Efficient Set Sharing using ZBDDs

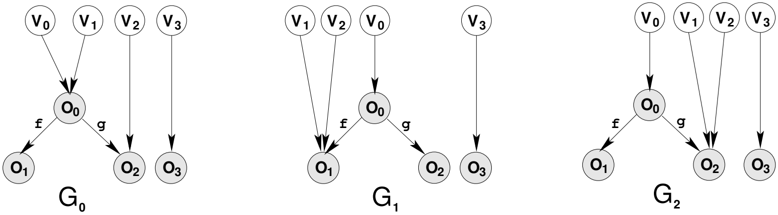
Mario Méndez-Lojo[1]
Ondřej Lhoták[2]
Manuel Hermenegildo[1,3]

[1]University of New Mexico (USA)
[2]University of Waterloo (Canada)
[3]IMDEA-Software and Technical University of Madrid (Spain)

July 31, 2008

Background: sharing property

A set of variables *share* if they reach the same memory location.



The three memory states have the same sharing representation:

$\{\{v_0, v_1, v_2\}, \{v_3\}\}$ — "v_0 reaches an object which is also reachable from v_1 and v_2, while v_3 cannot reach an object reachable from any of the other variables."

We say that v_0, v_1 and v_2 *share*, while v_3 *shares with itself*.

Background: why set sharing

- Our analysis tracks which variables *definitely do not share*.
- We use Abstract Interpretation [CC77] to ensure the correctness of this information at any program point.
- One of the uses of sharing information is for *parallelization*:
  - Assume that, in the example of the previous slide, the analysis is able to infer that the set sharing at runtime is indeed $\{\{v_0, v_1, v_2\}, \{v_3\}\}$.
  - Assume two invocations $\mathtt{m}(v_0, v_1, v_2)$ and $\mathtt{n}(v_3)$, just after that state.
  - These two method calls can be safely parallelized since they are independent: execution of $\mathtt{m}(v_0, v_1, v_2)$ cannot affect that of $\mathtt{n}(v_3)$ and they can proceed in parallel without interference.

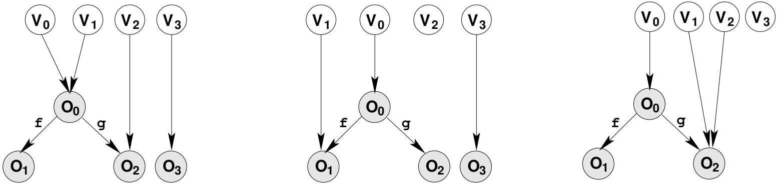
(This is of course a *safe approximation* of independence.)

Background: tracking sharing in a program

We use a set of sets of variables to approximate all the possible sharing sets (program states) that occur at a given program point:

$$SH_p = \{\{v_0, v_1\}, \{v_0, v_1, v_2\}, \{v_3\}\}$$

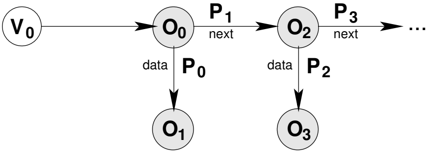
"In the set of memory states approximated by SH at program point p, v_0 may share with v_1 and v_2, or just with v_1; v_3 may point to a non-null location."



Analysis ensures that v_3 definitely does not share with v_0, or v_1, or v_2.

Background: store-aware set sharing

Sharing can be combined with *structural* information. In the previous slides we described sharing in terms of local variables, but set sharing can talk about *any* pointer. For example, this linked list:



Can be abstracted as

$$(\underbrace{\{v_0 = (\text{data}{:}p_0, \text{next}{:}p_1)\}}_{\text{shape}}, \underbrace{\{\{v_0, p_0\}, \{v_o, p_1\}\}}_{\text{set sharing}})$$

A statement like $v_1 = v_0$`.data` will result in a final abstract state:

$$(\{v_0 = (\text{data}{:}p_0, \text{next}{:}p_1)\}, \{\{v_0, v_1, p_0\}, \{v_o, p_1\}\})$$

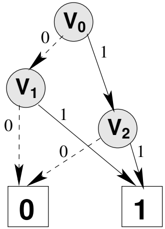
Motivation

- Our initial work [MLH08] showed that with set sharing we can achieve more *precise* results than with a related analysis [SS05] (pair sharing), for a set of small benchmarks.
- But the original implementation of set sharing did not scale:
  - Based on lists of lists.
  - We will show that even a `BitSet` list will not work.
- The main problem is the combinatorial nature of the domain.
  - Some abstract operations are exponential in both memory and time.
- Initial idea: use Binary Decision Diagrams (BDDs) –but they are not designed to (naturally) represent sets of sets.

