A Fast and Scalable Multidimensional Multiple-Choice Knapsack Heuristic

HAMID SHOJAEI, University of Wisconsin - Madison
TWAN BASTEN and MARC GEILEN, Eindhoven University of Technology
AZADEH DAVOODI, University of Wisconsin - Madison

Many combinatorial optimization problems in the embedded systems and design automation domains involve decision making in multidimensional spaces. The multidimensional multiple-choice knapsack problem (MMKP) is among the most challenging of the encountered optimization problems. MMKP problem instances appear for example in chip multiprocessor runtime resource management and in global routing of wiring in circuits. Chip multiprocessor resource management requires solving MMKP under real-time constraints, whereas global routing requires scalability of the solution approach to extremely large MMKP instances. This article presents a novel MMKP heuristic, CPH (for Compositional Pareto-algebraic Heuristic), which is a parameterized compositional heuristic based on the principles of Pareto algebra. Compositionality allows incremental computation of solutions. The parameterization allows tuning of the heuristic to the problem at hand. These aspects make CPH a very versatile heuristic. When tuning CPH for computation time, MMKP instances can be solved in real time with better results than the fastest MMKP heuristic so far. When tuning CPH for solution quality, it finds several new solutions for standard benchmarks that are not found by any existing heuristic. CPH furthermore scales to extremely large problem instances. We illustrate and evaluate the use of CPH in both chip multiprocessor resource management and in global routing.

Categories and Subject Descriptors: C.3 [**Special-Purpose and Application-Based Systems**]: Realtime and embedded systems; C.5.4 [**Computer System Implementation**]: VLSI Systems; G.1.6 [**Numerical Analysis**]: Optimization; G.2.1 [**Discrete Mathematics**]: Combinatorics

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Runtime management and design-time optimization for embedded systems, design automation, combinatorial optimization, knapsack problems, chip multiprocessors, VLSI routing

ACM Reference Format:
Shojaei, H., Basten, T., Geilen, M., and Davoodi, A. 2013. A fast and scalable multidimensional multiple-choice knapsack heuristic. ACM Trans. Des. Autom. Electron. Syst. 18, 4, Article 51 (October 2013), 32 pages.
DOI: http://dx.doi.org/10.1145/2541012.2541014

1. INTRODUCTION

Many combinatorial optimization problems can be formulated as variants of the 0-1 knapsack problem (see, e.g., [Pisinger 1995]). The basic 0-1 knapsack problem

Part of this work was supported by the EC through FP7 IST project 216224, MNEMEE.

Authors' Addresses: H. Shojaei (Corresponding author), Department of Electrical and Computer Engineering, University of Wisconsin—Madison 4621 Engineering Hall, 1415 Engineering Dr., Madison WI 53706; email: hamid.shojaei@gmail.com; T. Basten and M. Geilen, Department of Electrical Engineering, Eindhoven University of Technology PO Box 513, NL-5600 MB Eindhoven, The Netherlands, A. Davoodi, Department of Electrical and Computer Engineering, University of Wisconsin—Madison 4621 Engineering Hall, 1415 Engineering Dr., Madison WI 53706.

DOI: http://dx.doi.org/10.1145/2541012.2541014

considers N items, where each item has a value v and a resource cost r. The objective is to put items in a knapsack so that the resource capacity of the knapsack is not exceeded and the summed value of packed items is maximal. Instead of N items, N *groups* of items may be considered where one item from each group must be selected, leading to the multiple-choice knapsack problem (MCKP). The multidimensional knapsack problem (MDKP) is another variant in which a multidimensional resource cost is considered for each item and each dimension has its own capacity. The multidimensional multiple-choice knapsack problem (MMKP) [Moser et al. 1997] combines the two aforementioned variants, and is the focus of this article. MMKP considers groups of items with multidimensional resource costs.

Let N be the number of groups, with N_i items in group i ($1 \leq i \leq N$). Let R be the number of resource dimensions. The amount of available resources of type k ($1 \leq k \leq R$) is given by R_k. Suppose that item j of group i has value v_{ij} and requires resources r_{ijk} ($1 \leq k \leq R$). MMKP is then defined as follows, where the x_{ij} are binary-valued variables, denoting whether or not item j of group i is selected:

—Maximize $\sum_{1\leq i\leq N} \sum_{1\leq j\leq N_i} x_{ij} v_{ij}$
—Subject to
 $-x_{ij} \in \{0, 1\}$ $\quad 1 \leq i \leq N, 1 \leq j \leq N_i,$
 $-\sum_{1\leq j\leq N_i} x_{ij} = 1$ $\quad 1 \leq i \leq N,$
 $-\sum_{1\leq i\leq N} \sum_{1\leq j\leq N_i} x_{ij} r_{ijk} \leq R_k$ $\quad 1 \leq k \leq R.$

In the embedded and design automation domains, MMKP problem instances appear in, for example, runtime resource management [Ykman-Couvreur et al. 2006] and global routing of wiring in circuits [Shojaei et al. 2010]. Applications running on a multiprocessor platform may need a number of resources (the multidimensional aspect), where each application may have multiple configuration options (the multiple choice aspect). The goal of runtime resource management is to find a suitable configuration for the total set of active applications, which is an MMKP problem instance that needs to be solved at runtime under timing and resource constraints. In global routing, possible candidate routes for each net to be routed (the multiple choice aspect) come with different values for properties such as wire length, power, and routing resource usage (the multidimensional aspect). Combining candidate routes for all nets into a global routing solution for a given circuit is again an MMKP problem instance. In this case, the MMKP problem instances become extremely large, but they may be solved at design time.

MMKP is NP-hard. Various exact and heuristic methods for solving MMKP have been proposed. Exact methods [Khan 1998; Sbihi 2003, 2007] find an optimal solution for an MMKP instance, and can be used for small problem instances. Heuristic approaches are needed for solving larger MMKP instances and to gain efficiency. Such approaches [Moser et al. 1997; Khan 1998; Khan et al. 2002; Sbihi 2003; Hifi et al. 2004; Parra-Hernandez and Dimopoulos 2005; Akbar et al. 2006; Hifi et al. 2006; Hiremath and Hill 2007; Cherfi and Hifi 2008; Hanafi et al. 2009; Cherfi and Hifi 2009; Cherfi 2009; Crévits et al. 2012] find near-optimal solutions in acceptable computation times. Methods such as [Ykman-Couvreur et al. 2006; Ykman-Couvreur et al. 2011; Shojaei et al. 2009] aim to solve MMKP instances in real time, sacrificing some solution quality.

This article presents CPH, for Compositional Pareto-algebraic Heuristic. CPH is a heuristic framework based on our previous work [Shojaei et al. 2009] that presents a compositional heuristic for solving MMKP in real time. CPH uses the principles of Pareto algebra [Geilen et al. 2005, 2007], which is a framework for compositional calculation of Pareto-optimal solutions in multidimensional optimization problems. Compositionality in the MMKP context means that (partial) solutions for an MMKP instance can be computed by considering groups of items one at a time. This enables incremental and parallelized computation of solutions, resulting in a heuristic that is very well scalable to large MMKP instances. Scalability is important for the large-scale optimization problems that may occur in electronic design automation, such as the already mentioned global routing problem. The practical runtime of the compositional computations in the version of CPH presented in Shojaei et al. [2009] is determined by the number of partial solutions considered in each step. This provides a parameter to control the runtime of these computations and to trade off runtime and quality of the final solution. The runtime of CPH can be bound in terms of this parameter, which is particularly convenient in an online context with real-time constraints. The compositionality and parameterization are unique to CPH.

In the current article, we present a fully parameterized version of the heuristic of Shojaei et al. [2009]. We add several processing steps that provide parameters to tune CPH to the problem at hand. First, we add a new postprocessing step. Because of this, fewer partial solutions need to be maintained during the compositional steps to achieve a high solution quality. As a result, solution quality improves while at the same time computation time and memory requirements are reduced. When evaluated on the standard benchmarks of [Khan et al. 2002; MMKP benchmarks 2010] that can be solved in real time, CPH with postprocessing as presented in this article is the best real-time MMKP heuristic in terms of both solution quality and computation time. Besides the postprocessing step, we add several preprocessing steps and a step to improve the intermediate results during the compositional computation. CPH can thus also be tuned towards scalability or towards optimizing solution quality. When comparing CPH to the best-known heuristics among those that aim to find near-optimal solutions for MMKP instances, CPH yields two new best values for the 27 standard MMKP benchmarks of [Khan et al. 2002; MMKP benchmarks 2010] that are not found by any other heuristic. A variant of CPH that is amenable to parallelization finds 5 new best values. Furthermore, CPH scales to extremely large MMKP instances, as they occur in global routing.

To evaluate CPH in its intended application domain, we applied CPH in runtime resource management and global routing case studies. Chip multiprocessor (CMP) platforms are the platforms of choice for next generations of handheld devices. CMP runtime resource management can be seen as an MMKP variant [Ykman-Couvreur et al. 2006, 2011]. The CMP context is particularly challenging because MMKP instances need to be solved within the resource-constrained embedded platform, under firm real-time constraints. CPH meets these requirements. To obtain a global routing of wiring in integrated circuits, different global routing approaches can be used to generate different routing solutions. Solutions from the various approaches can be combined to build better solutions. This again is an MMKP, with typically extremely many groups and resource dimensions. Scalability of the computation is crucial. MMKP problem instances are such that none of the other methods can handle them; CPH does scale due to its parameterization and its compositional nature.

Overview. Section 2 presents related work. Section 3 introduces the CPH heuristic by means of an example; it further presents the relevant concepts and implementation aspects of Pareto algebra. Section 4 presents the Pareto-algebraic heuristic CPH in detail. Section 5 evaluates CPH on MMKP benchmarks. Sections 6 and 7 present the two case studies. Section 8 concludes.

2. RELATED WORK

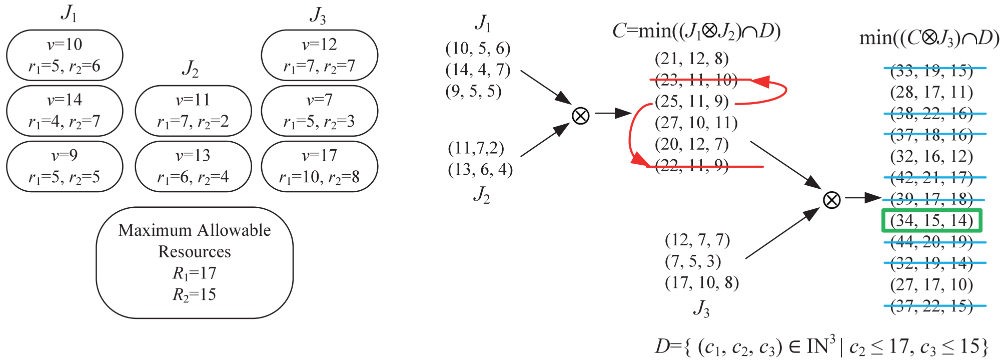
Two types of solution approaches for MMKP have been proposed in the literature. Exact solutions find an optimal solution for MMKP instances; heuristic solutions try to find a near-optimal solution, but require much less computation time than exact solutions. Heuristic solutions can be further categorized into two classes. Some of the heuristics aim for speed and are suitable for real-time applications. Others are slower, but are able to find higher-quality solutions.

