SPaC: A Symbolic Pareto Calculator *

Hamid Shojaei, Twan Basten, Marc Geilen, Phillip Stanley-Marbell
Department of Electrical Engineering, Eindhoven University of Technology, Eindhoven, Netherlands
{h.shojaei,a.a.basten,m.c.w.geilen,p.stanley-marbell}@tue.nl

ABSTRACT

The compositional computation of Pareto points in multi-dimensional optimization problems is an important means to efficiently explore the optimization space. This paper presents a symbolic Pareto calculator, SPaC, for the algebraic computation of multi-dimensional trade-offs. SPaC uses BDDs as a representation for solution sets and operations on them. The tool can be used in multi-criteria optimization and design-space exploration of embedded systems. The paper describes the design and implementation of Pareto algebra operations, and it shows that BDDs can be used effectively in Pareto optimization.

Categories and Subject Descriptors

C.3 [**Special-purpose and Application-based Systems**]: Realtime and embedded systems; I.1.1 [**Symbolic and Algebraic Manipulation**]: Expressions and Their Representation—*Representations*

General Terms

Algorithms, Design, Experimentation

1. INTRODUCTION

Multi-dimensional optimization has always been among the most challenging issues in embedded system design. Pareto algebra [8] is a method to compositionally specify and compute trade-offs in multi-dimensional optimization problems. The method helps to alleviate phase coupling or design-closure problems in a design trajectory or can be used for combined off-line/run-time Quality-of-Service management [8, 10]. In Pareto algebra, each parameter, quality metric, or other aspect of interest of a particular system is considered as a *quantity* and different options for realizing the system are considered as the system *configurations*. A minimal set of configurations which are Pareto optimal with respect to others is referred to as a *Pareto minimal set*. A *dominance relation* is used to find optimal configurations of the system.

In [7], a Pareto calculator is proposed for compositional computation of Pareto points based on Pareto algebra. The tool uses set data structures to store quantities and configurations explicitly. A Pareto algebra implementation involves working with many structures that contain sets and relations. For example, a *Pareto space* consists of a set of quantities, a set of configurations, a dominance relation per quantity, a dominance relation on the space, and a Pareto minimal set. Representing such sets and relations explicitly is often inefficient and sometimes infeasible. In addition, a Pareto algebra operation such as a Cartesian product results in an exponential increase in the size of the representation. It is therefore desirable to exploit regularity in the sets to obtain a more compact representation.

Binary Decision Diagrams (BDDs) [4] are well known as a means to efficiently, symbolically, represent sets and relations [5]. Therefore, we developed SPaC, a BDD-based symbolic Pareto calculator. BDDs have been used for many purposes including symbolic model checking [11], to represent the alphabet of automata in the tool MONA [9], in the RELVIEW system [3], an interactive manipulator of binary relations with a graphical user interface, in the GBDD package [1] to represent states and transition relations of infinite-state automata in a regular model-checking framework, and in Jedd [12] to provide a language extension to Java that supports a convenient way of programming with BDDs. The mentioned tools all use BDDs as an abstraction for sets and relations. Most are designed around binary relations. Much of the complexity of our work stems from the need to represent multi-dimensional configurations. GBDD and Jedd support n-ary relations. Building upon the BDD representations and notations for sets and relations of [4, 5], we combine ideas from GBDD and Jedd to represent multi-dimensional configurations, and extend the approach to support non-integer values. We then instead of relational operations, provide an abstraction level on top of BDDs to have full support for Pareto algebra operations. Our main contributions are efficient BDD representations of the various Pareto algebra concepts and a symbolic algorithm for computing the set of Pareto configurations from a configuration set. As BDDs have been in use for some time, there exist several excellent libraries providing efficient representations, algorithms, and memory management techniques for BDDs, including the C-based CUDD [13] package, which we use in our tool.

The paper is organized as follows. An overview of Pareto algebra is provided in the next section. Section 3 presents the proposed design and symbolic implementation of the Pareto calculator and our proposed symbolic minimization algorithm. Section 4 discusses memory usage and time complexity of our algorithms. Some experimental results are presented in Section 5. Section 6 concludes.

*This work was supported in part by the EC through FP7 IST project 216224, MNEMEE.

CODES+ISSS'08, October 19–24, 2008, Atlanta, Georgia, USA.

