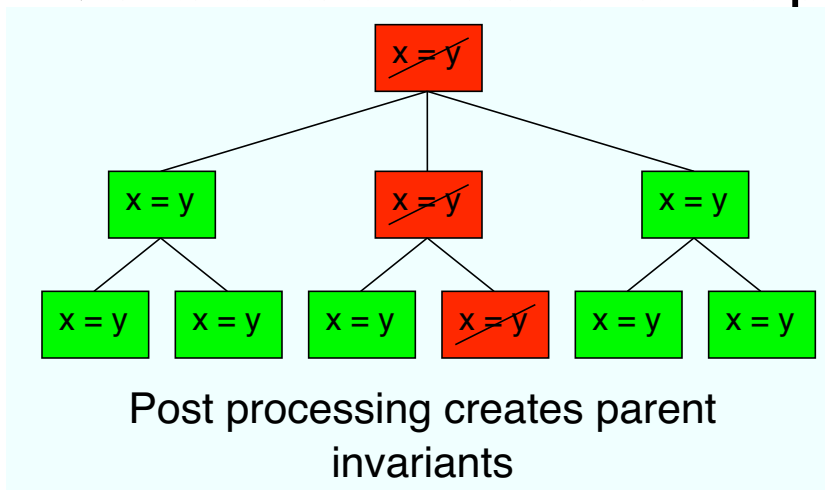


Program point and variable hierarchy

- Relationship between program points



- Samples are only processed at the leaves of the hierarchy
- Invariants are created at the parent *iff* it is true at each child



Suppression

- An invariant can be suppressed if it is logically implied by some set of other invariants. For example:
 - $(x = y) \wedge (z = 1) \Rightarrow x = y \cdot z$
 - $(x = 0) \wedge (y = 0) \Rightarrow x = y \ \& \ z$
- Other optimizations are special cases of suppression
- Goals
 - Instantiate/check only non-redundant invariants
 - Use *no* storage for a non-instantiated invariants
- When an antecedent is falsified
 - Each invariant that might be suppressed is checked
 - If a suppression held before the antecedent was falsified, but no suppression holds after, the invariant is instantiated

Outline

- Approaches to invariant detection
- Optimized incremental algorithm
- Complications
 - Missing variables
 - Optimizations interact
- Results

Missing variables

- Suppose a is null. What do we do with the invariant $a.b > x$?
- One choice is to falsify the invariant
 - The invariant thus means: $(a \neq \text{null}) \wedge (a.b > x)$
 - Problem: interesting invariants are lost
- Alternative is to retain the invariant
 - The invariant thus means: $(a \neq \text{null}) \Rightarrow (a.b > x)$
 - Problem: difficult to implement
- Optimizations must take missing into account
 - Constants must never be missing
 - Members of an equality set must have identical missing attributes
 - Suppressions can't assume transitivity
 - $(x > a.b) \wedge (a.b > y) \Rightarrow (x > y)$
 - $((a \neq \text{null}) \Rightarrow (x > a.b)) \wedge ((a \neq \text{null}) \Rightarrow (a.b > y)) \not\Rightarrow (x > y)$