Sbihi [2003, 2007] proposes an exact branch and bound algorithm for MMKP based on a best-first search strategy. The approach first finds a feasible solution for the problem. It then uses branch and bound to fix selected items during exploration, using linear programming to compute bounds during the search. This algorithm is the fastest exact algorithm, outperforming the exact method proposed by Khan [1998]. Also the approach by Khan uses a branch and bound search tree to represent the solution space and linear programming to find bounds. The order in which the decision variables are considered has an important effect on the size of the search tree. The approach depends on a separate heuristic for variable ordering.

The literature also describes heuristic methods. The first heuristics developed [Moser et al. 1997; Khan 1998; Khan et al. 2002; Hifi et al. 2004; Parra-Hernandez and Dimopoulos 2005; Akbar et al. 2006; Hiremath and Hill 2007] share the idea to project all resource dimensions of a candidate (partial) solution to a single aggregate resource, effectively reducing the multidimensional search space into a two-dimensional search space. Items are sorted with respect to a specific utility metric, which is unique for each approach. The approaches first find a feasible solution for an MMKP instance and then iterate over the sorted list of items to improve the candidate solution. Building on the thesis of Sbihi [2003], in recent years, several new MMKP heuristics have been proposed that outperform the earlier heuristics. Hifi et al. [2006] propose an algorithm which is based on reactive local search. The algorithm considers an initial solution and improves it using a fast iterative procedure. The approach uses deblocking and degrading techniques to overcome the effect of local optimality. An iterative relaxation-based heuristic (IH) in some different versions is proposed in Hanafi et al. [2009]. A column generation approach is used by Cherfi and Hifi [2008], explicitly targeting large-scale MMKP problems. The approach uses a rounding stage and then restricts the resource constraints and solves an exact instance of the restricted MMKP. Cherfi and Hifi [2009] extend this approach and propose a hybrid algorithm that combines local branching and column generation techniques to generate higher-quality solutions. Cherfi provides a final extension of this algorithm, called BLHG, in his PhD thesis [Cherfi 2009]. Recently, Crévits et al. [2012] proposed a new iterative relaxation-based heuristic for MMKP. The approach generates upper bounds for the problem using relaxation and refinement. It generates lower bounds based on a restricted version of the problem. A new semi-continuous relaxation leads to very high solution quality.

None of the approaches we have discussed is sufficiently fast for real-time applications on embedded platforms. Ykman-Couvreur et al. [2006, 2011] propose a greedy approach to find good solutions for MMKP instances for CMP runtime management. At the time, this heuristic, which we refer to as RTM (for Runtime Management), was by far the fastest among all heuristics available. For relevant MMKP instances, it finds solutions of good quality typically in the order of 1 ms on an embedded platform. In line with other approaches, RTM projects resource cost of items onto a single aggregate resource dimension, by taking a weighted sum of all the resources used. Items are then sorted according to the ratio of value over aggregate resource usage. Starting from the items with the lowest aggregate resource usage, a solution is constructed in a greedy way by traversing the sorted list while swapping items each time that such a swap increases the value and preserves feasibility. The approach assumes that the combination of items with the lowest aggregate resource usage is a feasible solution. This assumption is not always valid.

In Shojaei et al. [2009], we propose a compositional Pareto-algebraic MMKP heuristic that also targets real-time embedded applications. It uses a projection into a 2-d space



(a) (b)

Fig. 1. A running example (from a global routing problem).

in line with other heuristics. It is parameterized by the number of partial solutions considered in each compositional step. Pareto optimality is the central criterion to compare partial solutions and discard suboptimal ones. We achieve solution quality and computation times that are essentially the same as RTM. When compositionality can be exploited, for example, when applications in the CMP runtime management scenario start gradually over time, the approach is faster than RTM.

Scalability to MMKP instances with extremely many groups and extremely many dimensions has not been an explicit focus point for any of the existing heuristics. Except for our own work, none of the mentioned approaches to solve MMKP is compositional. Compositionality is particularly useful when incremental MMKP instances need to be solved and it is crucial for scalability. It allows even better scalability when parallelizing the compositional computations.

The current article extends our approach of Shojaei et al. [2009] improving both quality and speed and enabling a better tuning to the problem at hand by fully parameterizing the heuristic. We evaluate CPH for real-time performance, solution quality, and scalability, presenting among others a variant of CPH that can be parallelized.

3. PARETO ALGEBRA

This section provides an overview of Pareto algebra [Geilen et al. 2005, 2007; Geilen and Basten 2007]. We give an example illustrating how MMKP can be solved using Pareto-algebraic concepts. We then define the concepts and operations for multidimensional optimization in Pareto algebra to the extent that they are relevant for solving MMKP and are needed for a precise formalization of our approach. Finally, we discuss implementation aspects for Pareto algebra.

3.1. An Example

Figure 1 gives the example illustrating the Pareto-algebraic solution to MMKP. Figure 1(a) shows an MMKP instance with three groups with different numbers of items. Each item has a value v and a two-dimensional resource usage r_1 and r_2. Resource usage bounds are defined by R_1 and R_2. Figure 1(b) shows the essence of the Pareto-algebraic MMKP solution.

The first step in the Pareto-algebraic MMKP solution is that *groups* of *items* are represented as *sets* of *tuples*. For example, group J_2 is represented by the set $\{(11, 7, 2), (13, 6, 4)\}$, where the first element of each triple represents the value of the item, and the other two elements represent its resource usage in two dimensions.

The second step is that groups of items are pairwise, iteratively combined by means of the so-called *product-sum* operation $\otimes$. This operation takes all possible combinations of items from the two groups, summing their values and resource usage per dimension. Taking for example the first item from group J_1 with the first item of group J_2 gives $(10, 5, 6) + (11, 7, 2) = (21, 12, 8)$; $J_1 \otimes J_2$ then results in the set {(21, 12, 8), (23, 11, 10), (25, 11, 9), (27, 10, 11), (20, 12, 7), (22, 11, 9)}. The elements of $J_1 \otimes J_2$ can be seen as *partial solutions* to the MMKP instance. An important observation is that these partial solutions have the same representation as items, namely, as value-resource-usage tuples. Thus, the product-sum operation can also be applied to sets of partial solutions, implying that groups of items can be combined one-by-one. This is illustrated in Figure 1(b), where the partial solutions from $J_1 \otimes J_2$ are subsequently combined with J_3.

A third important aspect is that throughout the iterative process that combines groups of items into partial solutions, some of these partial solutions can be discarded. One reason to discard a partial solution is that it violates resource constraints. In Figure 1(b), set $D = \{(c_1, c_2, c_3) \in \mathbb{N}^3 \mid c_2 \leq 17, c_3 \leq 15\}$ captures all feasible (partial) solution candidates for the MMKP instance. Enforcing resource constraints can be done by taking the set intersection with this set. For example, $(J_1 \otimes J_2) \cap D$ enforces the resource constraints on the partial solutions in $J_1 \otimes J_2$. In this case, the result happens to be equal to $J_1 \otimes J_2$ because all partial solutions satisfy the resource constraints. Another reason to discard a (partial) solution candidate is that it is not better in any dimension than some other candidate. For example, partial solution $(25, 11, 9)$ from $J_1 \otimes J_2$ *dominates* both $(23, 11, 10)$ and $(22, 11, 9)$, because it has an equal or higher value for the same or lower resource costs than the dominated configurations. Such dominated partial solution candidates are not of interest because they can never lead to the optimal solution in the end. Removing dominated elements from a set is referred to as *Pareto minimization*, denoted by the min operation in Figure 1(b). The result of Pareto minimization is the set of so-called *Pareto points*, that is, the points (partial solutions, in this case) that are not dominated by any of the other points in the set. This is where the name Pareto algebra comes from. It defines an algebra of operations on sets of Pareto points.

As a final step in the Pareto-algebraic solution to MMKP, after iteratively combining all the groups of items into a final set of solution candidates and once more applying resource constraints and Pareto minimization, a set of *feasible solution candidates* for the MMKP instance is obtained. In Figure 1(b), the set $C \otimes J_3$ shown to the right gives twelve potential solution candidates of which eight do not satisfy the resource constraints. None of the remaining four feasible solution candidates dominates any of the other solution candidates. Among these four remaining candidates, the one with the highest value, (34, 15, 14), is *the solution* to the MMKP instance.

We may conclude that Figure 1 illustrates an exact, compositional, computation of all feasible solution candidates for an MMKP instance. Compositionality is important in this approach. It integrates groups of items one-by-one into partial solutions in a way that ultimately leads to the final solution. Each of the partial solutions kept in any of the intermediate steps is an optimal trade-off between resource usage and value, in the sense that it is not possible to get a higher value with items from the considered groups given the resource usage of a specific partial solution. The compositional computations furthermore maintain all such Pareto-optimal partial solutions. Set C in Figure 1(b), for example, provides four optimal value-resource-usage trade-offs for the combinations of items from groups J_1 and J_2. In this way, maximal flexibility is maintained for future steps in the computation. If resource constraints are loose, then the high-value, high-resource-usage partial solutions will likely contribute to the final solution; if resource constraints are tight, then the lower-value, low-resource-usage partial solutions will lead to a solution, if one exists.

The solution approach to MMKP outlined in this subsection uses the concepts of Pareto algebra, which allows compositional reasoning about multiobjective optimization problems. The sketched, exact, solution is not efficient and therefore not practical. It is however the basis for our heuristic CPH, described in the following section. To provide a solid basis, in the remainder of this section, we first summarize and formalize the basic definitions of Pareto algebra and present the most important implementation aspects concerning this algebra.

3.2. Definitions

The basic concept in Pareto algebra is that of a *quantity*. A quantity is a parameter, quality metric, or any other quantified aspect. It is represented as a set Q with a partial order $\preceq_Q$ that denotes preferred values (lower values being preferred). The subscript is dropped when clear from the context. If $\preceq_Q$ is total, the quantity is *basic*. In Figure 1(a), resource costs r_1 and r_2 are taken from finite intervals of nonnegative natural numbers that form two basic quantities with the usual total order $\leq$; value v is taken from the basic quantity of the natural numbers $\mathbb{N}$ with total order $\geq$ as the preference.