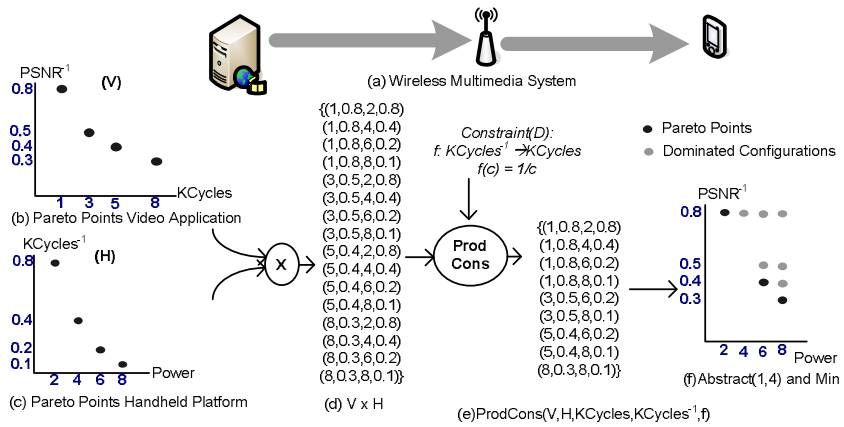

Figure 1: Computing Pareto points of a video decoder running on a handheld.

2. PARETO ALGEBRA

This section provides an overview of Pareto algebra [8]. We briefly define all concepts and operations for multi-dimensional optimization in this algebra. We use a small case study in which an MPEG-4 stream is sent from a media center through a wireless connection to a PDA taken from [7] (Fig. 1.a) as a running example. The goal is to find optimal trade-offs between video quality and power consumption of the video decoder mapped onto the PDA.

A *quantity* is a parameter, quality metric, or any other quantified aspect of a system. It is represented as a set Q with partial order $\preceq_Q$. The subscript is dropped when clear from the context. If $\preceq_Q$ is total, the quantity is *basic*; if $\preceq_Q$ is the identity, the quantity is *unordered*. In examples, we always enforce that less is better. In Fig. 1.b, $PSNR^{-1}$ and *KCycles* are two basic quantities with total order $\leq$.

A configuration space $\mathcal{S}$ is the Cartesian product $Q_1 \times Q_2 \times \ldots \times Q_n$ of a finite number n of quantities and a configuration $\bar{c} = (c_1, c_2, \ldots, c_n)$ is an element of such a space.

A dominance relation $\preceq \subseteq \mathcal{S}^2$ defines configurations that are preferred over others. If $\bar{c}_1, \bar{c}_2 \in \mathcal{S}$, then $\bar{c}_1 \preceq \bar{c}_2$ iff for every quantity Q_k of $\mathcal{S}$, $\bar{c}_1(Q_k) \preceq_{Q_k} \bar{c}_2(Q_k)$. If $\bar{c}_1 \preceq \bar{c}_2$, then $\bar{c}_1$ *dominates* $\bar{c}_2$, expressing that $\bar{c}_1$ is in all aspects at least as good as $\bar{c}_2$. Dominance is reflexive, i.e., a configuration dominates itself. The irreflexive strict dominance relation is denoted $\prec$. A configuration is a *Pareto point* of a configuration set iff it is not strictly dominated by any other configuration. Configuration set $\mathcal{C}$ is *Pareto minimal* iff it contains only Pareto points, i.e., for any $\bar{c}_1, \bar{c}_2 \in \mathcal{C}$, $\bar{c}_1 \not\prec \bar{c}_2$. Fig. 1.b and 1.c show the Pareto minimal configuration sets of the video application and the PDA in our example.

A crucial observation is that, by allowing partially ordered quantities, quantities, configuration sets, and configuration spaces become essentially the same concepts. Pareto algebra exploits this to define operations on configuration sets that can be used to compute or reason about Pareto points of composite systems. Let $\mathcal{C}$ and $\mathcal{C}_1$ be configuration sets of space $\mathcal{S}_1 = Q_1 \times Q_2 \times \ldots \times Q_m$ and $\mathcal{C}_2$ a configuration set of space $\mathcal{S}_2 = Q_{m+1} \times \ldots \times Q_{m+n}$. Let $k \in \{1, \ldots, m\}$. Let $\bar{c} \cdot \bar{d} = (c_1, \ldots, c_m, d_1, \ldots, d_n)$ for $\bar{c} = (c_1, \ldots, c_m) \in \mathcal{C}_1$ and $\bar{d} = (d_1, \ldots, d_n) \in \mathcal{C}_2$.

