Compact Difference Bound Matrices

Aziem Chawdhary and Andy King

University of Kent, Canterbury, CT2 7NF, UK

Abstract. The Octagon domain, which tracks a restricted class of two variable inequality, is the abstract domain of choice for many applications because its domain operations are either quadratic or cubic in the number of program variables. Octagon constraints are classically represented using a Difference Bound Matrix (DBM), where the entries in the DBM store bounds c for inequalities of the form $x_i - x_j \leqslant c$, $x_i + x_j \leqslant c$ or $-x_i - x_j \leqslant c$. The size of such a DBM is quadratic in the number of variables, giving a representation which can be excessively large for number systems such as rationals. This paper proposes a compact representation for DBMs, in which repeated numbers are factored out of the DBM. The paper explains how the entries of a DBM are distributed, and how this distribution can be exploited to save space and significantly speed-up long-running analyses. Moreover, unlike sparse representations, the domain operations retain their conceptually simplicity and ease of implementation whilst reducing memory usage.

1 Introduction

The Octagon domain [18] is a widely deployed [6] abstract domain, whose popularity stems from the polynomial complexity of its domain operations [2,9,18] and ease of implementation [13]. Systems of octagon constraints are conventionally represented [18] using difference bound matrices (DBMs). DBMs were originally devised for modelling (time) [10,16] differences where each difference constraint $x_i - x_j \leqslant c$ bounds a difference $x_i - x_j$ with a constant c. For a set of program variables $\{x_0, \ldots, x_{n-1}\}$, an inequality $x_i - x_j \leqslant c$ can be represented by storing c at the i, j entry of an $n \times n$ matrix, which is the DBM. The absence of an upper bound on $x_i - x_j$ is indicated by an entry of ∞. A DBM thus gives a natural representation for a system of n^2 difference constraints. Moreover, a Floyd-Warshall style, $O(n^3)$, all-pairs shortest path algorithm can be applied to check satisfiability and derive a canonical representation.

By working over an augmented set of variables $\{x'_0, \ldots, x'_{2n-1}\}$ and defining $x'_{2i} = x_i$ and $x'_{2i+1} = -x_i$, algorithms for manipulating difference constraints can be lifted to octagonal constraints [18]. Moreover, because of redundancy induced by the encoding, it is not necessary to deploy a DBM of dimension $2n \times 2n$, but instead the DBM can be packed into an array of size $2n(n+1)$. Nevertheless, space consumption is a problem for large n.

Space consumption is not just a space problem: memory needs to be allocated, initialised and managed, all of which take time. Running Callgrind [25] on an off-the-shelf abstract interpreter (EVA [8]), equipped with the de-facto implementation of Octagons (Apron [13]) on AES-128 code (taes of table 1) revealed that 36% of all the function calls emanated from qmpq_init which merely allocates memory and initialises the state of a rational number. When working over rationals, these indirect costs dampen or mask algorithmic improvements obtained by refactoring [2] and reformulating [9] domain operations.

One solution is to abandon rationals for doubles [14,23], which is less than satisfactory for the purposes of verification. Another recent trend is adopt a sparse representation [11,14], sacrificing the simplicity and regularity of DBMs which, among other things, makes DBMs amenable to parallelisation [3]. Instead, this paper proposes compact DBMs (CoDBMs) which exploit a previously overlooked property: the number of different DBM entries is typically small. This allows common matrix entries to be shared and reused across all CoDBMs, reducing memory pressure and factoring out repeated initialisation. To summarise, this paper makes the following contribution to the representation of DBMs and octagonal analysis in particular:

– It reports the relative frequency of read and write to DBMs, as well the total number of distinct numbers that arise during the lifetime of octagonal analyses using DBMs. These statistics justify the CoDBM construction.
– It proposes CoDBMs for improving the memory consumption of DBMs, which does not compromise the conceptual simplicity of DBMs or their regular structure, important to algorithmic efficiency.
– It provides experimental evidence which shows that the extra overheads induced by reading and writing to a CoDBM are repaid, often significantly, by the savings in memory allocation and initialisation.
– It analyses the performance gains in terms of the number of memory references and percentage of cache misses, explaining why the auxiliary datastructure used for reading a CoDBM has good locality of reference.

2 The Octagon Domain and its Representation

An octagonal constraint [2,17,18] is a two variable inequality of the syntactic form $x_i - x_j \leqslant c$, $x_i + x_j \leqslant c$ or $-x_i - x_j \leqslant c$ where c is a constant, and x_i and x_j are drawn from a finite set of program variables $\{x_0, \ldots, x_{n-1}\}$. This class includes unary inequalities $x_i + x_i \leqslant c$ and $-x_i - x_i \leqslant c$ which express interval constraints. An octagon is a set of points satisfying a system of octagonal constraints. The octagon domain over $\{x_0, \ldots, x_{n-1}\}$ is the set of all octagons defined over $\{x_0, \ldots, x_{n-1}\}$.

Implementations of the octagon domain reuse machinery developed for solving difference constraints of the form $x_i - x_j \leqslant c$. An octagonal constraint over $\{x_0, \ldots, x_{n-1}\}$ can be translated [18] to a difference constraint over an augmented set of variables $\{x'_0, \ldots, x'_{2n-1}\}$, which are interpreted by $x'_{2i} = x_i$ and

$$\begin{aligned} x_0 &\leqslant 3 \\ x_1 &\leqslant 2 \\ x_0 + x_1 &\leqslant 6 \\ -x_0 - x_1 &\leqslant 5 \\ -x_0 &\leqslant 3 \end{aligned} \qquad \begin{aligned} x'_0 - x'_1 &\leqslant 6 \\ x'_2 - x'_3 &\leqslant 4 \\ x'_0 - x'_3 &\leqslant 6 \\ x'_2 - x'_1 &\leqslant 6 \\ x'_1 - x'_2 &\leqslant 5 \\ x'_3 - x'_0 &\leqslant 5 \\ x'_1 - x'_0 &\leqslant 6 \end{aligned} \qquad \begin{array}{c|cccc} & x'_0 & x'_1 & x'_2 & x'_3 \\ \hline x'_0 & \infty & 6 & \infty & 6 \\ x'_1 & 6 & \infty & 5 & \infty \\ x'_2 & \infty & 6 & \infty & 4 \\ x'_3 & 5 & \infty & \infty & \infty \end{array}$$

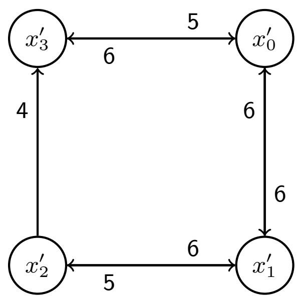

Fig. 1: Example of an octagonal system and its DBM representation

$x'_{2i+1} = -x_i$. The translation proceeds as follows:

$$\begin{aligned} x_i - x_j \leqslant c &\rightsquigarrow \quad x'_{2i} - x'_{2j} \leqslant c \;\wedge\; x'_{2j+1} - x'_{2i+1} \leqslant c \\ x_i + x_j \leqslant c &\rightsquigarrow x'_{2i} - x'_{2j+1} \leqslant c \;\wedge\; \quad x'_{2j} - x'_{2i+1} \leqslant c \\ -x_i - x_j \leqslant c &\rightsquigarrow x'_{2i+1} - x'_{2j} \leqslant c \;\wedge\; \quad x'_{2j+1} - x'_{2i} \leqslant c \\ x_i \leqslant c &\rightsquigarrow x'_{2i} - x'_{2i+1} \leqslant 2c \\ -x_i \leqslant c &\rightsquigarrow x'_{2i+1} - x'_{2i} \leqslant 2c \end{aligned}$$