A *configuration space* $\mathcal{S}$ is the Cartesian product $Q_1 \times \cdots \times Q_n$ of a finite number of n quantities; a *configuration* $\bar{c} = (c_1, \ldots, c_n)$ is an element of such a space, where $\bar{c}(Q_k)$ may be used to denote c_k. Figure 1(a) shows three configuration sets J_1, J_2, and J_3. Groups of items in an MMKP instance can thus be captured as *configuration sets* in Pareto algebra.

The selection of items from different groups in an MMKP instance results in a combined value and resource cost that can be captured by element-wise addition on the configurations representing the items. In Pareto-algebraic terminology, such combinations of items are again configurations. These configurations provide partial solution candidates for the MMKP instance.

All possible combinations of configurations from two configuration sets (items or partial solutions) can then be captured by taking a product using the already introduced product-sum operation $\otimes$. The operation is defined as follows: $\mathcal{C}_1 \otimes \mathcal{C}_2 = \{\bar{c}_1 + \bar{c}_2 \mid \bar{c}_1 \in \mathcal{C}_1, \bar{c}_2 \in \mathcal{C}_2\}$, for configuration sets $\mathcal{C}_1$ and $\mathcal{C}_2$, with $+$ denoting the element-wise addition on configurations.

A dominance relation $\preceq \subseteq \mathcal{S}^2$ on configuration space $\mathcal{S}$ defines preference among configurations. If $\bar{c}_1, \bar{c}_2 \in \mathcal{S}$, then $\bar{c}_1 \preceq \bar{c}_2$ iff for every quantity Q_k, the k-th component 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 $\bar{c}_2$. We have already seen examples of dominance. Dominance is reflexive, that is, 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, that is, for any $\bar{c}_1, \bar{c}_2 \in \mathcal{C}$, $\bar{c}_1 \not\prec \bar{c}_2$. Figure 1(b) shows the Pareto minimal configuration set resulting from $(J_1 \otimes J_2) \cap D$. The crossed out configurations are dominated and hence not Pareto points.

A crucial observation underlying Pareto algebra 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 for compound sets. We define those operations that are relevant for this article. Note that the product-sum operation $\otimes$ is not among the operations as defined in Geilen et al. [2007]; however, it can be expressed in terms of these basic operations, as shown later in the text.

Definition 3.1 (*Pareto Algebra*). Let $\mathcal{C}$ and $\mathcal{C}_1$ be configuration sets of space $\mathcal{S}_1 = Q_1 \times Q_2 \times \cdots \times Q_m$ and $\mathcal{C}_2$ a configuration set of space $\mathcal{S}_2 = Q_{m+1} \times \cdots \times Q_{m+n}$.

(1) *Computing Pareto points, minimization*. Configuration set $\min(\mathcal{C}) \subseteq \mathcal{S}_1$ is the set of Pareto points of $\mathcal{C}$: $\min(\mathcal{C}) = \{\bar{c} \in \mathcal{C} \mid \neg(\exists \bar{c}' \in \mathcal{C} : \bar{c}' \prec \bar{c})\}$. For example, in Figure 1(b), $\min((J_1 \otimes J_2) \cap D) = \min(J_1 \otimes J_2) = \{(21, 12, 8), (25, 11, 9), (27, 10, 11), (20, 12, 7)\}$.
(2) *Free, Cartesian product*. Configuration set $\mathcal{C}_1 \times \mathcal{C}_2 \subseteq \mathcal{S}_1 \times \mathcal{S}_2$ is the (free, Cartesian) product of $\mathcal{C}_1$ and $\mathcal{C}_2$. It takes all combinations of configurations from the two configuration sets. The product of configuration sets J_1 and J_2 from Figure 1(b) gives a configuration set with six 6-tuples: $J_1 \times J_2 = \{(10, 5, 6, 11, 7, 2), (10, 5, 6, 13, 6, 4), (14, 4, 7, 11, 7, 2), (14, 4, 7, 13, 6, 4), (9, 5, 5, 11, 7, 2), (9, 5, 5, 13, 6, 4)\}$.
(3) *Constraint*. Constraining a set of configurations can be captured as set intersection. Configuration set $\mathcal{C} \cap \mathcal{C}_1 \subseteq \mathcal{S}_1$ is the $\mathcal{C}_1$-constraint of $\mathcal{C}$. For example, the final set $\{(28, 17, 11), (32, 16, 12), (34, 15, 14), (27, 17, 10)\}$ in Figure 1(b) is obtained by applying the set of resource constraints $D = \{(c_1, c_2, c_3) \in \mathbb{N}^3 \mid c_2 \leq 17, c_3 \leq 15\}$ to the configuration set $C \otimes J_3$. The crossed out configurations do not satisfy the constraints specified by this set.
(4) *Derived quantity*. Let Q be a quantity and $f : \mathcal{S}_1 \to Q$ a function that is monotone (or order-preserving) with respect to orders $\preceq_{\mathcal{S}_1}$ and $\preceq_Q$. From configuration set $\mathcal{C}$, we can then derive a new configuration set $DQ(\mathcal{C}, f) = \{(c_1, \ldots, c_m, f(c_1, \ldots, c_m)) \mid (c_1, \ldots, c_m) \in \mathcal{C}\}$. For example, we can add the aggregate resource usage of an item, defined as the sum of all its resource usage, to a configuration in our example using standard addition on the second and third element in each configuration as the function f. Assume that $+_{i,j}$ is the function on tuples that adds the values of elements i and j of the tuple that it is applied to. Then, $DQ(J_2, +_{2,3}) = \{(11, 7, 2, +_{2,3}(11, 7, 2)), (13, 6, 4, +_{2,3}(13, 6, 4))\} = \{(11, 7, 2, 7+2), (13, 6, 4, 6+4)\} = \{(11, 7, 2, 9), (13, 6, 4, 10)\}$,
(5) *Abstraction*. Let $k \in \{1, \ldots, m\}$. Configuration set $\mathcal{C} \downarrow k = \{(c_1, \ldots, c_{k-1}, c_{k+1}, \ldots, c_m) \mid (c_1, \ldots, c_m) \in \mathcal{C}\} \subseteq Q_1 \times \cdots \times Q_{k-1} \times Q_{k+1} \times \cdots \times Q_m$ is the k-abstraction of $\mathcal{C}$. It removes dimension k from all the configurations. For a subset of indices $K \subseteq \{1, \ldots, m\}$, $\mathcal{C} \downarrow K$ denotes the simultaneous abstraction of all dimensions in K. Continuing the example of the previous item, we can remove the two original resource dimensions from the 4-d configurations obtained after adding aggregate resource cost: $DQ(J_2, +_{2,3}) \downarrow \{2, 3\} = \{(11, 9), (13, 10)\}$. Note that this combination of a derived quantity and abstraction implements the projection into a 2-d value vs. resource-cost space used by many MMKP heuristics, as mentioned in the related work section.

A key objective of Pareto algebra is to support *compositional reasoning* with sets of Pareto points. This means that Pareto minimization may be performed at any intermediate step, as illustrated in Figure 1(b) and explained in the previous subsection. For this to work, it should not be possible that configurations that are dominated in intermediate steps could potentially be optimal later on, because dominated configurations are discarded at intermediate steps. Essentially, this is what is captured by the mathematical concept of monotonicity. For this reason, for example, the functions underlying derived quantities need to be monotone. Consider for example configurations $(25, 11, 9)$ and $(23, 11, 10)$ from $J_1 \otimes J_2$ in Figure 1(b). We have already seen that the former dominates the latter, implying that $(23, 11, 10)$ is discarded as a partial solution. Consider one would allow adding nonmonotone "derived" quantities, for instance, a fourth value 2 resp. 1 from the basic quantity of natural numbers with order $\leq$ indicating that smaller values are preferred, leading to $(25, 11, 9, 2)$ and $(23, 11, 10, 1)$. Then, $(23, 11, 10, 1)$ is no longer dominated by $(25, 11, 9, 2)$. The order of Pareto minimization and adding derived quantities then influences the final result, which is undesirable for compositional reasoning. The basic idea is that derived quantities do not add any essential, new information, but only derived information. Standard functions like (weighted) addition, subtraction, minimization, maximization, etc. are all monotone. Information that cannot be derived in the sense of derived quantities should be taken into account from the beginning. For more details on the role of monotonicity in Pareto algebra, the reader is referred to Geilen et al. [2007].

The earlier defined product-sum operation $\otimes$ can be defined in terms of the operations of Definition 3.1. For the example of Figure 1(b) with 3-d configurations, $\otimes$ can be implemented by first taking a free product, then adding three derived quantities corresponding to summed values and summed resource costs for the two resource dimensions, and finally abstracting away the original values and resource costs. For example, $J_1 \otimes J_2 = DQ(DQ(DQ(J_1 \times J_2, +_{1,4}), +_{2,5}), +_{3,6}) \downarrow \{1, 2, 3, 4, 5, 6\}$. Note that strictly following this definition in terms of basic Pareto algebra operations does not yield the most efficient implementation. Directly implementing the definition of $\otimes$ given before Definition 3.1 is more efficient. We discuss relevant implementation aspects for Pareto algebra operations next.

3.3. Implementation

Pareto minimization is the crucial operation in Pareto algebra. This problem is also known as maximal vector computation in computational geometry [Preparata and Shamos 1985] or skyline computation in the context of relational databases [Borzsonyi et al. 2001]. Many algorithms have been developed to tackle this problem efficiently; see Godfrey et al. [2005] for an overview. Kung and Bentley [Kung et al. 1975; Bentley 1980] developed a divide-and-conquer approach, that is still the algorithm with the best worst-case computational complexity, $\mathcal{O}(N(\log N)^{d-1})$ [Kung et al. 1975; Yukish 2004], where N is the number of points in the set being minimized and d is the dimensionality of these points. A very simple algorithm for Pareto minimization is the Simple Cull algorithm [Preparata and Shamos 1985; Yukish 2004], also known as a block-nested loop algorithm [Borzsonyi et al. 2001]. Simple Cull is a nested loop that essentially compares configurations in the set of points to be minimized in a pairwise fashion, leading to an $\mathcal{O}(N^2)$ worst-case complexity. The observed runtime complexity is often lower though, and the cost of implementing Simple Cull is much lower than the cost of implementing the recursive divide-and-conquer approach.