- $\min(\mathcal{C}) \subseteq \mathcal{S}_1$ is the set of Pareto points of $\mathcal{C}$: $\min(\mathcal{C}) = \{\bar{c} \in \mathcal{C} \mid \neg(\exists \bar{c}' \in \mathcal{C} : \bar{c}' \prec \bar{c})\}$.
- $\mathcal{C}_1 \times \mathcal{C}_2 \subseteq \mathcal{S}_1 \times \mathcal{S}_2$ is the (free) product of $\mathcal{C}_1$ and $\mathcal{C}_2$.
- $\mathcal{C} \cup \mathcal{C}_1 \subseteq \mathcal{S}_1$ is the set of alternatives of $\mathcal{C}$ and $\mathcal{C}_1$.
- $\mathcal{C} \cap \mathcal{C}_1 \subseteq \mathcal{S}_1$ is the $\mathcal{C}_1$-constraint of $\mathcal{C}$.
- $\mathcal{C} \downarrow k = \{(c_1, \ldots, c_{k-1}, c_{k+1}, \ldots, c_m) \mid (c_1, \ldots, c_m) \in \mathcal{C}\} \subseteq \mathcal{S}_1 \downarrow k = Q_1 \times \ldots \times Q_{k-1} \times Q_{k+1} \times \ldots \times Q_m$ is the k-abstraction of $\mathcal{C}$.
- Join($\mathcal{C}_1$,$\mathcal{C}_2$,Q)=$(\mathcal{C}_1 \times \mathcal{C}_2) \cap \mathcal{D}$, where $\mathcal{S}_1$ and $\mathcal{S}_2$ both include unordered quantity Q and constraint $\mathcal{D}$ is defined by $\mathcal{D} = \{\bar{c}_1 \cdot \bar{c}_2 | \bar{c}_1 \in \mathcal{S}_1, \bar{c}_2 \in \mathcal{S}_2, \bar{c}_1(Q) = \bar{c}_2(Q)\}$.
- ProdCons($\mathcal{C}_1, Q_1, \mathcal{C}_2, Q_2, f$)=$(\mathcal{C}_1 \times \mathcal{C}_2) \cap \mathcal{D}$, where Q_1 is a designated quantity (the producer quantity) of $\mathcal{S}_1$ and Q_2 (consumer quantity) of $\mathcal{S}_2$, and $\mathcal{D} = \{\bar{c}_1 \cdot \bar{c}_2 | \bar{c}_1 \in \mathcal{S}_1, \bar{c}_2 \in \mathcal{S}_2, \bar{c}_2(Q_2) \preceq f(\bar{c}_1(Q_1))\}$ with $f : Q_1 \to Q_2$ a monotonically decreasing function.

Fig. 1 illustrates the compositional computation of Pareto optimal system configurations (video-quality vs. power trade-offs) from Pareto optimal configurations of system components (the video decoder and handheld). Constraint $\mathcal{D}$ enforces that processor cycles required by the video decoder should be at most the cycles provided by the handheld.

3. IMPLEMENTING PARETO ALGEBRA

Bryant [5] describes how to represent sets and relations with BDDs and how to implement operations such as union and intersection. We rephrase these concepts in order to describe the symbolic Pareto algebra implementation. We follow the notation of [4], with the difference that we take the (equivalent) view that BDDs are sets of Boolean words rather than functions from Boolean variables to Booleans. We first describe how quantities, configurations, and dominance relations are represented in BDDs, and then discuss how the Pareto algebra operations are performed at the BDD level.

3.1 BDD Representations of Pareto concepts

Two basic concepts to be implemented are the quantity and configuration concepts. An important observation is that Pareto algebra supports arbitrary types, such as (subsets of) integers, reals, or enumerated types. To support arbitrary types we exploit the fact that concrete configuration sets are always finite. Therefore, our first design decision is to *represent quantities and their values only if they are observed during the algebraic calculations.* To this end, each time a new configuration set is observed or created, we first create new quantities as needed and *map* all the observed quantity values to consecutive subranges of integers. These

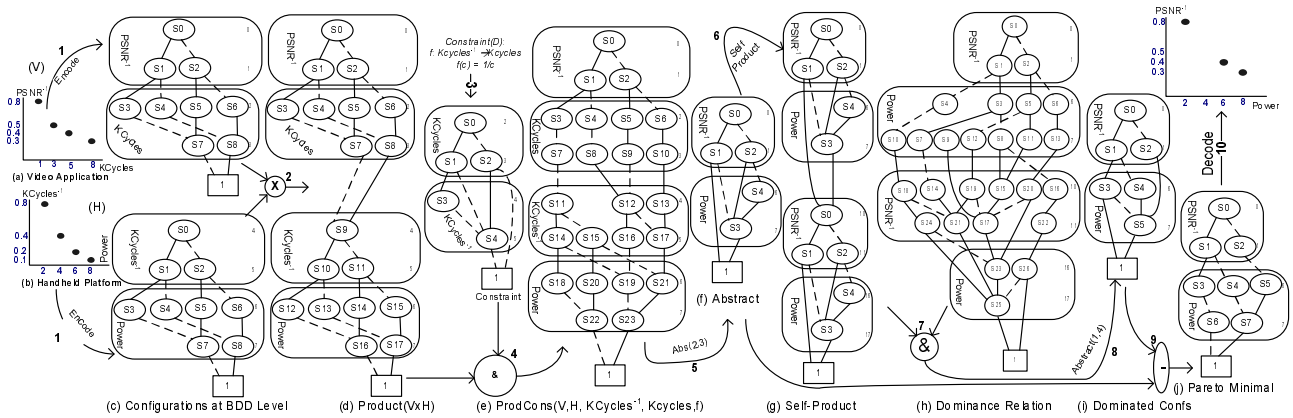

Figure 2: The symbolic computation of Pareto points of the video decoder running on the handheld.