Optimizations interact

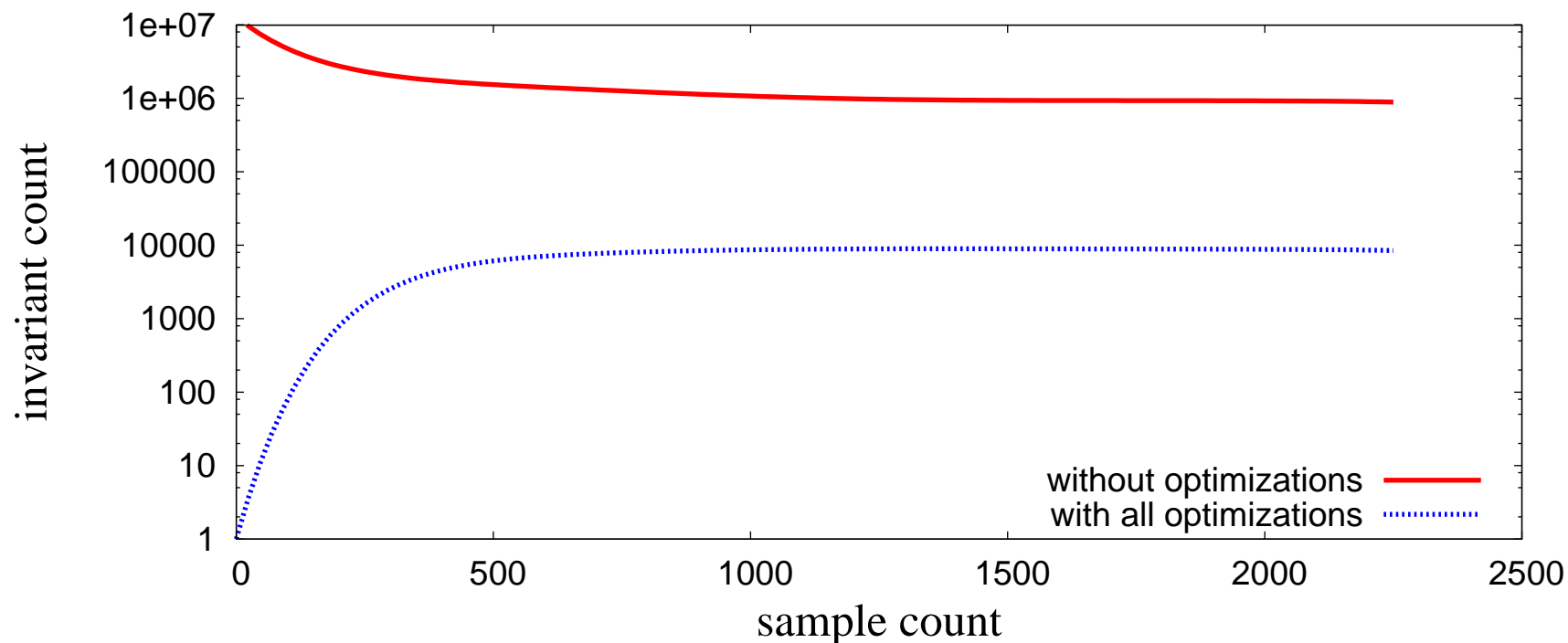
- When checking suppressions, uninstantiated invariants must be considered.
- Creating parent invariants using the program point hierarchy
 - Suppression optimizations must be undone
 - Constant and equality set information must be merged
 - Different equalities in different children require special processing
 - Uninstantiated invariants between constants must be considered

Outline

- Approaches to invariant detection
- Optimized incremental algorithm
- Complications
- Results
 - Optimizations are effective
 - Real programs can be processed
 - Performance comparison on the Daikon utilities
 - Contributions