In Geilen and Basten [2007], the divide-and-conquer approach was adapted to cope with partially ordered quantities and compared to Simple Cull. It turns out that for sets of up to several thousands of points, Simple Cull is in practice more efficient than the divide-and-conquer approach. Because of the compositional nature of the reasoning in Pareto algebra, which strives to keep configuration sets representing intermediate results small, Simple Cull is in practice often the most efficient way to compute Pareto points in Pareto-algebraic computations. An additional advantage of Simple Cull in the context of Pareto algebra is that other operations can easily be integrated into the Simple Cull algorithm. This can be illustrated by means of the integrated computation of the product-sum operation, enforcing resource bounds, and Pareto minimization, as it occurs in Figure 1(b). Procedure *ProductSum-Bound-Min* gives the pseudocode for this integrated computation.

Procedure *ProductSum-Bound-Min* is a three-deep nested loop that takes two configuration sets (groups of items or sets of partial solutions) and a vector of (resource) bounds as input. It outputs a Pareto-minimal result set of feasible, compound configurations, that is, partial solutions to the MMKP instance that consider all the groups of items that contributed to the two input configurations sets. Line 1 initializes the result set. Lines 2–4 create, and iterate over, all compound configurations, realizing the product-sum operation. Line 5 then checks feasibility of newly created configurations, enforcing the resource constraints. Lines 6–10 check whether the new compound

PROCEDURE 1: *ProductSum-Bound-Min*: Combine two configuration sets

Input: $\mathcal{C}_1, \mathcal{C}_2$ configuration sets, $\bar{b}$ a vector of bounds
Output: *result* a minimized, compound configuration set, with only feasible configurations

```
     // Initialize the result to the empty set
 1:  $result = \emptyset$;
     // Iterate over all combinations of configurations
 2:  for all $\bar{c}_i \in \mathcal{C}_1$ do
 3:      for all $\bar{c}_j \in \mathcal{C}_2$ do
             // Create a compound configuration
 4:          $\bar{c} = \bar{c}_i + \bar{c}_j$;
             // Continue only if the compound configuration is feasible
 5:          if $feasible(\bar{c}, \bar{b})$ then
                 // Minimize the result set
                 // Maintain an attribute to check whether current configuration $\bar{c}$ is dominated
 6:              $dominated =$ false;
                 // Iterate over all configurations so far in the result set
 7:              for all $\bar{c}' \in result$ do
                     // Check whether the configuration $\bar{c}'$ of the result set is dominated; if so,
                     remove it
 8:                  if $\bar{c} \prec \bar{c}'$ then $result = result \setminus \{c'\}$;
                     // Check whether configuration $\bar{c}$ is itself dominated; if so, stop further
                     minimization
 9:                  else if $\bar{c}' \prec \bar{c}$ then $dominated =$ true; break;
                 // Add configuration $\bar{c}$ to the result set if not dominated
10:              if not dominated then $result = result \cup \{\bar{c}\}$;
11:  return result;
```

configuration dominates any compound configurations in the result set that so far were Pareto points, or whether the new configuration is itself dominated. After completion of the loops, Line 11 returns the result.

Note that Procedure *ProductSum-Bound-Min* follows the structure of the two-deep nested loop of Simple Cull; in Simple Cull, the outer loop iterates over all points in the set to be optimized and the inner loop iterates over the result set being maintained. In *ProductSum-Bound-Min*, the outer loop of Simple Cull is turned into the nested loop of Lines 2 and 3 to create all the configurations that need to be considered for the result. The size of this configuration set is $N = |\mathcal{C}_1 \times \mathcal{C}_2|$; the worst-case size of the result set is also N (when all compound configurations are Pareto points). The worst-case complexity of *ProductSum-Bound-Min* is therefore $\mathcal{O}(N^2)$, that is, quadratic in terms of the size of the set to be minimized, which is exactly the complexity of Simple Cull.

Procedure *ProductSum-Bound-Min* is the core of our CPH heuristic as presented next. The low-overhead, Simple Cull based, integrated implementation of the various operations on the configuration sets representing groups of items and partial solutions is more efficient than an explicit implementation of each of the operations; it is therefore chosen in the implementation of CPH.

4. A PARAMETERIZED COMPOSITIONAL PARETO-ALGEBRAIC HEURISTIC

This section introduces our MMKP heuristic CPH. It follows the approach of Figure 1. We first explain the basic algorithm and its parameters. We then provide more detail about three of these parameters, namely the selection of partial solutions maintained in each compositional step, the compositional improvement step, and the postprocessing approach. Finally, we discuss the complexity of the approach and we present an approach to parallelize the heuristic.

4.1. The Basic Algorithm

CPH takes as input an ordered set S of configuration sets and a vector of resource bounds Rb, which together define the MMKP instance, and a parameter L that limits the number of partial solutions considered in each step of the algorithm. L allows to control the runtime (and space) complexity, as explained in the following. The input configuration sets may represent groups of items or sets of compound items obtained from adding value and resource costs of items from any number of different groups. The result of the algorithm is a configuration set of the same type. All configurations have one dimension corresponding to the value and R dimensions corresponding to the R resources. Conceptually, CPH follows the exact compositional approach outlined in Section 3. In each compositional step it essentially takes a product (the $\otimes$ product-sum operation) and then removes infeasible and dominated configurations. We avoid the explicit enumeration of products by integrating the computation of the product-sum operation with the feasibility and dominance checks (through Procedure *ProductSum-Bound-Min*).

PROCEDURE 2: CPH: A Pareto-algebraic heuristic for MMKP

Input: S a vector of configuration sets, Rb resource bounds, L maximum number of partial solutions
Output: *result*, a configuration set, of which any max value configuration is a solution to the MMKP instance

```
Step 1. // Keep only Pareto points
    For all $\mathcal{C}_i \in S$ do min($\mathcal{C}_i$);
Step 2. // Perform any desirable preprocessing
    $PreProcess(S)$;
Step 3. // Add aggregate resource usage to all configurations
    For all $\mathcal{C}_i \in S$ do $DQ(\mathcal{C}_i, aru)$;
Step 4. // Optionally, compute an upper bound on the value
    $vb = SolveLP(S, Rb)$;
Step 5. // Initialize the set of partial solutions $\mathcal{C}_{ps}$
    $\mathcal{C}_{ps} = S[1]; S = S - S[1]$;
Step 6. // The compositional computations
    For all $\mathcal{C}_i \in S$ do
        // Select (at most) $L$ partial solutions
        $\mathcal{C}_{ps} = SelectPS(\mathcal{C}_{ps}, L)$;
        // Optionally, improve partial solutions
        $Improve(\mathcal{C}_{ps})$;
        // Select (at most) $L$ configurations from $\mathcal{C}_i$
        $\mathcal{C}_i = SelectPS(\mathcal{C}_i, L)$;
        // Combine partial solutions with configurations from $\mathcal{C}_i$
        $\mathcal{C}_{ps} = ProductSum\text{-}Bound\text{-}Min(\mathcal{C}_{ps}, \mathcal{C}_i, vb; Rb)$;
Step 7. // Perform any desirable postprocessing
    $result = PostProcessing(\mathcal{C}_{ps})$;
Step 8. Return $result$;
```

Procedure CPH provides pseudo-code. CPH gets as input an N-sized vector of configuration sets S, an R-sized vector of resource bounds Rb, and the value for parameter L.

As a first step, Step 1 finds Pareto points in each configuration set. Dominated configurations cannot contribute to an optimal solution of the MMKP instance. A for-loop applies the Pareto-algebraic min operation of Definition 3.1(1).

In Step 2, some form of preprocessing may be performed. For groups with different numbers of items, for example, it is beneficial to consider groups with fewer items first. The fewer items the two configuration sets being combined in a compositional step have, the faster is the composition and the more accurate is the result after reducing the set of partial solutions being maintained to L partial solutions. Therefore, when groups differ in the number of items, the preprocessing sorts the groups on their cardinality, in ascending order.

Step 3 adds aggregate resource costs to each configuration, via a derived-quantity operation DQ of Definition 3.1(4) with aggregate resource cost function aru. Some operations in CPH may be done in a 2-d value-aggregate resource usage space. Different ways of aggregation may be chosen for different purposes. We discuss several alternatives in Section 4.2. Adding aggregate resource usage costs to configurations makes these configurations $(R + 2)$-dimensional. In an implementation, the $(R + 1)$-d value-resource usage subspace and the 2-d value-aggregate resource usage subspace can thus efficiently be stored in one data structure.

Step 4 is optional. If enabled, it computes an upper bound on the value for the solution to the MMKP instance. The MMKP instance, which is an integer linear program, is relaxed to a linear program (LP). This is done by allowing the decision variables x_{ij} to take any real value in the [0, 1] interval and by using '=' instead of '$\leq$' in the set of resource constraints in the MMKP formulation (the third set of constraints). A solution to this LP consists of arbitrary fractions of items of each group that maximize the total value while fully exploiting the available resources. This yields an upper bound for the value of any solution to the original MMKP instance. This upper bound can be used to prune partial solutions in the same way as resource constraints are used (see Section 3.1): if a partial solution has a higher value than the computed upper bound, it cannot be completed to a feasible solution candidate for the MMKP instance. When targeting an embedded implementation of CPH, computing the value upper bound should preferably be done offline beforehand (if all relevant information is available) or online by means of an efficient algorithm to solve LPs. Alternatively, the value bound can simply be omitted because CPH works fine without it. For offline computation of the value bound, one can use a tool like IBM ILOG Cplex [Cplex 2012]. We experimented with Cplex version 9. The computation is very fast and the execution time is negligible for standard MMKP benchmarks. In an actual implementation, for efficiency, the value upper bound may be appended to the resource bounds as a derived quantity.

Step 5 does the initialization for the compositional computations. During the compositional steps, a configuration set $\mathcal{C}_{ps}$ of partial solutions is maintained. This set is initialized with the first configuration set in vector S, $S[1]$, which is subsequently removed from the vector of configurations.

The for-loop in Step 6 implements the compositional steps. It iterates over the list of remaining configuration sets. In each iteration, it performs four actions. First, Procedure *SelectPS* selects (at most) L partial solutions from configuration set $\mathcal{C}_{ps}$. The details of *SelectPS* are given in Section 4.2. Second, an optional improvement step is performed (Procedure *Improve*, see Section 4.3). Third, (at most) L configurations from configuration set $\mathcal{C}_i$ are selected. Fourth, the selected partial solutions and new configurations from $\mathcal{C}_i$ are combined, through procedure *ProductSum-Bound-Min* that has already been discussed in Section 3.3.