Zero-supressed BDDs

ZBDDs [iM93] are a data structure similar to BDDs, but designed to encode sets of combinations.

- A ZBDD is a rooted directed acyclic graph (DAG) of non-terminal and terminal ($\boxed{0}$,$\boxed{1}$) nodes.
- Each path through the ZBDD that ends at the $\boxed{1}$ node defines a set of variables (those that the path leaves along a 1-edge).
- ZBDDs work particularly well when representing *sparse* sets.



Universe of variables$=\{v_0, v_1, v_2, v_3\}$

ZBDD represents $v_0\overline{v}_1 v_2\overline{v}_3 + \overline{v}_0 v_1\overline{v}_2\overline{v}_3$
$= \{\{v_0, v_2\}, \{v_1\}\}$

Set sharing + ZBDDs

- Idea: replace the naive implementation by a ZBDD-based version, which is expected to *at least* use less memory.
- Efficient algorithms exist for common operations on the set of sets encoded by a ZBDD: union, difference, intersection...

Set	ZBDD	example
$SH_1 \cup SH_2$	$SH_1 + SH_2$	$\{\{v_0, v_1\}\} + \{\{v_0\}, \{v_2\}\} = \{\{v_0\}, \{v_0, v_1\}, \{v_2\}\}$
$SH_1 \uplus SH_2$	$SH_1 * SH_2$	$\{\{v_0, v_1\}\} * \{\{v_0\}, \{v_2\}\} = \{\{v_0, v_1\}, \{v_0, v_1, v_2\}\}$
SH_v	$SH // v$	$\{\{v_0, v_1\}\} // \{\{v_0\}\} = \{\{v_0\}\}$
SH_{-v}	$SH \% v$	$\{\{v_0, v_1\}\} \% \{\{v_0\}\} = \{\{v_1\}\}$

- We do not need to alter the existing set sharing semantics!
- Example: approximatting the effects of a variable load.

$$\begin{aligned} \mathcal{SE}^{l}_{\pi}[\![v]\!](SH) &= (\{\{res\}\} \uplus SH_{\mathbf{v}}) \cup SH_{-v} && \text{(sets } \textbf{ops}\text{)} \\ &= res * (SH // v) + SH \% v && \text{(primitive ZBDD } \textbf{ops}\text{)} \end{aligned}$$

Transfer functions in terms of ZBDD operations

- In practice, we replace certain combinations of *primitive* ($+$,$*$, $\%$,$\uplus$) operations by a custom, equivalent ZBDD algorithm.
- Example: approximatting the effects of a variable load.

$$\begin{array}{rcll} \mathcal{SE}^{I}_{\pi}[\![v]\!](SH) & = & (\{\{res\}\}\uplus SH_{\mathbf{v}})\cup SH_{-v} & \text{(sets } \textbf{ops}\text{)} \\ & = & res*(SH//v) + SH\,\%v & \text{(primitive ZBDD } \textbf{ops}\text{)} \\ & = & \textbf{setResEqTo}(SH, v) & \text{(dedicated ZBDD } \textbf{op}\text{)} \end{array}$$

The dedicated algorithm **setResEqTo** runs faster.

- Example: approximatting the effects of a field load.

$$\begin{array}{rcll} \mathcal{SE}^{I}_{\pi}[\![v\,.\,\mathtt{f}]\!](SH) & = & SH\cup(\{\{v, res\}\}\uplus\bigcup_{S\in SH_v}\mathcal{P}(S|_{-v})) & \text{(sets)} \\ & = & SH+v*res*\textbf{powUnion}(SH/v) & \text{(ZBDD)} \end{array}$$

A dedicated algorithm: powerset computation

Approximatting the effects of a field load (v.f) and store (v.f=*expr*) in state *SH* implies computing:

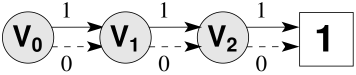
$$\textbf{powUnion}(SH) = \bigcup_{S \in SH} \mathcal{P}(S)$$

The native implementation of **powUnion** is a key factor in the overall performance of the analysis.