Optimizations are effective

Candidate invariant count after each sample is processed

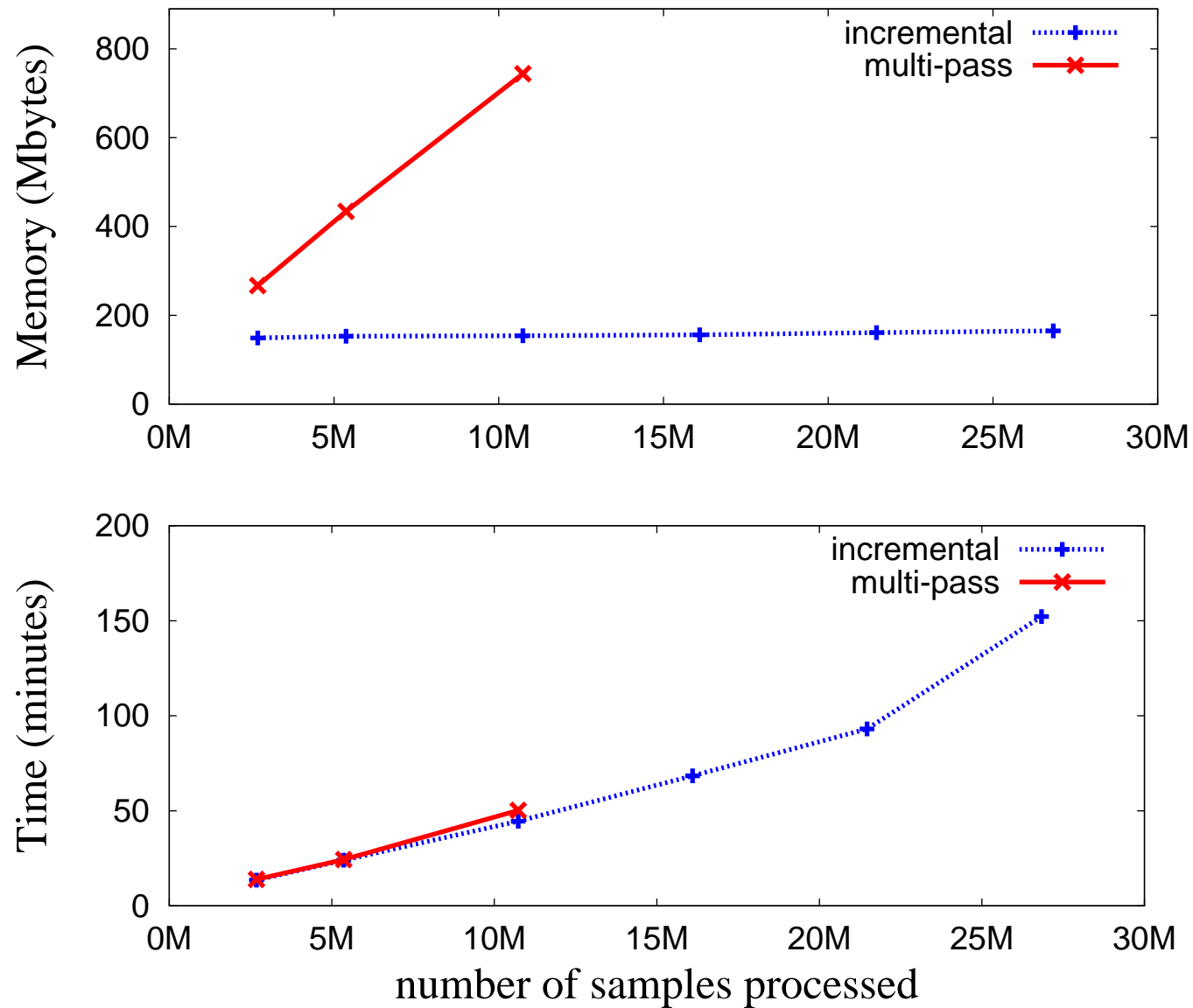


- 100 times fewer invariants with the optimizations

Real programs can be processed

- The optimized algorithm can process non-trivial programs in a reasonable amount of time and space
- The multi-pass and simple incremental approaches cannot process our experiments
- Experiments
 - Flex lexical analyzer generator
 - 391 program points averaging 275 variables each
 - 232,000 samples (9.2 Gbytes of data)
 - Processing time of 4 hours
 - Max memory use of 750 Mbytes
 - Daikon utilities
 - 1593 program points averaging 60 variables each
 - 26 million samples (11.5 Gbytes of data)
 - Processing time of 1.5 hours
 - Max memory use of 150 Mbytes

Performance comparison on the Daikon utilities



Contributions

- Effective optimizations in an incremental context
 - Redundant invariants are neither instantiated or checked
 - When antecedents are falsified, the optimization is undone and invariants that are no longer redundant are created
- Result is usable in a wide variety of contexts
 - Handles non-trivial programs
 - Supports a rich set of derived variables and invariants
 - Supports on-line operation