integers are then *encoded* in terms of BDDs, storing the mapping function so it is possible to reconstruct the actual quantity values whenever needed. Our current implementation does not support the dynamic addition of values to already encoded quantities. We consider an extension allowing dynamic addition of quantity values an interesting topic for future work. As a second design decision, we decide to allocate a *fixed number of bits per quantity*, $\log_2 n$ bits for a quantity with n values.

Consider the example of Fig. 2.a. Both the $PSNR^{-1}$ and *KCycles* quantities can be coded with two bits. Values 0.3 and 0.5 of this quantity are mapped to integers 0 and 2, respectively, and binary codes 00 and 10, respectively.

A specific configuration in a Pareto space can now be represented by a Boolean word. For example, configuration (0.3,8) of the video decoder of Fig. 2.a is represented by Boolean word 0011. A configuration set is then easily represented as a BDD. For example, the BDD shown in the top part of Fig. 2.c captures the fact that (0.3,8) is a Pareto point of the video decoder via the S0S2S6S8 path, where the dashed lines denote a 0 edge and the solid lines a 1 edge.

We refer to the bit positions corresponding to a given quantity in the Boolean encoding of configurations as a *domain*. The top BDD in Fig. 2.c has two domains, 01 corresponding to quantity $PSNR^{-1}$ and 23 corresponding to *KCycles*. Since the number of quantities can vary dynamically during computations, as a third decision, we decide to *represent the universe of all quantities by an unbounded Boolean word*. New quantities can be added by assigning a new domain to them. We exploit the fact that such an addition does not affect already existing BDDs. For example, when observing quantities $KCycles^{-1}$ and *Power* from Fig. 2.b, we allocate domains 45 and 67 to them. This does not affect the BDD representation of the video configurations, and the encoding of the platform configurations (bottom part of Fig. 2.c) is independent of the domains 01 and 23 corresponding to the video quantities.

It remains to choose a representation for (strict) dominance relations on quantities and configuration spaces. Any relation can be represented as a set of pairs. Thus, a dominance relation can be represented by pairs of dominating-dominated configuration pairs. To encode a dominance relation on a set encoded with n bits, we select a domain in our unbounded Boolean word of $2n$ bits. The first n domain bits can be reused. For example, to represent the dominance relation on the video decoder configuration space given that domain bits 0-7 are allocated as discussed above, we may select domain 0189. The dominance relation then is a BDD over this domain. Fig. 2.h shows an example of the strict dominance relation in the 2-d $PSNR^{-1}$ and *Power* space. The BDD has domain 0 1 6 7 10 11 16 17. The path S0S2S6S13S20S23S25 encodes that configuration 0011 (0.3,8) strictly dominates both 1011 (0.5,8) and 1111 (0.8,8).

3.2 Symbolic Pareto algebra operations

Before discussing the symbolic implementations of the Pareto operations of Sec. 2, we observe that for some of these operations, the domain assignment must be the same for all operands. This requires a ***Renaming*** operation on BDDs, which takes a BDD and constructs a new BDD with the same configurations, but under a different domain assignment. This operation is implemented using an operation called *SwapVariables* in the CUDD package that we use.

The first two Pareto algebraic operations that we discuss, ***Alternative*** and ***Constraint***, correspond to standard set union and intersection operations. BDDs are in essence sets of words, and union and intersection are provided in CUDD. They are implemented using recursive procedures on the operand BDDs. The reader interested in the detailed BDD implementations is referred to [13].

To implement a ***free product*** in BDDs, we must first carefully set up the domain assignment. The domains of the two operands must be disjoint, in order to avoid that quantities present in both operands are identified in the product result. Such a domain assignment can be obtained via an appropriate renaming. Then, the product itself corresponds to a BDD intersection. Since the domains of the two operands are disjoint, the domain bits of one operand take arbitrary values from the viewpoint of the other operand. Set intersection then gives the desired result. In fact, for configuration sets $\mathcal{C}_1 \subseteq \mathcal{S}_1$ and $\mathcal{C}_2 \subseteq \mathcal{S}_2$ (with $\mathcal{S}_1$ and $\mathcal{S}_2$ spaces), the implementation uses the following identity: $\mathcal{C}_1 \times \mathcal{C}_2 = (\mathcal{C}_1 \times \mathcal{S}_2) \cap (\mathcal{S}_1 \times \mathcal{C}_2)$. The *product* of configurations of the video decoder and handheld configurations is shown in Fig. 2.d. As can be seen, a product boils down to merging the 1-terminal node of the first operand with the root node of the second operand.