Procedure *ProductSum-Bound-Min* is called with $\mathcal{C}_{ps}$ and $\mathcal{C}_i$ as input configuration sets, and, assuming a value bound is available, with the concatenation $vb; Rb$ of the value bound and the resource bounds. *ProductSum-Bound-Min* essentially computes $\min((\mathcal{C}_{ps} \otimes \mathcal{C}_i) \cap D)$ with $\mathcal{D} = \{(v, r_1, \ldots, r_R, r) \in \mathcal{C} \mid v \leq vb \wedge \forall_{1 \leq k \leq R}\, r_k \leq Rb[k]\}$. The $\otimes$ product-sum operation creates all compound configurations. Note that configurations are $(R + 2)$-d and include aggregate resource cost r. The product-sum operation sums aggregate resource costs of constituent configurations to compute aggregate resource costs for compound configurations. The set intersection defines a constraint operation (Definition 3.1(3)) that allows only configurations that respect both the value bound and all the resource capacity constraints. Dominance (the $\prec$-operation in Lines 8 and 9 of *ProductSum-Bound-Min* and the min operation in the given expression) is not computed in the $(R + 2)$-d space, but either in the original $(R + 1)$-d space or in the 2-d value-aggregate resource usage space. The latter is faster and may be employed in online embedded applications; the former is more accurate and may be used when targeting solution quality.

When CPH completes Step 6, $\mathcal{C}_{ps}$ contains feasible Pareto-optimal configurations that are all solution candidates for the considered MMKP instance. Any maximal-value configuration among these candidates is typically already a good solution for the MMKP instance. Step 7 of CPH nevertheless applies a postprocessing algorithm that tries to improve the result of Step 6. The specific postprocessing is again a parameter of CPH. It may be tuned to the situation at hand. Section 4.4 describes three different postprocessing methods that we applied in our CPH implementations.

Finally, Step 8 returns the result, which is a set of configurations, of which any maximal-value configuration is a solution to the MMKP instance.

4.2. Selecting Partial Solution Candidates

Whenever the number of configurations in the compositional steps of CPH, either in the set of partial solutions or in the additional group of items that is being considered, exceeds threshold L, we need to select L configurations that provide a good approximation of the configuration set at hand. This selection needs to be done efficiently, because it is a part of the compositional phase of CPH. The selection mechanism can be considered to be a parameter of CPH. Any mechanism can be used. In our implementations so far, we applied a mechanism that determines a selection of partial solutions in the 2-d value-aggregate resource usage space. Threshold L and the aggregate resource usage function *aru* are parameters of this selection mechanism.

Many MMKP heuristics use the 2-d value-aggregate resource usage space to simplify the search for solution candidates. Akbar et al. [2006] suggest the transformation of the R resource dimensions to a single dimension by taking the Euclidean distance to the origin in the resource subspace (obtained by omitting the value dimension from all configurations). Many other heuristics follow this approach. To avoid expensive square-root computations, which is particularly relevant for embedded applications, as a default, we simply take the sum of all resources: for configuration $\bar{c} = (v, r_1, \ldots, r_R)$, the aggregate resource usage $aru(\bar{c}) = \sum_{1 \leq k \leq R} r_k$. If one wants to take the scarcity of resources into account, one could take the sum of the relative resource usage $aru(\bar{c}) = \sum_{1 \leq k \leq R} r_k / R_k$; the use of scarce resources can be made more expensive by introducing a parameter $\alpha > 1$: $aru(\bar{c}) = \sum_{1 \leq k \leq R} (r_k / R_k)^{\alpha}$. Note that *aru* is only computed for input configuration sets. Procedure *ProductSum-Bound-Min* builds upon this by aggregating resource usage of compound configurations by summing the costs for constituent configurations.

Figure 2 gives an example of the selection process from one of the standard MMKP benchmarks, I2, introduced later in this article. The selection occurs in the 2-d

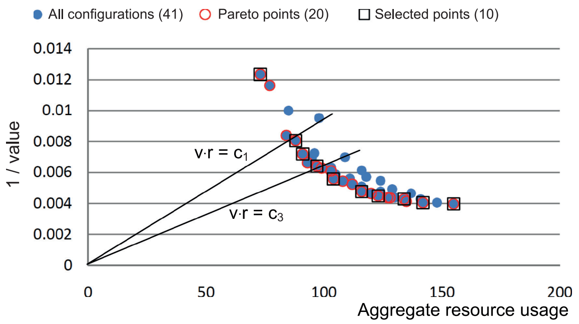


Fig. 2. Minimization and reduction in a two-dimensional configuration space (taken from one of the standard benchmarks).

value-aggregate resource space. Figure 2 assumes the default aggregate resource usage function introduced earlier and it uses the inverse value for illustration purposes, so that less is better in both dimensions in the figure. The example moreover assumes that Pareto minimization of configuration sets in *ProductSum-Bound-Min* is done in the 2-d value-aggregate resource usage space, so that only configurations that are Pareto points in that 2-d space are considered during selection.

Configurations in the 2-d space are characterized by their product $v \cdot r$, where v is the value and r the aggregate resource cost. High value typically comes with high resource usage and thus a high product value; low value and low resource usage give a low product value. The basic idea of our selection is to choose configurations that cover the entire product range uniformly. This provides the highest possible flexibility in selecting fitting configurations in later steps of CPH.

The sketched idea is implemented by Procedure *SelectPS*. Assume L to be at least 2. For any given constant c, $v \cdot r = c$ provides a line through the origin of the value-aggregate resource usage space, as shown in Figure 2. We select our L points, by first taking the two Pareto points $(r_{\max}, v_{\max})$ and $(r_{\min}, v_{\min})$ that result in two lines $v_{\max} \cdot r_{\max} = c_{\max}$ and $v_{\min} \cdot r_{\min} = c_{\min}$ with maximal and minimal values for the constant c. These two points cover the extreme points in the value-resource usage product range. We then divide the range $c_{\max}..c_{\min}$ in $L - 1$ equally large parts, conceptually resulting in $L - 2$ new lines through the origin of the space. For each of the $L - 2$ constants c, we take a Pareto point of which the $v \cdot r$ product is closest to c. In this way, *SelectPS* efficiently selects a good spread of Pareto points over the space.

It is important to observe that the choice for the aggregate resource usage cost *aru* in combination with the choice to do Pareto minimization in either the $(R + 1)$-d value-resource usage space or in the 2-d value-aggregate resource usage space can be tuned to the intended application of CPH. When aiming for efficiency, one should follow the previous example with Pareto minimization in 2-d space and an aggregation function that fits with the expected type of MMKP instances that need to be solved. When aiming for maximal quality, one should do Pareto minimization in the original $(R + 1)$-d value-resource usage space. By doing Pareto minimization in the $(R + 1)$-d value-resource usage space, partial solutions that are Pareto optimal because of low resource usage in any individual dimension are kept; an appropriate aggregate resource usage function then maximizes the chance that configurations with low resource usage in any dimension corresponding to a scarce resource is selected in Procedure *SelectPS*, because it pushes configurations with a high resource usage for scarce resources to the extreme of the value-resource usage product range, that is, the $v \cdot r$ range introduced earlier.

4.3. Compositional Improvement

During the compositional computations, CPH optionally applies an improvement step (Procedure *Improve*, Step 6). All the partial solutions in the set $\mathcal{C}_{ps}$ provide a good trade off between value and resource costs. Because CPH is a heuristic, the partial solutions may have room for improvement though, in the sense that a higher value may be obtained with the same or a lower resource usage. The specific improvement step is a parameter that can be tuned to the intended use of CPH. In our experiments, we applied compositional improvement only when tuning for solution quality. We implemented an improvement step that, after a given number of compositional steps, optimizes $\mathcal{C}_{ps}$ using the IBM ILOG Cplex solver (version 9) [Cplex 2012]. In this implementation of Procedure *Improve*, each configuration in $\mathcal{C}_{ps}$ is optimized independently. For each configuration, an MMKP instance is solved to optimality using Cplex. These MMKP instances are much smaller than the original MMKP instance and have tighter resource bounds; they can therefore be solved exactly without problems. Assume the last improvement step was executed K compositional steps before the current step that performs *Improve*. *Improve* then solves for each of the L configurations the MMKP instance that takes (i) the resource usage of the considered configuration in the current set $\mathcal{C}_{ps}$ as resource bounds; (ii) the result of the last execution of *Improve* and the K configuration sets $\mathcal{C}_i$ considered since then as groups. The outcome of *Improve* is a set of L improved partial solutions to the original MMKP instance, that is fed into the next compositional step of CPH.

4.4. Postprocessing

After completion of the compositional phase, CPH optionally applies a postprocessing step to further improve the results (Procedure *PostProcessing*, Step 7 in CPH). The specific postprocessing approach is again a parameter of CPH. We implemented three different methods. Procedure *GrdyPP* is a greedy approach that aims for speed and is suitable for online use in embedded applications; *OptPP* aims for solution quality, and is intended for offline use; *ImprPP* aims for a value improvement within the given resource usage. *ImprPP* is in fact the same as the compositional improvement step of the previous subsection. It provides a compromise between speed and solution quality and is intended for offline use.

Recall that all configurations in the set $\mathcal{C}_{ps}$ resulting from the compositional phase of CPH are feasible solution candidates for the MMKP instance. All these solution candidates provide a good trade off between value and resource costs, but they may have room for improvement. The highest-value solution candidate can potentially be improved because CPH is a heuristic. The other solutions may have room for improvement because, by definition of Pareto-optimality, their resource usage is lower than the resource usage of the highest-value solution candidate. It may be that such a solution candidate can be further improved than the best solution candidate so far.

The greedy postprocessing procedure *GrdyPP* tries to improve all the solution candidates in set $\mathcal{C}_{ps}$. It first sorts all configurations of the original MMKP instance that contribute to one of the solution candidates in $\mathcal{C}_{ps}$ based on value over aggregate resource-usage ratio (using the default aggregate resource usage). The reason to limit attention to configurations that contribute to one of the solutions in $\mathcal{C}_{ps}$ is that, by the fact that they appear in the found Pareto-optimal solution candidates for the MMKP instance, those configurations provide a good value-cost ratio. The algorithm then iterates once over this sorted list. For each configuration $\bar{c}$ in the list, it considers the configuration selected so far for the group to which $\bar{c}$ belongs in each of the solution candidates in set $\mathcal{C}_{ps}$. If configuration $\bar{c}$ can further improve the solution value without violating the resource constraints, the configuration selected for the group is updated to $\bar{c}$. This algorithm is very fast and it turns out from our experiments that it significantly improves the value when applying CPH in a real-time setting.