A difference bound matrix (DBM) [10,16], which is a square matrix of dimension $n \times n$, is commonly used to represent a systems of n^2 (syntactically irredundant [15]) difference constraints over n variables. The entry $\mathbf{m}_{i,j}$ that represents the constant c of the inequality $x_i - x_j \leqslant c$ where $i, j \in \{0, \ldots, n-1\}$. Since an octagonal constraint system over n variables translates to a difference constraint system over $2n$ variables, a DBM representing an octagon has dimension $2n \times 2n$.

Example 1. Figure 1 serves as an example of how an octagon translates to a system of differences. The entries of the DBM correspond to the constants in the difference constraints. Note how differences which are (syntactically) absent from the system lead to entries which take a symbolic value of ∞. Observe too how that DBM defines an adjacency matrix for the illustrated graph where the weight of a directed edge abuts its arrow.

The interpretation of a DBM representing an octagon is different to a DBM representing difference constraints. Consequently there are two concretisations for DBMs: one for interpreting differences and another for interpreting octagons, although the latter is defined in terms of the former.

Definition 1. *Concretisation for rational* ($\mathbb{Q}^n$) *solutions:*

$$\begin{aligned} \gamma_{\mathit{diff}}(\mathbf{m}) &= \{\langle v_0, \ldots, v_{n-1}\rangle \in \mathbb{Q}^n \mid \forall i, j. v_i - v_j \leqslant \mathbf{m}_{i,j}\} \\ \gamma_{\mathit{oct}}(\mathbf{m}) &= \{\langle v_0, \ldots, v_{n-1}\rangle \in \mathbb{Q}^n \mid \langle v_0, -v_0, \ldots, v_{n-1}, -v_{n-1}\rangle \in \gamma_{\mathit{diff}}(\mathbf{m})\} \end{aligned}$$

where the concretisation for integer ($\mathbb{Z}^n$) *solutions can be defined analogously.*

$$\begin{array}{c|cccc|} & x'_0 & x'_1 & x'_2 & x'_3 \\ \hline x'_0 & 11 & 6 & 11 & 6 \\ x'_1 & 6 & 11 & 5 & 9 \\ x'_2 & 9 & 6 & 11 & 4 \\ x'_3 & 5 & 11 & 16 & 11 \end{array} \qquad \begin{array}{c|cccc|} & x'_0 & x'_1 & x'_2 & x'_3 \\ \hline x'_0 & 0 & 6 & 11 & 6 \\ x'_1 & 6 & 0 & 5 & 9 \\ x'_2 & 9 & 6 & 0 & 4 \\ x'_3 & 5 & 11 & 16 & 0 \end{array} \qquad \begin{array}{c|cccc|} & x'_0 & x'_1 & x'_2 & x'_3 \\ \hline x'_0 & 0 & 6 & 11 & 5 \\ x'_1 & 6 & 0 & 5 & 5 \\ x'_2 & 5 & 5 & 0 & 4 \\ x'_3 & 5 & 11 & 16 & 0 \end{array}$$

Fig. 2: DBM after shortest path, closed DBM and strongly closed DBM

Example 2. Since octagonal inequalities are modelled as two related differences, the DBM of figure 1 contains duplicated entries, for instance, $\mathbf{m}_{1,2} = \mathbf{m}_{3,0}$.

Operations on a DBM representing an octagon must maintain equality between the two entries that share the same constant of an octagonal inequality. This requirement leads to the notion of coherence:

Definition 2 (Coherence). *A DBM* $\mathbf{m}$ *is coherent iff* $\forall i.j.\mathbf{m}_{i,j} = \mathbf{m}_{\bar{j},\bar{\imath}}$ *where* $\bar{\imath} = i + 1$ *if* i *is even and* $i - 1$ *otherwise.*

Example 3. Observe from figure 1 that $\mathbf{m}_{0,3} = 6 = \mathbf{m}_{2,1} = \mathbf{m}_{\bar{3},\bar{0}}$. Coherence holds in a degenerate way for unary inequalities, note $\mathbf{m}_{2,3} = 4 = \mathbf{m}_{2,3} = \mathbf{m}_{\bar{3},\bar{2}}$.

Care should be taken to preserve coherence when manipulating DBMs, either by carefully designing algorithms or by using a data structure that enforces coherence [17, Section 4.5], as realised in the Apron library [13]. Finally to check if a DBM represents a satisfiable octagonal system, we have the following notion:

Definition 3 (Consistency). *A DBM* $\mathbf{m}$ *is consistent iff* $\forall i.\mathbf{m}_{i,i} \geqslant 0$.

2.1 Definitions of Closure

Closure properties define canonical representations of DBMs, and can decide satisfiability and support operations such as join and projection. Bellman [5] showed that the satisfiability of a difference system can be decided using shortest path algorithms on a graph representing the differences. If the graph contains a negative cycle (a cycle whose edge weights sum to a negative value) then the difference system is unsatisfiable. The same applies for DBMs representing octagons. Closure propagates all the implicit (entailed) constraints in a system, leaving each entry in the DBM with the sharpest possible constraint entailed between the variables. Closure is formally defined below:

Definition 4 (Closure). *A DBM* $\mathbf{m}$ *is closed iff*

- $\forall i.\mathbf{m}_{i,i} = 0$
- $\forall i, j, k.\mathbf{m}_{i,j} \leqslant \mathbf{m}_{i,k} + \mathbf{m}_{k,j}$

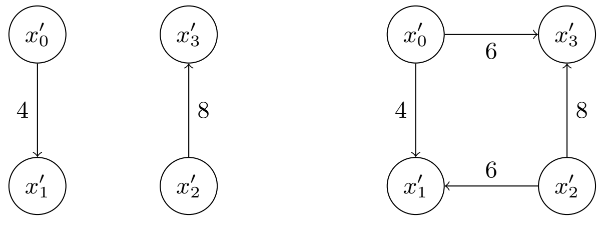

Fig. 3: Two representations of the same octagon constraints $x_0 \leqslant 2$, $x_1 \leqslant 4$

Example 4. The DBM of Figure 1 is not closed. By running an all-pairs shortest path algorithm the left DBM of figure 2 is obtained. Shortest path algorithms derive all constraints implied by the original system. Notice the diagonal has non-negative elements implying that the constraint system is satisfiable. Once satisfiability has been established, the diagonal values are set to zero to satisfy the requirements of closure, giving the middle (closed) DBM.