Abstraction removes a quantity from a configuration set. In terms of our BDD representation, this means that the domain bits of the abstracted quantity may take arbitrary

Algorithm 1 Producer-Consumer Algorithm

```
1: ProducerConsumer(C_p, Q_p, C_c, Q_c, f)
2: set PIntValues = ExtractValues(C_p, Q_p);
3: set CIntValues = ExtractValues(C_c, Q_c);
4: BddBinaryRelation PC(domain(Q_p), domain(Q_c));
5: PC = empty;
6: for all i_p ∈ PIntValues do
7:    for all i_c ∈ CIntValues do
8:       if map(i_c) ⪯ f(map(i_p)) then
9:          add (i_c, i_p) to PC;
10:      end if
11:   end for
12: end for
13: res = restrict (product(C_p, C_c)) by PC;
14: return res;
```

Algorithm 2 Symbolic Minimization Algorithm

```
1: SymbolicMinimization(C, S)
2: dominance = GetDomSpace(S);
3: identity = IdentityRelation(S);
4: strictdominance = dominance − identity;
5: C_p = Product(C, C);
6: C_p = MapDomain(C_p, strictdominance);
7: dominatedset = strictdominance ∧ C_p;
8: for all Q ∈ S do
9:    dominatedset = Abstract(dominatedset, Q);
10: end for
11: dominatedset = MapDomain(dominatedset, S);
12: return C − dominatedset;
```

values in the resulting configuration set. The operation is implemented using the existential quantification BDD operation on all domain bits of the abstracted quantity. In step 5 of Fig. 2, abstraction is applied to remove the *KCycles* and $KCycles^{-1}$ quantities from the configuration set. Since removing a quantity from two configurations that differ only in that quantity makes the configurations equal, abstraction may reduce the number of configurations in a set.

Our implementation of the ***Join*** operation slightly generalizes the definition given in Section 2, taken from [8, 7]. The original join operation matches on a single quantity, and the configurations in the end result have two identical entries for the matching quantities. Our implementation allows to match on more than one quantity, and the default implementation abstracts away one copy of any two matching quantities. Optionally, both copies can be kept. We use renaming to assign the quantities being matched in the two configuration sets to the same domains, and the remaining quantities to disjoint domains. Then, *join* reduces to an intersection. Since the quantities being compared are mapped to the same domains, the set intersection will find exactly those pairs from the two sets where these quantities match, keeping one copy of the value in the end result. If desirable, an extra copy of matching quantity values can be added.

The ***producer-consumer*** operation uses a monotonically decreasing function that needs the actual values of quantities to relate the producer quantity to the consumer quantity. Since we map the original quantity values to consecutive integers to store them into BDDs, special measures need to be taken. Algorithm 1 shows our implementation of the producer-consumer operation. The algorithm extracts integer encodings of the producer and consumer quantities from the configuration sets and stores them in two separate sets. The loop then maps the integers to the actual values to apply the producer-consumer matching function. The integer representation of each matching pair is stored into the PC binary relation BDD. Step 3 of Fig. 2 shows the PC relation in our example. The last step of the algorithm constructs the product of the configuration sets, restricting it by the PC BDD (a constraint/intersection operation, step 4 in Fig. 2).

3.3 Symbolic Minimization

It remains to discuss a *symbolic minimization* algorithm. The minimization operation extracts the Pareto configurations from a particular configuration set. The Pareto calculator of [7] implements the well-known quadratic Simple Cull algorithm, which looks at the configurations one by one and maintains a set of Pareto points among the points observed so far. The calculator also implements a Divide-and-Conquer algorithm that splits the set of configurations into two halves, minimizes these sets separately and merges the results. This algorithm is known to have optimal complexity, but it has a high overhead because of the complex recursion. The Simple Cull algorithm therefore often outperforms the Divide-and-Conquer algorithm. In this section, we propose a new symbolic approach to find the Pareto points of a configuration set.