The *OptPP* postprocessing procedure uses IBM ILOG Cplex again. It is a brute-force approach that uses the highest-value solution candidate obtained through the compositional computations as an initial solution for Cplex applied to the original MMKP instance. Cplex is then run up to a given time limit (or up to completion if it completes within the time limit). As mentioned, an MMKP instance is an ILP. The Cplex documentation mentions that a good starting point for ILP solving, that is, in our case a good initial value for the decision variables $x_{i,j}$ in the MMKP instance, has a big influence on the speed of the Cplex analysis or the quality of the result if Cplex analysis is terminated early. Setting a starting point for ILP solving can be achieved via function `CPXcopymipstart` and by setting parameter `CPX_PARAM_MIPSTART` to 1. Our experiments reported in Section 5 show that the compositional phase of CPH provides a good starting point for further optimization through Cplex, in particular when targeting solution quality and when speed of the analysis is less important.

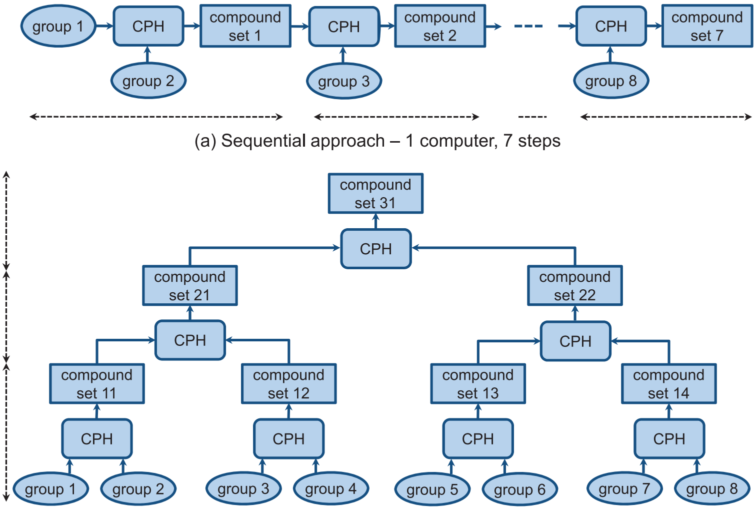
4.5. Runtime Complexity

It is interesting to consider the complexity of CPH. It turns out that, in practice, the running time of CPH is typically dominated by either the compositional part (when targeting embedded application and speed of the analysis) or the compositional improvement and postprocessing steps (when targeting offline usage and quality). We limit our analysis to the compositional part of CPH, Step 6. We exclude the compositional improvement and postprocessing steps from the analysis, because their complexity depends on the chosen algorithms. We also exclude the first part up to and including Step 5 of the algorithm. In practice, this part does not really contribute in any substantial way to the runtime.

Recall that we have N configuration sets in input vector S and assume that $N_{\max}$ is the maximum number of configurations in any of the sets in S. The complexity in each step of the compositional part depends on the number of configurations in the considered configuration sets ($\mathcal{C}_{ps}$ and $\mathcal{C}_i$) and on parameter L. In each step of the optimization, CPH first limits the size of configuration sets $\mathcal{C}_{ps}$ and $\mathcal{C}_i$ to L; the second part invokes *ProductSum-Bound-Min* on the reduced $\mathcal{C}_{ps}$ and $\mathcal{C}_i$. Observe that the size of any of the intermediate $\mathcal{C}_{ps}$ results is bounded by L^2, because in the worst case all L^2 configurations in the product computed in *ProductSum-Bound-Min* might be Pareto-optimal. The two executions of *SelectPS* in each iteration of the loop in Step 6 of CPH each require a sorting of the configuration set and a walk through the result to pick L configurations. Therefore, given that the size of $\mathcal{C}_{ps}$ may be L^2 and that the size of $\mathcal{C}_i$ is at most $N_{\max}$, the complexity of these two reductions is $O(L^2 \log L)$ and $O(N_{\max} \log N_{\max})$. Since the product computed by the two loops in Lines 2 and 3 of *ProductSum-Bound-Min* is $O(L^2)$, the complexity of *ProductSum-Bound-Min* is $O(L^4)$, as explained in Section 3.3. Since CPH does $N-1$ compositional steps, the overall complexity of the compositional part is $O(N(\max(N_{\max} \log N_{\max}, L^4)))$.

The compositional part of CPH hardly ever shows its worst-case behavior though, because the Pareto-dominance checks and the checks on the resource and value bounds prune the search space. Thus, the size of $\mathcal{C}_{ps}$ is often much closer to L than to L^2. Also, $N_{\max}$ may be smaller than L, which also reduces the practical complexity of every compositional step.

Practical runtime complexity is most important for runtime application of CPH in embedded real-time applications. In such applications, we omit the optional compositional improvement step (Procedure *Improve* in Step 6) and we apply the greedy postprocessing step *GrdyPP*. Because the size of $\mathcal{C}_{ps}$ is often smaller than its worst



(a) Sequential approach – 1 computer, 7 steps

(b) Parallel approach – 4 computers, 3 rounds of computation

Fig. 3. Sequential vs. parallel compositional computation.

case, also *GrdyPP* typically performs better than its worst case. In addition, often only a small subset of the configurations in the input actually contributes to solutions found in the compositional phase of CPH, which contributes further to the good practical performance of *GrdyPP*. Given an MMKP instance with N groups, it turns out that the computation time $T(N, L)$ of CPH with parameter L, with *GrdyPP* postprocessing, and without compositional improvement is bounded in practice as follows:

$$T(N, L) \leq C \cdot (N \cdot L^2 + L), \tag{1}$$

where C is a constant depending on the processor being used for the computations.

We evaluate the bound of Equation (1) in the experiments in the next two sections. A similar equation can be derived to bound memory usage for CPH. In the general case, it is important to observe that the parameters of CPH (L, aru, 2-d vs. $(R+1)$-d Pareto minimization, the preprocessing and optional value bound addition, the compositional improvement approach, and the postprocessing approach) can change the behavior of the heuristic from a very short runtime, limited memory usage and solutions with potentially limited quality towards longer runtimes with a higher memory usage and solutions close to the optimal solution of the MMKP instances. This makes CPH a very effective and versatile heuristic. The compositionality furthermore provides very good scalability in terms of the size of MMKP instances, as for example illustrated by the global routing case study in Section 7, and it enables parallel variants of CPH, as illustrated in the following subsection.

4.6. Parallelizing CPH

Section 4.1 presents CPH as an iterative procedure. Figure 3(a) shows a compositional execution of this iterative computation. The example assumes an MMKP instance with 8 groups. For the sake of presentation, assume that CPH is performed without pre- and postprocessing, and without compositional improvement. Every call to CPH in Figure 3(a) combines two groups. The first call combines groups 1 and 2. Every subsequent step combines the compound configuration set resulting from the previous call to CPH with the next group from the input. In this way, the compositional computation

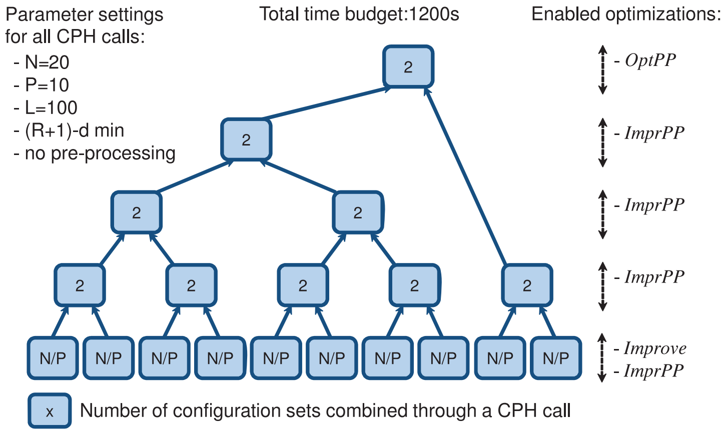


Fig. 4. pCPH with 10 processors.

takes 7 calls to solve the MMKP instance. Note that the computation essentially follows the structure of the example computation given in Figure 1.

The compositionality of CPH allows to break down an MMKP instance in any number of sub-problems. Figure 3(b) shows a parallelized computation solving an MMKP instance with 8 groups. The compositional computation follows a tree pattern. In the first round of computations, four pairs of groups are combined on four separate computers or processors. The results are combined in two parallel steps; the result of these two steps are then combined in a final step that yields the solution to the MMKP instance. The number of calls to CPH is the same as in the sequential computation, but for N groups the number of steps executed in sequence reduces from $O(N)$ to $O(\log N)$.

In general, CPH can be turned into a parallelizable heuristic, pCPH, that solves an MMKP instance on P computers (or cores) as follows. First, partition the N groups in P partitions. Second, run CPH(p, Rb, L) on each partition p; this can be done in parallel. Third, collect all the resulting compound configuration sets and divide them in as many pairs as possible. Fourth, run CPH(q, Rb, L) on each pair q. Taking pairs maximizes the potential parallelism. If the number of configuration sets is odd, one set is moved to the next round of computations. Fifth, continue the third and fourth step until only one configuration set remains. This last set gives the solution to the original MMKP instance. Figure 4 gives an example parallel computation for 10 processors, that is, $P = 10$, assuming the number of groups N is at least 20.

It is important to observe that parameter settings in pCPH can be chosen separately for each call to CPH. Figure 4 lists the parameter settings we chose in our experiments reported in the next section. We target pCPH to design-time optimization problems with large MMKP instances. In those cases, parallelism can be optimally exploited. The parameter settings are tuned towards design-time optimization. The first round of computations in pCPH solves the largest MMKP instances. Therefore, for this round, compositional improvement (Procedure *Improve*) is enabled during Step 6 of CPH; *ImprPP* is performed as postprocessing, because it provides a good compromise between speed and quality. During the other rounds of computation no compositional improvement is done, because only two configuration sets are combined each time. *ImprPP* is still performed as postprocessing though, except during the final step, when *OptPP* is performed to optimize quality. Since the individual MMKP instances that are being solved in the parallelized computations are relatively small, a large value of parameter L (that sets the number of maintained partial solutions) is possible. We used $L = 100$ in our experiments. The default setting to do no preprocessing is selected. Pareto points in *ProductSum-Bound-Min* are computed in the original $(R + 1)$-d value-resource-usage space, prioritizing quality over speed. Assuming that the total available time budget is sufficient to complete the parallelized computations up to the last step, a time budget for the overall computation can be set by setting an appropriate time budget for *OptPP*. In the experiments reported in the next section, we set the overall time budget to 1200s.