Closure by itself is not enough to provide a canonical form for DBMs representing octagons. A stronger notion is required called strong closure [17,18]:

Definition 5 (Strong closure). *A DBM* $\mathbf{m}$ *is strongly closed iff*

- $\mathbf{m}$ *is closed*
- $\forall i, j . \mathbf{m}_{i,j} \leqslant \mathbf{m}_{i,\bar{\imath}}/2 + \mathbf{m}_{\bar{\jmath},j}/2$

The strong closure of DBM m can be computed by propagating the following the property: if $x'_j - x'_{\bar{\jmath}} \leqslant c_1$ and $x'_{\bar{\imath}} - x'_i \leqslant c_2$ both hold then $x'_j - x'_i \leqslant (c_1 + c_2)/2$ also holds. This sharpens the bound on the difference $x'_j - x'_i$ using the two unary constraints encoded by $x'_j - x'_{\bar{\jmath}} \leqslant c_1$ and $x'_{\bar{\imath}} - x'_i \leqslant c_1$, namely, $2x'_j \leqslant c_1$ and $-2x'_i \leqslant c_2$. Note that this constraint propagation is not guaranteed to occur with a shortest path algorithm since there is not necessarily a path from a $\mathbf{m}_{i,\bar{\imath}}$ and $\mathbf{m}_{\bar{\jmath},j}$. An example in figure 3 illustrates such a situation: the two graphs represent the same octagon, but a shortest path algorithm will not propagate constraints on the left graph; hence strengthening is needed to bring the two graphs to the same normal form. Strong closure yields a canonical representation: there is a unique strongly closed DBM for any (non-empty) octagon [18]. Thus any semantically equivalent octagonal constraint systems are represented by the same strongly closed DBM. Strengthening is the act of computing strong closure.

Example 5. The right DBM of figure 2 gives the strong closure of the middle DBM of the same figure.

Thus the overall algorithm for computing the strong closure of an octagonal DBM is to first run a closure algorithm, check for consistency by searching for a negative entry in the diagonal, and then apply strengthening [9,18]. These closure algorithms are not detailed for reasons of brevity; they are a (long) study in their own right [2,9,18], and the CoDBM representation, and its relationship to a DBM, can be followed without detailed understanding of these algorithms.

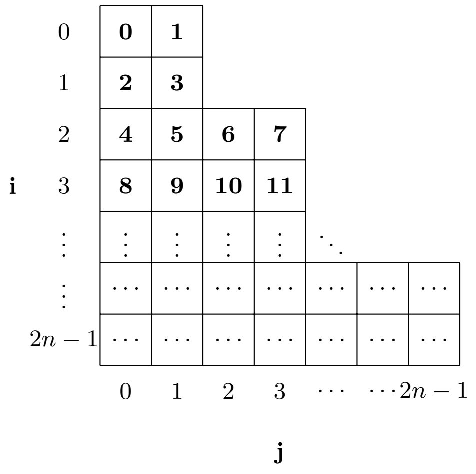

Fig. 4: Half-Matrix Representation of a DBM

2.2 Apron Library

The Apron library is the most widely used Octagon domain implementation [13]. The Apron library is implemented in C, with bindings for C++, Java and OCaml. The library implements the box, polyhedra and octagon abstract domains. Various number systems are supported by the Apron library, such as single-precision floats and GNU multiple-precision (GMP) rationals. Numbers are represented by a type `bound_t`, which depending on compile time options will select a specific header file containing concrete implementations of operations involving numbers extended to the symbolic values of $-\infty$ and $+\infty$. Every `bound_t` object has to be initialised via a call to `bound_init`, which in the case of GMP rationals will call a `malloc` function and heap allocate space for the rational number.

DBMs are stored in memory by taking advantage of the half-matrix nature of octagonal DBMs which follows by coherence. A (linear) array of `bound_t` objects is then used to represent the half-matrix, as shown in figure 4. If $i \geqslant j$ or $i = \bar{\jmath}$ then the entry at (i, j) in the DBM is stored at index $j + \lfloor i^2/2 \rfloor$ in the array. Otherwise (i, j) is stored at the index location reserved for entry $(\bar{\jmath}, \bar{\imath})$. A DBM of size n requires an array of size $2n(n+1)$ which gives a significant space reduction.

However, DBMs are still not compact: if a rational occurs repeatedly in a DBM then each occurrence of that number is heap allocated separately.

3 Compact DBMs

The rationale for compact DBMs (CoDBMs) is to redistribute the cost of memory allocation and initialisation, and do so in a way that is sensitive to the

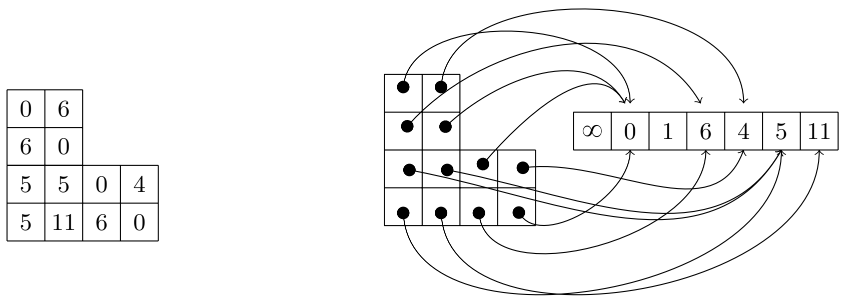

Fig. 5: Example illustrating the difference between DBMs and CoDBMs

relative frequency of DBM reads to DBM writes, whilst improving locality of reference.

CoDBMs are difference bound matrices where the entries are identifiers, rather than numeric values, and the identifiers are interpreted and maintained with the aid of two arrays, namely values and sorted. Conceptually, CoDBMs differ from DBMs by storing numbers in a common shared array, and elements in the DBM refer to these rather than storing the numeric bound itself, as shown in figure 5.

The array values has size elements and has elements of type $\mathbb{Q} \cup \{\infty\}$. The values array is used to map an identifier, which is an integer, to its value. It is used when reading an entry in a CoDBM. Dually, the array sorted is used when writing an entry to a CoDBM. This array is indexed from 0, contains size records, where each record has value and id fields of type $\mathbb{Q} \cup \{\infty\}$ and $\mathbb{N}$ respectively. This array records of all CoDBM entry values which have been seen thus far, including ∞, with their identifiers. The records are arranged in strictly ascending order, using the value field as the key. The ordering provides an efficient way of searching to determine whether a value has been encountered previously. If found, the id field gives the corresponding identifier.

An identifier of 0 is preassigned for the symbolic value ∞. This is achieved by initialising the values and sorted arrays so that $\mathsf{values}[0] = \infty$, $\mathsf{sorted}[0].\mathsf{value} = \infty$ and $\mathsf{sorted}[0].\mathsf{id} = 0$. Finally, identifiers are allocated in increasing order, and next is used to record the next available unused identifier; it is initialised to 1, since 0 is used to identify ∞.