Let $\mathcal{C}$ be a configuration set of space $\mathcal{S} = Q_1 \times Q_2 \times \ldots \times Q_n$. Set $\min(\mathcal{C}) \subseteq \mathcal{S}$ is the set of Pareto points of $\mathcal{C}$ that we want to determine. From the definitions given in Sec. 2, we can make the following derivation, explained in more detail below.

$$
\begin{aligned}
\min(\mathcal{C}) &= \{\bar{c} \in \mathcal{C} \mid \neg(\exists \bar{c}' \in \mathcal{C} : \bar{c}' \neq \bar{c} \wedge \bar{c}' \preceq_{\mathcal{S}} \bar{c})\} && (1)\\
&= \mathcal{C} \backslash \{\bar{c} \in \mathcal{C} \mid (\exists \bar{c}' \in \mathcal{C} : \bar{c}' \neq \bar{c} \wedge \bar{c}' \preceq_{\mathcal{S}} \bar{c})\} && (2)\\
&= \mathcal{C} \backslash \{\bar{c} \in \mathcal{S} \mid (\exists \bar{c}' \in \mathcal{S} : c' \in \mathcal{C} \wedge \bar{c}' \neq \bar{c} \wedge \bar{c}' \preceq_{\mathcal{S}} \bar{c})\} && (3)\\
&= \mathcal{C} \backslash \{\bar{c} \in \mathcal{S} \mid (\exists \bar{c}' \in \mathcal{S} : c' \in \mathcal{C} \wedge \bar{c}' \neq \bar{c} \wedge && (4)\\
&\qquad (\forall k, 1 \leq k \leq n : \bar{c}'(Q_k) \preceq_{Q_k} \bar{c}(Q_k)))\}
\end{aligned}
$$

The first equality in the derivation simply applies the definition of minimization, writing strict dominance in terms of dominance. The second step states that the Pareto points can be obtained from $\mathcal{C}$ by removing all dominated configurations. The third step shows that $\bar{c}$ and $\bar{c}'$ can be taken from the entire configuration space $\mathcal{S}$ when moving the requirement that dominating configurations should be from $\mathcal{C}$ into the predicate of the existential quantification. The final step shows that the dominance relation on the space can be computed from the dominance relations of the quantities via a universal quantification. Since all operations in these equations are Boolean operations that are straightforwardly implemented on BDDs, the equations provide the basis for a symbolic minimization algorithm, given as Algorithm 2.

The algorithm takes as input a configuration set C and its space S. It returns $\min(C)$. In line 2, the dominance relation $\preceq_S$ is computed. According to Eqn. (4), this can be done via simple and operations on the dominance relations of the quantities in S, which are computed separately from the specified partial orders. Since dominance relations on quantities and spaces may be reusable for other operations, one can trade off processing effort with memory requirements by storing dominance relations for later use.

In line 3, the identity relation on S is computed, which is implemented using a standard equality function in CUDD. As before, this identity relation can be stored for later reuse. Line 4 removes the identity from the dominance relation, yielding the strict dominance used in the definition of the Pareto minimal set (the '$\bar{c}' \neq \bar{c} \wedge \bar{c}' \preceq_{\mathcal{S}} \bar{c}$' part in the above

equations). Fig. 2.h shows the strict dominance relation on the 2-d $PSNR^{-1}$ and *Power* space in our running example.

The computed relation is the strict dominance relation on the entire configuration space. We need to restrict this relation to the configuration set. By taking a self-product of the configuration set ($C^2 = C \times C$, line 5, step 6 in Fig. 2.), renaming the domain of the result to the domain of the dominance relation (line 6), and taking the intersection with the strict dominance relation on S (line 7 in Alg. 2, step 7 in Fig. 2), the strict dominance relation on C is obtained.

The strict dominance relation on C is used to initialize variable *dominatedset*. The last step in our minimization algorithm, based on Eqn. (2) above, is to remove the set of *strictly dominated* configurations from the set of configurations C (where the set of strictly dominated configurations is defined by the second operand of the setminus ($\setminus$) operation in Eqn. (2)). Observe that the top half of the BDD representing a strict dominance relation contains configurations that strictly dominate some other configuration, whereas the bottom half contains precisely all strictly dominated configurations. In the example of Fig. 2.h, domain bits 0 1 6 7 represent the strictly dominating configurations and domain bits 10 11 16 17 represent strictly dominated configurations. Note that these two sets are not necessarily disjoint. Some configurations can be in both sets, so the dominating configurations are not necessarily all Pareto points. However, as shown by Eqn. (2), the set of Pareto points can be obtained by removing strictly dominated configurations from the configuration set. Algorithm 2 abstracts away all quantities corresponding to the top part of the strict-dominance BDD (lines 8-10, step 8 in Fig. 2). Note that these domains are simply those of the configuration space. It then replaces the domain of the resulting set of strictly dominated points with the domain of the configuration space (line 11). Finally, the strictly dominated configurations are removed from the main configuration set, and the algorithm returns the result (line 12, step 9 in Fig. 2).

4. MEMORY USAGE AND COMPLEXITY

An important property of BDD implementations is that some operations can be performed in constant time and most of the operations have time complexity proportional to the sizes of the BDDs being operated on [4, 5], and hence are quite efficient as long as the BDDs do not grow too large.