- Standard computation of the powerset presents a combinatorial explosion in both memory and running time.

$$\mathcal{P}(\{v_0, v_1, v_2\}) =$$
$$\{\{\}, \{v_0\}, \{v_1\}, \{v_2\}, \{v_0, v_1\}, \{v_0, v_2\}, \{v_1, v_2\}, \{v_0, v_1, v_2\}\}$$

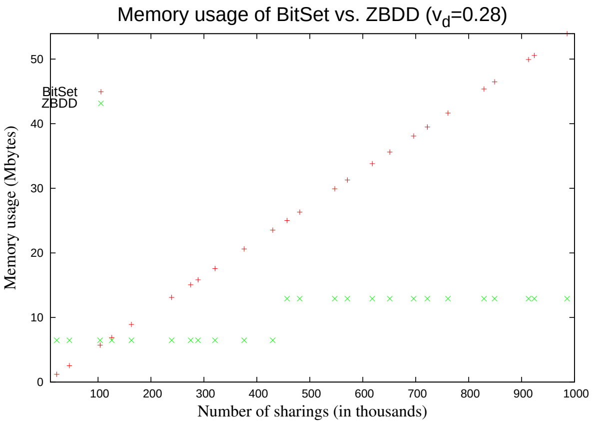
- The same powerset is compactly represented by a ZBDD.



Experimental results - memory usage

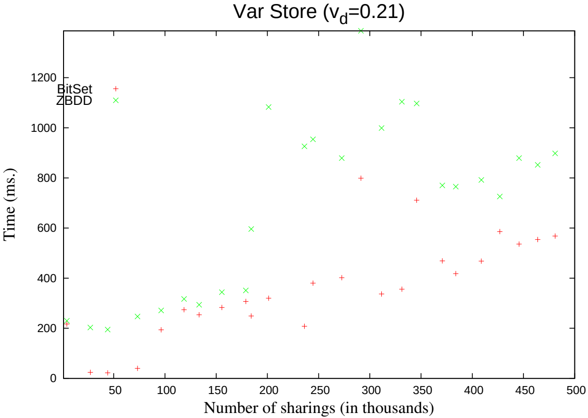
We compare a `BitSet` and a `ZBDD`-based implementation.

- The `BitSet` representation uses 50 bytes per set ($N \leq 32$).
- The ZBDD version behaves better for large set sharings (5x improvement).



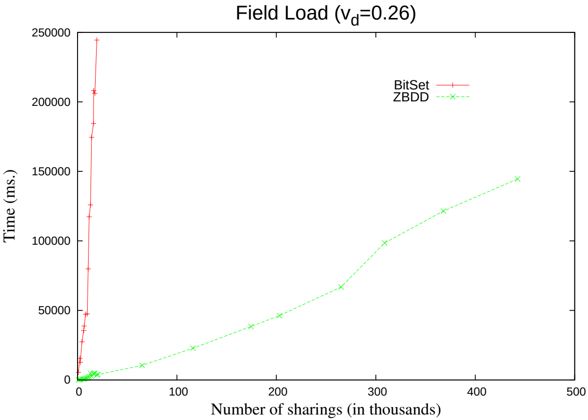
Experimental results - performance (I)

- Some `BitSet`-based operations are faster: for example, computing the effects of a variable store.
- Variable load semantics are calculated in similar times.



Experimental results - performance (II)

Powerset calculations set a major performance difference when computing the effects of a field load or store.



Conclusions and future work

- ZBDDs are adequate for encoding large sets of sets.
- Any analysis based on the set of sets representation can probably benefit from ZBDDs.
- We focused on set sharing:
  - Memory usage is improved because of the compact ZBDD encoding.
  - Better performance is achieved through efficient powerset computations.
- We are currently reimplementing the full (Java) analysis so it is ZBDD-based.
  - It will allow us to evaluate the gain in performance in the framework.
  - The expected gain in scalability will allow assessing the actual impact of the analysis in a client application (parallelization).

[CC77] P. Cousot and R. Cousot.
Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints.
In *Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[iM93] Shin ichi Minato.
Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems.
In *DAC*, pages 272–277, 1993.

[MLH08] M. Méndez-Lojo and M. Hermenegildo.
Precise Set Sharing Analysis for Java-style Programs.
In *9th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'08)*, number 4905 in LNCS, pages 172–187. Springer-Verlag, January 2008.

[SS05] S. Secci and F. Spoto.
Pair-sharing analysis of object-oriented programs.
In *Static Analysis Symposium (SAS)*, pages 320–335, 2005.