3.1 CoDBM data-structure invariants

The next, size and values and sorted arrays are maintained to satisfy the following invariants:

– $\mathsf{values}[0] = \infty$

```
(1)   function get_id(value: $\mathbb{Q}$)
(2)   begin
(3)       lower $\leftarrow$ 0;
(4)       upper $\leftarrow$ next - 1;
(5)
(6)       while (lower $\leqslant$ upper)
(7)       begin
(8)           mid $\leftarrow$(lower + upper) div 2;
(9)           if (value < sorted[mid].value) upper $\leftarrow$ mid - 1;
(10)          else if (value > sorted[mid].value) lower $\leftarrow$ mid + 1;
(11)          else return sorted[mid].id;
(12)      end
(13)
(14)      if (next $\geqslant$ size)
(15)          return lower;
(16)      else
(17)      begin
(18)          for (i = next; i > lower; i $\leftarrow$ i - 1)
(19)          begin
(20)              sorted[i].value $\leftarrow$ sorted[i - 1].value;
(21)              sorted[i].id $\leftarrow$ sorted[i - 1].id;
(22)          end
(23)
(24)          sorted[lower].value $\leftarrow$ value;
(25)          sorted[lower].id $\leftarrow$ next;
(26)          values[next] $\leftarrow$ value;
(27)
(28)          next $\leftarrow$ next + 1
(29)          return sorted[lower].id;
(30)      end
```

Fig. 6: Searching and extending the sorted and values arrays (with idealised arithmetic)

- $1 \leqslant \mathsf{next} \leqslant \mathsf{size}$
- $\forall 0 < i < \mathsf{next}.(\mathsf{sorted}[i-1].\mathsf{value} < \mathsf{sorted}[i].\mathsf{value})$
- $\forall 0 \leqslant i < \mathsf{next}.\exists 0 \leqslant j < \mathsf{next}.(\mathsf{sorted}[j].\mathsf{value} = \mathsf{values}[i] \wedge \mathsf{sorted}[j].\mathsf{id} = i)$

3.2 CoDBM algorithms

Figure 6 presents the get_id function which maps its single argument, value, to an identifier. If value has been encountered previously, its identifier is returned, otherwise a fresh identified is allocated and returned, and the arrays values and sorted adjusted to compensate.

The get_id function applies applies binary (or half-interval) search [26] on lines 3-12. Line 8 assigns mid to the semi-sum of lower and upper using integer division, which rounds towards 0. The while loop will terminate within

```
(1)  procedure set_ddbm_entry(ddbm: $\mathbb{N}^*$, i: $\mathbb{N}$, value: $\mathbb{Q}$)
(2)  begin
(3)      ddbm[i] $\leftarrow$ get_id(value);
(4)  end
(5)
(6)  function get_ddbm_entry(ddbm: $\mathbb{N}^*$, i: $\mathbb{N}$)
(7)  begin
(8)  return values[ddbm[i]];
(9)  end
```

Fig. 7: Reading (getting) and writing (setting) an entry of a CoDBM

$\lceil \log_2(\mathsf{size}) \rceil + 1$ iterations, either returning the identifier of value, or exiting at line 14 with $\mathsf{lower} = \mathsf{upper} + 1$. To see this, let lower' and upper' denote the values of these variables on loop exit, and their unprimed counterparts denote their values at the start of the last iteration. Hence $\mathsf{lower} \leqslant \mathsf{upper}$ and $\mathsf{lower}' > \mathsf{upper}'$. If $\mathsf{upper}' = \mathsf{mid} - 1$ then $\mathsf{mid} - 1 = \mathsf{upper}' < \mathsf{lower}' = \mathsf{lower} \leqslant \mathsf{mid}$ hence $\mathsf{lower} = \mathsf{mid}$ whence $\mathsf{upper}' = \mathsf{mid} - 1 = \mathsf{lower} - 1 = \mathsf{lower}' - 1$. Conversely if $\mathsf{lower}' = \mathsf{mid} + 1$ it likewise follows that $\mathsf{lower}' = \mathsf{upper}' + 1$.

The exit condition $\mathsf{lower} = \mathsf{upper} + 1$ indicates where to insert a new record for value in sorted. Observe that if $0 \leqslant i < \mathsf{lower}$ then $\mathsf{sorted}[i].\mathsf{value} < \mathsf{value}$ and conversely if $\mathsf{upper} < i < \mathsf{next}$ then $\mathsf{sorted}[i].\mathsf{value} > \mathsf{value}$. In particular $\mathsf{sorted}[\mathsf{upper}].\mathsf{value} < \mathsf{value} < \mathsf{sorted}[\mathsf{lower}].\mathsf{value}$ indicating that the value record needs to be inserted at position lower of sorted, once the record at this position and the higher positions are all shuffled along. The for loop commencing at line 18 enacts the shuffle and lines 24 and 25 adjust the record at position lower record value and its identifier next. Line 26 updates values to map the identifier next to value. The next counter is updated at line 18 and the identifier for value returned at line 29.

The check at line 14 detects whether the capacity of the arrays values and sorted are exceeded. Suppose this line is reached. Because values and sorted are initialised to store the value ∞ (coupled with the 0 identifier) it follows $\mathsf{value} \neq \infty$ hence $\mathsf{value} \in \mathbb{Q}$. Thus there exists a recorded value (even if it is ∞) which is strictly larger than value and indeed $\mathsf{value} < \mathsf{sorted}[\mathsf{lower}].\mathsf{value}$. Moreover $\mathsf{sorted}[\mathsf{lower} - 1].\mathsf{value} = \mathsf{sorted}[\mathsf{upper}].\mathsf{value} < \mathsf{value}$ hence lower is the identifier for the smallest value strictly larger than value. This provides a way to update a CoDBM entry with a relaxed value when one does not want to resize the arrays. This, in effect, widening the CoDBM in a way that is sensitive to space capacity. For completeness, figure 7 shows how the entries of a CoDBM are read and written, where for a CoDBM of dimension n, i is in index into the linear array of identifiers, hence $0 \leqslant i < 2n(n + 1)$.

Finally to remark on complexity, the set_ddbm_entry function resides in $O(n)$, where n is the size of the value and sorted arrays, because of the potential for copying in the get_id function. The get_ddbm_entry function is in $O(1)$.

```
(1)     function get_id(value: $\mathbb{Q}$)
(2)     begin
(3)         lower' $\leftarrow$ 1;
(4)         upper' $\leftarrow$ next;
(5)
(6)         while (lower' $\leqslant$ upper')
(7)         begin
(8)             mid' $\leftarrow$ lower' + ((upper' - lower') div 2);
(9)             if (value < sorted[mid' - 1].value) upper' $\leftarrow$ mid' - 1;
(10)            else if (value > sorted[mid' - 1].value) lower' $\leftarrow$ mid' + 1;
(11)            else return sorted[mid' - 1].id;
(12)        end
(13)        lower $\leftarrow$ lower' - 1
...         ...
(31)    end
```

Fig. 8: Searching and extending the sorted and values arrays (with machine arithmetic)

3.3 Binary search with machine arithmetic

By way of a postscript, figure 8 gives a revised version of get_id which is sensitive to the limitations of machine arithmetic. Since natural numbers are used for identifiers it is natural to employ unsigned integers. However, if $\mathsf{lower} = \mathsf{upper} = 0$ then $\mathsf{mid} = 0$ hence $\mathsf{mid} - 1$ will underflow at line 9. The listing in figure 8 avoids this adding a positive offset of 1 to lower and upper to give lower' and upper' which is duly compensated for at line 13. Overflow will not occur on $\mathsf{mid}' + 1$ at line 10, however, if size, hence next, is strictly smaller than the largest representable number. Another subtlety is that the semi-sum $(\mathsf{lower} + \mathsf{upper})\ \mathsf{div}\ 2$ on line 8 of Figure 6 can overflow [22]. Hence the alternative formulation of $\mathsf{lower}' + ((\mathsf{upper}' - \mathsf{lower}')\ \mathsf{div}\ 2)$ in figure 8.