Interestingly, configuration sets that contain only a few or most of the configurations from a space, can typically be represented efficiently. When BDDs of configuration sets are small, also relations on those sets can typically be represented compactly. In general, the BDD size depends heavily on the number of configurations and the number of values per quantity. Our experiments indicate that the number of nodes in the BDD representation of a configuration set is usually less than the number of configurations for large sets. Furthermore, the memory required for a BDD node is quite small compared to the memory needed for an explicit representation of a configuration. So the SPaC tool usually works more efficiently, especially on large sets, in terms of memory usage when compared to a non-symbolic implementation. The next section gives some numerical results.

We use standard BDD operations to implement symbolic Pareto operations. Hence, most Pareto operations inherit time complexity from BDD operations. Berghammer et al. [3] gives a complete description of the complexity of the BDD operations. The complexity is dominated by the size of the BDD and the number of domain bits needed, which in the current context depend on the number of configurations in configuration sets and the number of dimensions (quantities) of those sets. The two exceptions are the *producer-consumer* and *minimization* operations. The producer-consumer operation uses, in addition to standard BDD operations, *decoding* to convert a BDD to integer values, and *mapping* to map integers to the actual values. The time for decoding and mapping depends heavily on the number of actual values in the producer and consumer quantities. This results in a time complexity proportional to the number of actual values of the producer and consumer quantities. The time complexity of minimization is dominated by the required *encoding* of the dominance relation on the configuration space at hand, which means that the worst case time complexity depends on the number of configurations in the space, which in turn depends on the number of values in all of the constituent quantities.

5. EXPERIMENTAL RESULTS

To compare SPaC to a non-symbolic implementation, as described in [7], we conducted experiments on randomly generated configuration sets. We applied operations on 2-d and 4-d sets of various sizes, considering ten randomly generated sets for each size. Table 1 shows the average run-times in seconds on a 2 GHz laptop with 2 GB main memory, running Windows XP. The **S** columns report results for SPaC and the **NS** columns for the non-symbolic Pareto calculator.

The third column of Table 1 reports the time needed to encode a configuration space into BDDs, which includes the mapping of quantity values to integers. The fourth column reports the decoding time needed to convert a BDD back to the explicit representation. It turns out that encoding and decoding are among the most time consuming operations in SPaC. However, encoding and decoding are one-time operations (see Fig. 2) and the required time can be amortized over all algebraic operations performed in between.

The fifth column reports the memory required for representing a configuration set, before applying any operation, showing that SPaC is efficient in memory. The reported numbers include the memory needed for storing the mapping of the original quantity values to integers.

The last six columns report execution times for various Pareto algebra operations. For binary operations, both the operands are of the reported size. The non-symbolic product operation yields a set with a size proportional to the size of the operands, which is exponential in the number of configuration sets of which the product is computed. Hence, it cannot always be completed because the memory is exceeded (reported as EM in Table 1). The most time consuming Pareto algebra operations in SPaC are *minimization* and *producer-consumer*. The minimization times include the time needed to encode dominance relations on quantities. This time is not needed in the non-symbolic case when the dominance relation is inherited from the order in built-in types like the integers or reals (which is the case for all our experiments). This explains why non-symbolic minimization requires less time than SPaC in our experiments. Note, however, that in algebraic computations involving multiple operations, it may sometimes be possible to reuse the encoded dominance relations. The times reported for the producer-consumer operation include the required decoding, which is

Table 1: Experimental results (Execution times in seconds; memory in MB)

# Confs	# Dims	Enc	Dec	Mem		Alternative		Abstract		Product		Join		ProdCons		Min	
		S	S	NS	S	NS	S	NS	S	NS	S	NS	S	NS	S	NS	S
100	2	0.01	0.01	2	0.3	0.03	0.01	0.03	0.01	1.2	0.01	0.2	0.01	1.9	4	0.03	0.06
100	4	0.03	0.01	9	0.8	0.06	0.02	0.05	0.01	1.8	0.01	0.3	0.01	1.4	3	0.04	0.07
1000	2	1	0.1	7	5	0.1	0.03	0.1	0.01	105	0.03	0.4	0.02	14	19	0.07	1
1000	4	2	0.4	15	23	0.3	0.04	0.2	0.01	182	0.04	1	0.03	16	34	0.1	2
10000	2	7	0.5	11	4	0.5	0.04	0.4	0.01	EM	0.09	5.5	0.07	14	20	0.5	17
10000	4	9	10	44	12	0.7	0.06	0.9	0.07	EM	0.2	6.6	0.12	16	33	1	71
100000	2	67	9	60	8	4.5	0.11	2	0.02	EM	0.17	78	0.1	18	20	3	94
100000	4	94	119	150	20	9.1	5	7.1	0.7	EM	0.7	96	0.5	20	33	15	200

why the non-symbolic implementation slightly outperforms the symbolic one.