Note that pCPH is not a parallelization of CPH in the classical sense. Due to the nonuniform parameter settings for the various subcomputations performed in the various parallel rounds, pCPH performs an algorithm that differs from the algorithm implemented by CPH. Since both pCPH and CPH allow to set the overall time budget to be spent for the analysis, we envision the use of pCPH with the same total time budget as the budget that would be available to CPH. pCPH can then exploit any processors that are available, according to the scheme laid out in Figure 4. The aim is to improve solution quality and not to speed up the computation. As the results reported in the next section show, pCPH gives better results than CPH in the same time budget.

Note that cost of communication is small compared to the cost of computation. Only the Pareto points obtained for each of the subproblems need to be communicated. Computation therefore dominates the total analysis time. In the current version of pCPH the number of processors that can be used in each round of computation, halves. As future work, it would be interesting to devise different parallelization schemes of the computations and/or load balancing schemes that optimally exploit the available number of processors.

5. EVALUATION OF CPH ON MMKP BENCHMARKS

The previous section has introduced our heuristic CPH, the various parameters that allow tuning of the procedure, and a parallel variant of CPH. In this section, we experimentally investigate the various aspects of CPH on standard benchmarks. First, Section 5.1 explains the experimental setup. Second, Section 5.2 illustrates how to trade off analysis time with solution quality. Third, in Section 5.3, we evaluate CPH for real-time analysis. Finally, in Section, 5.4, CPH scalability in terms of solution quality is evaluated. The latter includes an evaluation of pCPH, the parallelizable variant of CPH.

5.1. Experimental Setup

The experiments performed in this section use standard MMKP benchmarks [MMKP benchmarks 2010] that are used in most of the work on MMKP. Table I summarizes the characteristics of the benchmarks. In line with Ykman-Couvreur et al. [2006], we divide the benchmark problems into two groups. Instances I1-I6 are considered representatives for real-time embedded applications. Heuristics targeting this class of applications should emphasize efficiency of the analysis. The second group of benchmarks, I7-I13 and INST01-INST20, provides larger test cases for quality-driven approaches to solving MMKP. As Table I shows, the optimal values for I1-I6 are known; for the other benchmarks, only an upper bound on the optimal value is known, obtained by solving the MMKP instances as a linear program.

In the standard benchmarks, all groups of a benchmark have an equal number of items; further, the number of resource dimensions is fixed and all items use resources in all resource dimensions with a uniform distribution. To cover a wider range of MMKP instances in our evaluations, we constructed additional benchmarks, described in Table II, that exhibit a more irregular structure. The number of items per group in a benchmark varies and an item does not necessarily use resources in all the resource dimensions. Also the new benchmarks are divided into small benchmarks for

Table I.
MMKP benchmarks of [MMKP benchmarks 2010], with the number of groups (N), the number of items per group i (N_i), the number of resource dimensions (R), and the optimal value (val.) or an upper bound on this value (entries marked with *).

Bench.	N	N_i	R	val.	Bench.	N	N_i	R	val.
I1	5	5	5	173	INST4	70	10	10	14478*
I2	10	5	5	364	INST5	75	10	10	17078*
I3	15	10	10	1602	INST6	75	10	10	16856*
I4	20	10	10	3597	INST7	80	10	10	16459*
I5	25	10	10	3905.7	INST8	80	10	10	17533*
I6	30	10	10	4799.3	INST9	80	10	10	17779*
I7	100	10	10	24607*	INST10	90	10	10	19337*
I8	150	10	10	36904*	INST11	90	10	10	19462*
I9	200	10	10	49193*	INST12	100	10	10	21757*
I10	250	10	10	61486*	INST13	100	30	10	21593*
I11	300	10	10	73797*	INST14	150	30	10	32887*
I12	350	10	10	86100*	INST15	180	30	10	39174*
I13	400	10	10	98448*	INST16	200	30	10	43379*
					INST17	250	30	10	54372*
INST1	50	10	10	10753*	INST18	280	20	10	60478*
INST2	50	10	10	13633*	INST19	300	20	10	64943*
INST3	60	10	10	10985*	INST20	350	20	10	75627*

Table II.
Irregular MMKP benchmarks [MMKP benchmarks 2012], with the number of groups (N), the maximum number of items per group i (max N_i), the number of resource dimensions (R), and the optimal value (val.) or an upper bound on this value (entries marked with *).

Bench.	N	max N_i	R	val.	Bench.	N	max N_i	R	val.
RTI7	10	10	10	564	INST21	100	10	10	44315*
RTI8	20	10	10	6576	INST22	100	10	20	42076*
RTI9	30	10	10	7806	INST23	100	10	30	42763*
RTI10	30	20	10	7031	INST24	100	10	40	42252*
RTI11	30	20	20	6880	INST25	100	20	10	44201*
RTI12	40	10	10	11564	INST26	100	20	20	45011*
RTI13	50	10	10	10561	INST27	200	10	10	87650*
					INST28	300	10	10	134672*
					INST29	400	10	10	179245*
					INST30	500	10	10	214257*

real-time analysis, RTI7-RT13, and large benchmarks, INST21-INST30, for evaluation of scalability and solution quality. The new benchmarks are available via [MMKP benchmarks 2012]. For RTI7-RT13, Table II gives the optimal values; for the other benchmarks an upper bound on the value was computed by relaxing the MMKP instance to a linear program.

We implemented CPH in the C programming language. Where Cplex [Cplex 2012] is used, this refers to the academic edition, version 9, with the default parameter settings. All experiments in this section were done on a PC with a 2.8Ghz Intel CPU and 12GB of memory.

5.2. Trading Off Time and Quality

In the first experiment, we show that CPH allows computation time and solution quality to be traded off, and that the computation time can be bounded beforehand. These aspects are particularly relevant when MMKP instances need to be solved

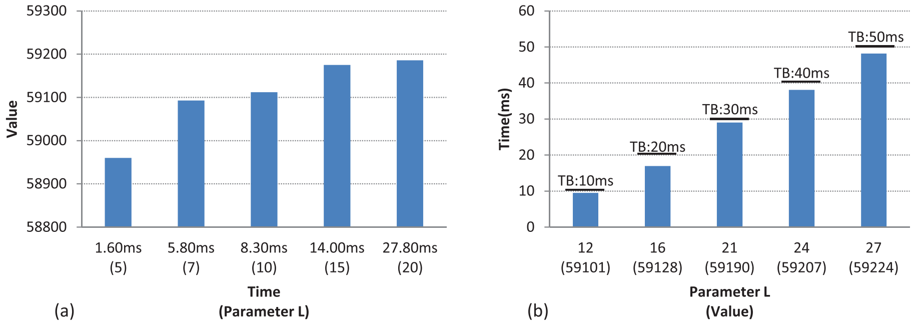


Fig. 5. Trading off value vs. computation time (a) and controlling time budgets (b), illustrated for benchmark I10.

under time constraints. We take CPH without preprocessing but with value bound, without intermediate compositional optimization steps and with greedy postprocessing (Procedure *GrdyPP*). Pareto minimization is done in the projected 2-d value-aggregate resource usage space with the default aggregation function.

Figure 5(a) illustrates the trade-off between quality of the solution and the computation time for CPH. Results are reported for benchmark I10, for different values of L, that is, the number of intermediate partial solutions kept in the compositional computations. These results are illustrative for all benchmarks, and confirm that the trade-off can be controlled by L.

An interesting use of this trade-off, is that it allows to limit the computation time to small, predefined time bounds, via Equation (1). We used 10, 20, 30, 40, and 50 ms as bounds. To compute L, we use Equation (1) with constant $C = 0.0004$. This value was measured on the used PC configuration using one of the benchmarks as a test case. Figure 5(b) again shows the results for MMKP benchmark I10. The figure shows the time bound and the actual computation time of CPH in milliseconds, the value of the computed parameter L, and the value of the obtained solution. All the results (for all benchmarks) confirm that the bound of Equation (1) is conservative.

5.3. Evaluating CPH for Real-Time Analysis

In our second set of experiments, we evaluate the real-time performance of CPH, using the mechanism to control the time budget illustrated in the previous subsection. We compare CPH to the fastest among the other MMKP heuristics to date, RTM [Ykman-Couvreur et al. 2006, 2011]. We take CPH without preprocessing and without compositional optimization, with greedy postprocessing, and with Pareto minimization in 2-d space. The default resource usage aggregation is used unless explicitly stated otherwise. The value of parameter L is determined by the available time budget. We implemented the RTM heuristic in the C programming language, in line with the implementation of CPH. For these comparative experiments we use the PC setup described earlier, for efficiency and flexibility in doing the experiments. In the next section, we evaluate CPH on an embedded platform.

Table III shows the results of our experiments with the standard benchmarks meant for real-time analysis, I1-I6. We took the time needed by RTM as the time bound for CPH. The '$t \leq$ t(RTM)' part of the table gives the results. The 'val' and 't(ms)' columns report obtained values per MMKP instance and computation times in milliseconds, respectively. The 'L' labeled column gives the value of L computed from Equation (1). As expected, execution times for CPH are, with the computed value for L, always within the bound given by RTM for the corresponding case. The solution quality of the two approaches is compared in the '%' column, which gives the relative performance of CPH

Table III.
Comparison with RTM [Ykman-Couvreur et al. 2006; Ykman-Couvreur et al. 2011] for real-time benchmarks. Computation times in ms.

Bench.	Opt. val.	RTM		CPH (t ≤ t(RTM))			
		val	t(ms)	L	val	t(ms)	%
I1	173	148	0.04	4	157	0.03	6.08
I2	364	327	0.06	4	327	0.04	0.00
I3	1602	1407	0.13	4	1410	0.11	0.21
I4	3597	3325	0.16	4	3310	0.15	−0.45
I5	3905.7	3705	0.19	4	3900	0.16	5.26
I6	4799.3	4313	0.24	4	4651	0.20	7.84
avg							**3.16**

Table IV.
Results for irregular real-time benchmarks. Time in ms.

Bench.	Opt. val	RTM		CPH (t ≤ t(RTM))							
				default aggregation				scarcity-aware aggr.			
		val	t(ms)	L	val	t(ms)	%	L	val	t(ms)	%
RTI7	564	534	0.01	4	534	0.01	0.00	4	534	0.01	0.00
RTI8	6576	$^{+}$6000	0.06	4	6000	0.06	0.00	4	6130	0.06	2.17
RTI9	7806	$^{+}$5445	0.18	4	6817	0.16	25.20	4	7367	0.17	35.30
RTI10	7031	5432	0.27	4	6089	0.26	12.09	4	6096	0.27	12.22
RTI11	6880	$^{+}$5208	0.49	4	5277	0.48	1.32	4	5449	0.45	4.63
RTI12	11564	10746	0.12	4	10746	0.11	0.00	4	10860	0.12	1.06
RTI13	10561	$^{+}$8139	0.21	4	8636	0.21	6.11	4	8097	0.20	-0.52
avg							**6.26**				**7.84**

compared to RTM. On average, we may conclude that our obtained solution quality is slightly better than that of RTM when given equal time budgets. The actual computation time for CPH is always less than the time needed by RTM. Other advantages of our approach are that the computation-time bound can be set beforehand and that solution quality can be improved by increasing L if the time budget allows. Both are not possible for RTM.