4 Experiments

The abstract interpretation plugin for Frama-C, EVA [8], was used for gathering salient statistics on octagons, and then comparing CoDBMs against DBMs. The statistics were gathered on a Linux box equipped with 128GB of RAM and dual 2.0GHz Intel Xeon E5-2650 processors. EVA has options to use the Apron library [13], a widely-used numerical domain library, with implementations of polyhedra, boxes (intervals) and octagons. EVA is a prototype analyser for C99, and as such does not provide state-of-the-art optimisations such as automatic variable clustering [12] or access-based localisation [4]; which precludes the analysis of very large programs. Therefore, table 1 lists modestly sized programs drawn from the Frama-C open source case studies repository (`github.com/Frama-C/open-source-case-studies`), which were used for benchmarking. The case studies repository was designed to test the default value analysis

Abbrv	Benchmark	LOC	Description
lev	levenstein	187	Levenstein string distance library
sol	solitaire	334	card cipher
2048	2048	435	2048 game
kh	khash	652	hash code from klib C library
taes	Tiny-AES	813	portable AES-128 implementation
qlz	qlz	1168	fast compression library
mod	libmodbus	7685	library to interact with Modbus protocol
mgmp	mini-gmp	11787	subset of GMP library
unq	unqlite	64795	embedded NoSQL DB
bzip	bzip-single-file	74017	bzip single file for static analysis benchmarking

Table 1: Benchmarks

Abbrv	#Ids	#Entries	#Reads	#Writes	DBM Time	CoDBM Time	Speedup
lev	900	795345356	221230162	76969699	14.18	10.59	25%
sol	2161	3044202788	565119059	258963325	45.44	33.70	25%
2048	358	1919995058	445261291	144479736	24.44	16.53	32%
kh	196	30165440	8465749	3640830	1.37	1.32	3%
taes	140	106396722938	-	-	803.30	505.90	37%
qlz	10	69126	17216	13742	1.32	1.41	-7%
mod	3627	45157313792	1921289169	661225970	336.22	214.10	36%
mgmp	126	101752918	22603122	8805323	2.01	1.56	22%
unq	-	-	-	-	1.50	1.56	-4%
bzip	262	373715276692	-	-	591.85	196.69	66%
total					1821.63	983.36	46%

Table 2: Experimental Statistics (tabulated)

plugin of Frama-C, and not the EVA plugin, and in fact some of the case studies did not terminate using the EVA plugin with the octagon domain.

4.1 Execution Time

Table 2 reports the headline results, giving both the key statistics, and timing information. The #Ids and #Entries columns give, respectively, the total number of identifiers allocated using CoDBMs, and the exact number of matrix entries required across all DBMs. The former details how many rationals need be stored for CoDBMs; the latter for DBMs. The upper graph of figure 9 illustrates these numbers, showing that the reduction in memory allocation and initialisation is typically by three orders of magnitude. The #Reads and #Writes columns report the total number and reads and writes to the difference matrices (which is the same for both DBMs and CoDBMs). The lower graph of figure 9 illustrates these as proportions, showing that there are typically 2 and 3 times as many reads

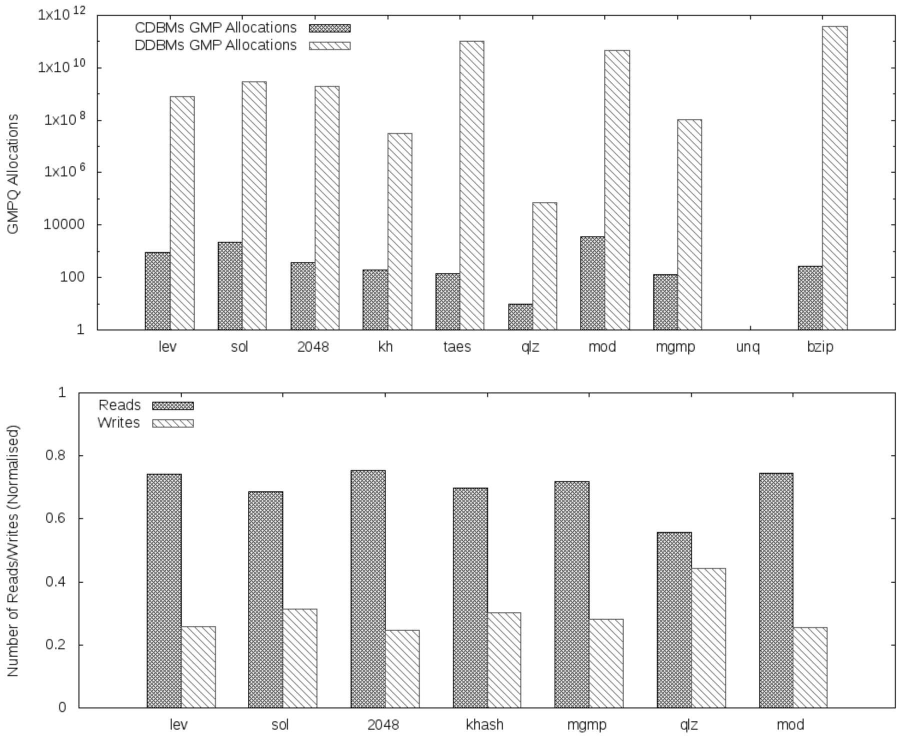

Fig. 9: Experimental Statistics (visualised)

as writes to the difference matrices entries over the lifetime of an analysis. The exception is unq. (Statistics are omitted for runs with excessively long traces.)

The table also reports the total execution time, in seconds, for the DBM and CoDBM analyses to parse their input, reach the fixpoint, and then output their results (which were identical). The last row totals the execution times over all benchmarks. CoDBMs are faster than DBMs on the longer-running analyses. The unq benchmark is a surprising corner case which does not produce any two variable constraints, and thus does not create any difference matrices. This gives an inexplicable slowdown, possibly due to an increase in code size. CoDBMs give a modest slowdown for unq, presumably because of the higher proportion of writes to reads; elsewhere CoDBMs give a significant speedup.