Table 1 does not report results on the constraint operation. The symbolic and non-symbolic implementation of this operation cannot be compared directly. SPaC implements the constraint operation in its generic form as a set intersection, whereas the non-symbolic implementation typically applies some specific predicate to a given configuration set.

The experimental results indicate that the relative performance of the symbolic and non-symbolic implementations will depend on the application scenario. Consider for example the MPEG-4 streaming scenario of [8], which elaborates the scenario illustrated in Fig. 1; an MPEG-4 video stream is streamed from a media center over a wireless connection to a handheld device. In the compositional trade-off computation, two producer-consumer operations and three minimization operations are performed. Hence, the non-symbolic implementation is expected to work faster than the symbolic one in this case. Another interesting application for Pareto algebra is the multi-dimensional multiple-choice knapsack problem (MMKP) of [2]. MMKP extends the classical 0-1 knapsack problem. Items are divided into groups and each item has a value and a multi-dimensional resource cost. We need to choose exactly one item per group, maximizing the total value of the selected items, without exceeding the resource constraints in any dimension. MMKP can be solved straightforwardly using Pareto algebra. This solution requires many product operations, no producer-consumer operations and only one minimization. Therefore, we expect that the symbolic implementation may perform well.

It seems worthwhile to consider a hybrid implementation of Pareto algebra that uses a combination of symbolic and non-symbolic implementations of the operations. Some operations are faster symbolically and others non-symbolically, and the symbolic and non-symbolic implementations of constraint operations may complement each other well. Furthermore, the symbolic implementation is in general more memory efficient, which might be particularly relevant for operations that are expensive in memory, such as explicit free product operations.

We consider the development of a hybrid Pareto algebra implementation and an elaborate experimental evaluation of the symbolic, the non-symbolic, and the hybrid versions in relevant application scenarios such as the streaming and MMKP scenarios as an important topic for future work.

6. CONCLUSION AND FUTURE WORK

This paper presents SPaC, a symbolic implementation of Pareto algebra. It allows the compositional computation of trade-offs, applicable in, for example, multi-dimensional embedded-system design-space exploration. The experimental results show that BDDs can be applied effectively to Pareto optimization, providing very compact representations of system configurations and very efficient implementations of several operations. The results suggest that a hybrid Pareto algebra implementation combining symbolic and non-symbolic elements is worth investigating.

SPaC maps all quantity values to consecutive integers and then encodes the integer values into BDDs. In this way, floating point numbers can also be handled. In some cases however, the actual values may be needed, requiring a (partial) decoding of the BDD representation. Recently, Taylor Expansion Diagrams (TEDs) [6] were introduced, which are particularly suited for supporting algebraic computations on arbitrary functions without the need for integer encoding. It is interesting to investigate the use of TEDs to further improve the efficiency of the Pareto calculator.

The Pareto calculator, including the symbolic implementation, is available at `http://www.es.ele.tue.nl/pareto/`.

7. REFERENCES

[1] P.A. Abdulla et al. Regular model checking made simple and efficient. In *CONCUR '02*, p. 116–130. Springer, 2002.

[2] M. Akbar et al. Solving the multidimensional multiple-choice knapsack problem by constructing convex hulls. *Computers and Operations Research*, 33(5):1259-1273, 2006.

[3] R. Berghammer, B. Leoniuk, U. Milanese. Implementation of relational algebra using binary decision diagrams. *ReIMICS '01*, p. 241–257, 2002.

[4] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, 1986.

[5] R.E. Bryant. Symbolic boolean manipulation with ordered Binary-decision diagrams. ACM Computing Surveys, 24(3):293-318, 1992.

[6] M. Ciesielski et al. Taylor expansion diagrams: A canonical representation for verification of data flow designs. *IEEE Trans. Comput.*, 55(9):1188–1201, 2006.

[7] M. Geilen and T. Basten. A calculator for Pareto points. In *DATE '07*, p. 285–290, 2007.

[8] M. Geilen, T. Basten, B. Theelen, R. Otten. An algebra of Pareto points. *Fundamenta Informaticae*, 78(1):35–74, 2007.

[9] J.G. Henriksen et al. Mona: Monadic second-order logic in practice. In *TACAS '95*, p. 89–110, 1995.

[10] R. Hoes et al. Analysing QoS trade-offs in wireless sensor networks. In *MSWiM '07*, p. 60–69. ACM, 2007.

[11] J.R. Burch et al. Symbolic Model Checking: 10^{20} States and Beyond. In *LICS '90*, p. 1–33. IEEE CS, 1990.

[12] O. Lhoták and L. Hendren. Jedd: a bdd-based relational extension of java. *SIGPLAN Not.*, 39(6):158–169, 2004.

[13] F. Somenzi. Cudd: Cu decision diagram package, http://vlsi.colorado.edu/~fabio/cudd.