4.2 Memory Consumption

Table 3 shows end-to-end memory consumption statistics of each Frama-C benchmark. Memory consumption was measured using the GNU time utility which returns the maximum resident set size of the process during its lifetime. The

Benchmark	DBM	CoDBM	Reduction (%)
lev	3317872	2531096	23.71
sol	11694896	7538160	35.54
2048	6389484	3584856	43.89
kh	284776	198656	30.24
taes	35453276	32361744	8.72
qlz	332252	331788	0.14
mod	51832508	31831468	38.59
mgmp	443932	236236	46.79
unq	207332	208560	-0.59
bzip	11047816	5781492	47.67
Total	121335932	84604056	30.27

Table 3: Memory Usage Statistics (in Kb)

results show that memory consumption is reduced in all but one of the benchmarks. Memory consumption is most notably reduced for the longer running benchmarks, with bzip showing the greatest reduction. Overall, the results show that the CoDBM representation has a positive effect in reducing the memory usage of the Apron library.

4.3 Cache Behaviour

Modern processors have become reliant on caching as the speed of the core has steadily outstripped the clock frequency of the memory bus and the performance of RAM. For example, the 2.0 GHz Xeon (Sandy Bridge) E5-2650 has a separate level 1 instruction cache and level 1 data cache, and unified caches at level 2 and 3. The latency of the level 3 cache is 28 cycles, whereas the latency of RAM is 28 cycles plus 49 nanoseconds [21], which equates to $28 + (49 \times 10^{-9}) \times (2 \times 10^9) = 218$ cycles. This underscores the importance of reducing cache misses, even by small percentages, and the role of temporal and spatial locality to the performance of DBMs and CoDBMs. Reading an element from a CoDBM incurs an extra layer of indirection compared to a DBM and writing to a CoDBM can incur multiple memory references, so one might expect these additional memory references to put pressure on the cache and worsen memory performance. This section thus investigates the memory behaviour of CoDBMs.

Table 4 summarises memory and cache statistics for CoDBMs and DBMs gathered with Cachegrind [19]. Cachegrind simulates a theoretical architecture with a first-level data (D1) cache, first-level instruction (I1) cache and a last-level (LL) unified data and instruction cache. LL hints at the way Cachegrind abstracts a three cache architecture: it auto-detects the cache configuration (cache size, associativity and line size) and sets LL to model the L3 rather than the L2 cache. This is because missing L3 has more effect on runtime than missing L2. Although Cachegrind abstracts the cache structure of a Xeon, it nevertheless reports accurate counts for the total number of memory references.

activity		lev	sol	2048	kh	teas	qlz	mod	mgmp	unq	bzip
D refs	r	22b	14b	63b	825m	696b	358m	379b	3b	5b	1125b
	w	17b	11b	48b	559m	511b	207m	285b	2b	3b	918b
	t	**39b**	**25b**	**111b**	**1,384m**	**1,207b**	**566m**	**665b**	**6b**	**8b**	**2044b**
D1 misses	r	1.8%	2.1%	2.0%	2.0%	2.0%	1.9%	1.9%	1.9%	1.4%	2.5%
	w	1.6%	1.9%	1.6%	2.0%	1.6%	2.5%	1.6%	2.0%	2.9%	1.8%
	t	**1.7%**	**2.0%**	**1.8%**	**2.0%**	**1.8%**	**2.1%**	**1.8%**	**2.0%**	**2.0%**	**2.2%**
LL misses	r	0.1%	0.1%	0.1%	0.0%	0.5%	0.o%	0.1%	0.0%	0.1%	0.2%
	w	0.5%	0.7%	0.5%	0.3%	0.6%	0.3%	0.8%	0.3%	0.1%	1.1%
	t	**0.1%**	**0.1%**	**0.1%**	**0.1%**	**0.5%**	**0.1%**	**0.2%**	**0.1%**	**0.1%**	**0.3%**
D refs	r	16b	8b	35b	681m	512b	205m	200b	2.5b	5b	234b
	w	9b	5b	20b	407m	258b	104m	106b	1.4b	3b	92b
	t	**25b**	**13b**	**55b**	**1,088m**	**771b**	**310m**	**306b**	**4b**	**8b**	**326b**
D1 misses	r	0.7%	1.2%	0.7%	1.6%	0.3%	2.4%	0.4%	1.2%	1.4%	0.4%
	w	0.9%	1.4%	1.1%	1.9%	0.5%	2.3%	0.8%	1.9%	2.9%	0.6%
	t	**0.8%**	**1.3%**	**0.9%**	**1.7%**	**0.4%**	**2.4%**	**0.5%**	**1.4%**	**2.0%**	**0.4%**
LL misses	r	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.1%	0.0%
	w	0.0%	0.7%	0.5%	0.3%	0.2%	0.5%	0.5%	0.2%	0.1%	0.0%
	t	**0.0%**	**0.1%**	**0.0%**	**0.0%**	**0.0%**	**0.1%**	**0.0%**	**0.0%**	**0.1%**	**0.0%**

Table 4: Cachegrind statistics: DBMs upper-half; CoDBMs lower-half

While data handling is determined by the programmer in the design of the data-structures, the compiler itself will generate the low-level instructions, and normally do so in way that aids spacial and temporal locality. Indeed the I1 statistics for the DBM and CoDBM code are almost identical, and therefore not discussed further.

Number of memory references Table 4 focuses on D1 and LL misses, and the number of memory references (D refs). The table presents the total number of D refs (t), as well as the number of reads (r) and writes (w), where k, m and b respectively denote the multipliers 10^3, 10^6 and 10^9. The table also details the number of cache misses for D1 and LL, expressed as a percentage of the total number of memory references. These statistics are augmented with percentages for the number of misses on read and write. The upper portion of the table presents the statistics for DBMs; the lower for CoDBMs.

The main result is that, with the exception of unq, the number of reads and writes to memory are both reduced with CoDBMs. For 2048, mod and bzip in particular, the difference to the number of writes is considerable and reflects the CoDBM design which factors out shared sub-structure in the difference matrix. Specifically, each distinct number is allocated and initialised exactly once over the lifetime of the whole analysis. Thus, when a fresh CoDBM is created, in contrast with a DBM, it is not necessary to allocate fresh memory for each rational stored in the CoDBM: only those few, if any, which have not been encountered previously. In fact, the most common use-case for CoDBM creation involves merely allocating the matrix of identifiers and then setting each cell of the matrix to the identifier for ∞. This initialises the matrix to represent a system of vacuous difference constraints. CoDBM creation thus saves many memory references, particularly writes, relative to DBMs. Copying a CoDBM likewise avoids multiple memory allocations and initialisations, since only the matrix of identifiers need be created and then copied over one-by-one, again saving many memory references, mainly writes, relative to the equivalent operation on a DBM.

Furthermore, although reading an entry of a CoDBM requires one extra memory reference per read, for an incremental closure algorithm [9], the total number of reads is actually is only linear in the size of the matrix which, in turn, is quadratic in the number of program variables. Moreover, incremental closure is the most frequently applied domain operation, after the creation and copying a difference matrix. In fact, with code hoisting, the inner loop of incremental closure (which dominates the overall cost) requires just read operation per iteration [9]. Hence the increase in memory references though the reads of incremental closure is more than compensated by the reduction obtained by factoring out matrix sub-structure, and the subsequent reduction in memory allocation and initialisation.

As a final check, taes was again instrumented with Callgrind, which revealed that the number of calls to gmpq_init drops from 36% of the total using DBMs to 12% using CoDBMs. This is not reduced to zero because of the need to store intermediate rational numbers in various domain operations.

Locality of the memory references The D1 and LL misses reported in table 4 act as a proxy for temporal and spatial locality and suggest that, despite the extra reference incurred on each read, there is an overall, sometimes dramatic, improvement to locality. The significant result here is that in the majority of cases the number of LL misses reduce to 0%. We suspect that locality is improved, in part, because a CoDBM is denser than the equivalent DBM so that a single cache line stores more CoDBM entries than DBM entries. Moreover, the values and sorted arrays are often much smaller than a moderately sized CoDBM, which is itself much smaller than the corresponding DBM. This again is good for locality. The table also explains the slowdown on qlz: although the number of memory references almost halve, the number of D1 misses increase, which illustrates the importance of locality.

But figure 10 reveals a more subtle explanation for locality. Figure 10 gives distributions of the identifiers which arise on CoDBM reads during the analysis of taes, lev and sol. For a given prefix of the values array, each distribution gives the total number of times the array prefix is used to map an identifier to a value. The distributions are normalised. It is interesting to see that accesses cluster at the bottom end of values, implying to there is both temporal and spacial locality to the way the array is accessed. In fact, for taes, the first 7 elements of values attract over 75% of the reads, and for both lev and sol the first 3 elements are responsible for over 75% of the reads. This bias is not unique to taes, lev and sol; it is a general pattern.

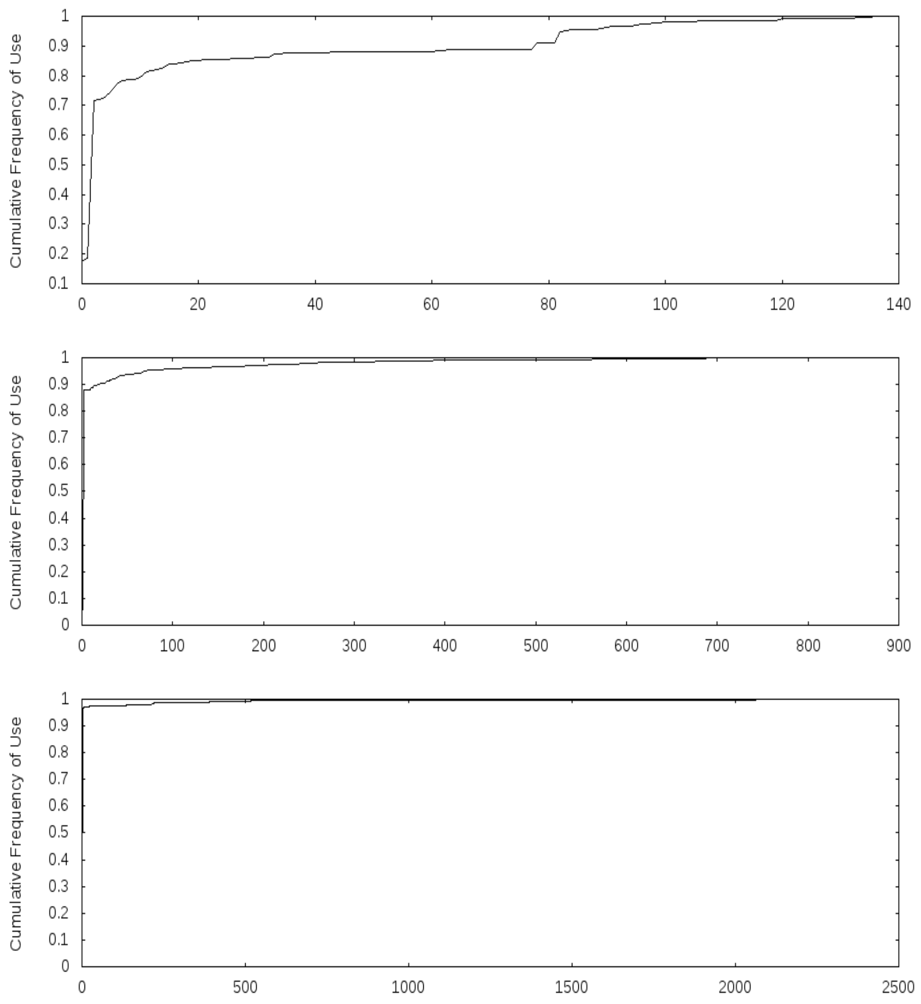

Fig. 10: Cumulative Frequency Distribution of Identifiers (taes, lev, sol)

This pattern of access suggests that the portion of `values` needed for a read is likely to reside in a cache line. This stems from the way identifiers are created on demand when a matrix is updated with a previously unseen number. Numbers which arise with highest frequency tend to be encountered earlier in the lifetime of an analysis hence are inclined to be assigned smaller identifiers, giving the biased frequency distributions of figure 10. Moreover, the preallocated value ∞, needed for space widening, is also accessed with high frequency.

The first few iterations of the while loop at line 6 of figure 6 are not likely to be conducive to locality, which suggests that a high proportion of writes to reads is likely to degrade locality, which squares with unq.

5 Related Work

A DBM can track any octagonal constraint between any pair of two variables, whether that constraint is needed or not. In response, variable clustering has been proposed [6,17,24] for grouping variables into sets which scope the relationships which are tracked. However, deciding variable groupings is an art, although there has been recent progress made in its automation [12].

Other remedies for the size of DBMs include sparse analyses [20] and access-based localisation techniques [4]. Access-based localisation uses scoping heuristics to adjust the size of the DBM to those variables that can actually be updated [4]. Sparse analyses generalise access-based localisation techniques, using data dependencies to adjust the size of abstract states propagated to method calls: [20] defines a generic technique to apply sparse techniques to abstract interpretation and combines this with variable packing to scale an octagon based abstract interpreter for C programs. Variable clustering, access-based localisation and sparse frameworks are orthogonal to our work, and can take advantage of the CoDBMs introduced in this paper.

Sparse matrix representations have been proposed for Octagons [14] and Differences [11] as an alternative to DBMs, but these representations sit at odds with the simplicity of the original domains algorithms. The desirable property of strong closure [18] (the normal form for Octagons) does not hold for a sparse representation, motivating the need to rework the domain operations in order to retain precision [14]. The CoDBM representation proposed in this paper is reminiscent of a sparse set representation [7], which likewise uses two arrays. The key difference is that sparse sets use their two arrays to reference one another to speed up set operations. They are also limited to a fixed size universe. CoDBMs, on the other hand, uses a single lookup table to store all rational numbers occurring in all the difference matrices and moreover the elements of the difference matrices are not prescribed upfront.

There is a trend towards using doubles instead of rationals so as to take advantage of modern instruction sets [23]. In contrast, we aim to retain the precision of rationals and not sacrifice performance for soundness. The memory footprint of a DBM directly relates to the underlying arithmetic: a double requires 8 bytes in the IEEE 754-1983 standard and a rational number at least 12 bytes using the GNU multiple precision library applied in Apron [13]. But if space is the primary concern, then a CoDBM over rationals will be smaller than its corresponding DBM over doubles.

6 Discussion

CoDBMs are designed to change the economy of difference matrices: the overheads of memory management in DBMs are exchanged in CoDBMs for the costs of reading the matrix through an indirection and writing to the matrix through search. Callgrind instrumentation shows that one hot spot occurs at lines 9 and 10 of the get_id function, in comparing rational numbers. If the writes to the matrix have temporal locality, then a splay trees [1] could improve search by ensuring that recently accessed rationals will require fewer comparisons to be found again. Alternatively hashing could reduce the number of comparisons, while matching the conceptual simplicity of Octagons.

We suspect that CoDBMs will confer further advantages for parallelisation where multiple processes increase the size of the working set, and exert additional pressure on the caches. It is worth noting that CoDBMs were designed to counter the size of DBMs, but actually are faster too. As qlz illustrates, the key difference is not the number of memory references, but locality. Moreover, as CPUs continue to out pace memory, locality will only become a more important issue in the implementation and realisation of abstract domains.

Finally, it is important to emphasise that CoDBMs are not a substitute for variable clustering and access-based localisation: these are orthogonal techniques that can benefit from CoDBMs.

7 Conclusions

This paper proposes CoDBMs as a compact, new representation for difference bound matrices (DBMs) in which each distinct rational number in the DBM is assigned a unique (small) identifier. The identifier is then stored as an entry in the matrix as a substitute for the number it represents. The matrix is used in tandem with two arrays. The first array, used for reading the matrix, maps the identifier to its number. The second array is ordered, used for writing to the matrix, maps an number to its identifier (to provide a partial inverse). CoDBMs retain the regular structure of DBMs, hence the simple, attractive loop structures of their domain operations, but also reduce space consumption while improving cache behaviour, and speeding up long-running analyses.

Acknowledgements We thank Colin King at Canonical Limited for his tireless patience and help with the performance analysis which underpinned this work.

References

1. B. Allen and I. Munro. Self-Organizing Binary Search Trees. *Journal of the ACM*, 25(4):526–535, 1978.
2. R. Bagnara, P. M. Hill, and E. Zaffanella. Weakly-relational Shapes for Numeric Abstractions: Improved Algorithms and Proofs of Correctness. *Formal Methods in System Design*, 35(3):279–323, 2009.
3. F. Banterle and R. Giacobazzi. A Fast Implementation of the Octagon Abstract Domain on Graphics Hardware. In *SAS*, volume 4634 of *LNCS*, pages 315–335. Springer, 2007.
4. E. Beckschulze, S. Kowalewski, and J. Brauer. Access-Based Localization for Octagons. *Electronic Notes in Theoretical Computer Science*, 287:29–40, 2012.
5. R. Bellman. On a Routing Problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

6. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A Static Analyzer for Large Safety-Critical Software. In *PLDI*, pages 196–207, 2003.
7. P. Briggs and L. Torczon. An Efficient Representation for Sparse Sets. *ACM Letters on Programming Languages and Systems*, 2(1-4):59–69, 1993.
8. D. Bühler, P. Cuoq, B. Yakobowski, M. Lemerre, A. Maroneze, V. Perrelle, and V. Prevosto. *The EVA Plug-in.* CEA LIST, Software Reliability Laboratory, Saclay, France, F-91191, 2017. `https://frama-c.com/download/frama-c-value-analysis.pdf`.
9. A. Chawdhary, E. Robbins, and A. King. Simple and Efficient Algorithms for Octagons. In *APLAS*, volume 8858 of *LNCS*, pages 296–313. Springer, 2014.
10. D. Dill. Timing Assumptions and Verification of Finite-State Concurrent Systems. In J. Sifakis, editor, *CAV*, volume 407 of *LNCS*, pages 187–212. Springer, 1989.
11. G. Gange, J. A. Navas, P. Schachte, H. Søndergaard, and P. Stuckey. Exploiting Sparsity in Difference-bound Matrices. In *SAS*, volume 9837 of *LNCS*, pages 189–211, 2016.
12. K. Heo, H. Oh, and H. Yang. Learning a Variable-Clustering Strategy for Octagon From Labeled Data Generated by a Static Analysis. In *SAS*, volume 9837 of *LNCS*, pages 237–256, 2016.
13. B. Jeannet and A. Miné. APRON: A library of numerical abstract domains for static analysis. In *CAV*, volume 5643 of *LNCS*, pages 661–667, 2009.
14. J.-H. Jourdan. *Verasco: a Formally Verified C Static Analyzer.* PhD thesis, Université Paris Diderot (Paris 7) Sorbonne Paris Cité, May 2016.
15. J.-L. Lassez, T. Huynh, and K. McAloon. Simplication and Elimination of Redundant Linear Arithmetic Constraints. In *Constraint Logic Programming*, pages 73–87. MIT Press, 1993.
16. M. Measche and B. Berthomieu. Time Petri-nets for analyzing and verifying time dependent communication protocols. In H. Rudin and C. West, editors, *Protocol Specification, Testing and Verification III*, pages 161–172. North-Holland, 1983.
17. A. Miné. *Weakly Relational Numerical Abstract Domains.* PhD thesis, École Polytechnique En Informatique, 2004.
18. A. Miné. The Octagon Abstract Domain. *HOSC*, 19(1):31–100, 2006.
19. N. Nethercote. *Dynamic Binary Analysis and Instrumentation.* PhD thesis, Trinity College, University of Cambridge, 2004.
20. H. Oh, K. Heo, W. Lee, W. Lee, D. Park, J. Kang, and K. Yi. Global Sparse Analysis Framework. *ACM TOPLAS*, 36(3):8:1–8:44, 2014.
21. I. Pavlov. Lempel-Ziv-Markov chain Algorithm CPU Benchmarking, 2017. `http://www.7-cpu.com/`.
22. S. Ruggieri. On Computing the Semi-Sum of Two Integers. *Information Processing Letters*, 87(2):67–71, 2003.
23. G. Singh, M. Püschel, and M. Vechev. Making Numerical Program Analysis Fast. In *PLDI*, pages 303–313. ACM Press, 2015.
24. A. Venet and G. Brat. Precise and Efficient Static Array Bound Checking for Large Embedded C Programs. In *PLDI*, pages 231–242, 2004.
25. J. Weidendorfer, M. Kowarschik, and C. Trinitis. A Tool Suite for Simulation Based Analysis of Memory Access Behavior. In *International Conference on Computational Science*, volume 3038 of *LNCS*, pages 440–447. Springer, 2004.
26. L. F. Williams Jr. A Modification to the Half-Interval Search (Binary Search) Method. In *Proceedings of the 14th ACM Southeast Conference*, pages 95–101.