We also compared RTM and CPH on the new irregular benchmarks, RTI7-RTI13. Table IV shows the results. An important aspect is that RTM assumes that the combination of items with the lowest values in each group always provides a feasible solution. While this is true for all the original MMKP benchmarks, this is not necessarily true in general, as we for example observed in our global routing case study (Section 7). Also in the newly generated irregular benchmarks, the combination of lowest-value items leads in some cases to an infeasible solution. Hence, the original RTM approach is not directly applicable to these new benchmarks. To tackle this problem, we added an initial step to RTM in which a feasible initial solution is found and passed to RTM. If the starting solution assumed by RTM is infeasible, this initial step first sorts all the items in ascending order of aggregate resource usage; it then visits the items in the list one by one, swapping an item from the list with a selected item if this decreases the usage of at least one resource dimension without hurting any other dimension. As soon as a feasible solution is found, it is passed to the original RTM heuristic. Benchmarks marked with "+" in the RTM column of Table IV require to compute an initial solution to enable RTM.

CPH is directly applicable to the new benchmarks. It does not assume any specific information about an MMKP instance. Moreover, CPH can take advantage of irregularity in an MMKP instance to improve the quality and speed of the procedure. The runtime of CPH strongly depends on the number of items per group. Runtime improves

Table V.
The Impact of greedy postprocessing (procedure *GrdyPP*). Computation times in ms.

	CPH (t $\le$ t(RTM))						
	w/o pp			with *GrdyPP*			
Bench.	L	val	t(ms)	L	val	t(ms)	%
I1	11	154	0.03	4	157	0.03	1.95
I2	10	325	0.06	4	327	0.04	0.62
I3	9	1337	0.14	4	1410	0.11	5.46
I4	9	2870	0.17	4	3310	0.15	15.33
I5	9	3900	0.17	4	3900	0.16	0.00
I6	9	4554	0.24	4	4651	0.20	2.13
avg							**4.25**

if groups with less items are considered first. This also decreases the need to eliminate partial solutions in the *SelectPP* procedure, which improves solution quality. Therefore, for the irregular benchmarks RTI7-RTI13, we enable the preprocessing step, Step 1 of CPH. We sort the groups of items in ascending order of cardinality and start CPH with the group that has the least items. For these irregular benchmarks, we experimented with both the default aggregate resource usage function $aru(\bar{c}) = \sum_{1\le k\le R} r_k$ for $\bar{c} = (v, r_1, \ldots, r_R)$ and with the function $aru(\bar{c}) = \sum_{1\le k\le R}(r_k/R_k)^2$ that penalizes high usage of scarce resources (see Section 4.2). Given the irregularity, the latter may give better results.

Table IV compares the two instances of CPH with RTM for RTI7-RTI13. We again considered the RTM computation time as time bound for CPH. Where relevant, the time to compute an initial solution for RTM is included in the reported time for RTM. For CPH, the time for preprocessing is included. The results show that with one exception CPH provides better results than RTM. Using scarcity-aware resource usage aggregation in CPH gives in general slightly better results than the default aggregation, showing that it may be beneficial to adapt the aggregation function to the type of groups that occur in the MMKP instance.

As a final experiment in this subsection, we explore the impact of postprocessing in CPH. We make a comparison to the CPH variant of Shojaei et al. [2009] that does not include postprocessing. We again considered the computation time needed by the RTM heuristic as a bound for the runtime for CPH. Using Equation (1) and a variant of it without the $C \cdot L$ factor for CPH without postprocessing, we computed the values of L for both CPH with and without postprocessing. Table V gives the results. The '%' column shows the improvement in terms of value obtained by greedy postprocessing. CPH with postprocessing is always at least as fast as CPH without postprocessing while at the same time it always gives a value that is at least as good as without postprocessing.

The results of Tables IV and V show that the parameterized version of CPH as presented in the current article outperforms the earlier version presented in Shojaei et al. [2009].

5.4. Evaluating the Scalability of CPH for Solution Quality

In the final set of experiments in this section, we compare CPH with the state-of-the-art heuristics for solution quality [Crévits et al. 2012; Cherfi 2009]. The goal is to evaluate scalability of CPH both in terms of solution quality and runtime. We tune CPH to prioritize quality by including Cplex-based compositional improvement (Procedure *Improve* in Step 6 of CPH) and postprocessing (Procedure *OptPP*). Procedure *Improve* is applied every 10 iterations of the main compositional loop. We set parameter L to 10 to maintain 10 partial solutions during each of the compositional steps; we use the

Table VI.
Comparison with quality-based MMKP approaches.

Benchmark	Previous Results				Cplex		CPH +*OptPP*			pCPH
	ILPH	IMIPH	IIRH	BLHG	t = 1200s	t = 3600s	Compos.		t = 1200s	t = 1200s
	[Crévits et al. 2012]			[Cherfi 2009]			value	time		
I7	24591	24587	**24592**	24587	24584	24589	24584	128	**24592**	**24592**
I8	36888	36889	36888	**36894**	36885	36886	36843	156	36885	36886
I9	49174	49183	49179	49179	49171	49176	49139	188	49179	**49185**
I10	61469	**61471**	61466	61464	61464	61465	61423	228	61464	61465
I11	**73784**	73779	73779	73783	73777	**73788**	73739	233	73780	73782
I12	**86091**	86083	**86091**	86080	86080	86080	86035	252	86081	86084
I13	98435	**98445**	98433	98438	98433	98437	98380	302	98433	98437
INST1	**10738**	10728	10728	**10738**	10720	**10738**	10693	101	10727	10733
INST2	**13598**	**13598**	**13598**	**13598**	**13598**	**13598**	13580	120	**13598**	**13598**
INST3	10949	10943	10949	10944	10942	10946	10897	98	**10955**	**10955**
INST4	14442	14445	14446	14442	14442	14447	14411	130	**14452**	**14452**
INST5	17058	17055	17057	17053	17058	**17059**	17025	131	17055	**17059**
INST6	**16835**	16832	16832	16827	16830	16832	16791	115	16827	16830
INST7	**16440**	**16440**	**16440**	**16440**	**16440**	**16440**	16397	121	**16440**	**16440**
INST8	17510	**17511**	17510	17510	17507	**17513**	17474	145	17507	17509
INST9	17760	17760	17753	**17761**	17754	17754	17705	126	17757	17754
INST10	**19320**	**19320**	19314	19316	19301	19306	19266	127	19314	19316
INST11	**19446**	**19446**	**19446**	19441	19429	19435	19396	114	19441	19441
INST12	21733	**21738**	**21738**	21732	21732	21732	21698	137	**21738**	**21738**
INST13	21577	**21580**	21577	21577	21577	21577	21569	209	21577	21577
INST14	32873	32872	32872	**32874**	32871	32871	32850	192	32872	32872
INST15	39161	39160	**39162**	39160	39159	39160	39151	320	39160	39161
INST16	**43366**	43363	43363	43362	43363	43364	43343	238	43363	43363
INST17	**54361**	54360	54358	54360	54360	54360	54344	394	54360	54360
INST18	**60467**	**60467**	60465	60464	60464	**60467**	60454	213	60465	60464
INST19	64930	**64932**	64929	64925	64930	64931	64914	244	64931	**64932**
INST20	75613	75611	75613	75612	75613	**75615**	75595	238	75613	**75615**
Best values	11/4	11/4	7/1	6/3					6/2	10/5

default resource-usage aggregation for selecting partial solutions. We do not perform preprocessing, but we do compute a value upper bound in Step 4 of CPH. Since speed is less important than quality, we compute Pareto points in *ProductSum-Bound-Min* in the original $(R+1)$-dimensional value-resource usage space.

Table VI presents the results. Columns 2 to 5 report the results of the state-of-the-art heuristics. Since this version of CPH uses Cplex, we also solved the benchmarks using pure Cplex, both with a 1200s and a 1 hour time limit, to be used as a reference. Columns 6–7 give the results. Columns 8–9 show the values and computation times of the compositional part in CPH, including the compositional improvement step, but before postprocessing. Column 10 shows the final result after Cplex postprocessing, where we consider a time bound of 1200s on the total time available to CPH. That is, the time bound for Cplex postprocessing is set to the difference between 1200s and the runtime of the compositional part. The final column reports the results for pCPH, the parallelizable variant of CPH (see Section 4.6. Given multicore processing and cloud computing trends, the availability of many processors for parallel processing is rapidly becoming realistic. We emulate a parallel computation assuming 10 free processors and the parameter settings reported in Figure 4. Up to the last Cplex postprocessing step, Procedure *OptPP*, the computations always easily complete within

1200s, so also for pCPH, we set the overall time budget to 1200s, meaning that any sequence of sequential steps in the scheme of Figure 4 needs to complete within this budget. Communication costs are small compared to the computation costs given the time budget of 1200s, because only Pareto points need to be communicated between subsequent steps; communication costs are ignored in our emulation.

Table VI reports best values in bold. Unique best values are underlined. In this comparison, we consider heuristics ILPH, IMIPH, IIRH, BLHG, and (p)CPH, but not the Cplex results. The bottom row in Table VI summarizes results, where an entry x/y indicates that a heuristic finds x best values of which y values are unique, in the sense that they are not found by any of the other heuristics. CPH obtains 6 best values out of 27 benchmarks with 2 unique best values. pCPH yields 10 best values with 5 unique ones. The results show that no single heuristic outperforms all other heuristics. All heuristics find at least one unique best value. We may conclude that CPH and pCPH scale in terms of solution quality, when tuned towards solution quality. pCPH outperforms CPH when given the same time budget, showing that the parallel optimization of subproblems pays of in terms of solution quality. It should be noted however that the heuristics of [Crévits et al. 2012] are the fastest heuristics, typically converging to their optimal value within 400s. For these 27 standard benchmarks, CPH and pCPH do not scale as well as the state-of-the-art heuristics in terms of runtime. This is due to the use of Cplex in the compositional improvement and postprocessing steps. This aspect is not important though when considering CPH for realtime application and for MMKP instances that are much larger than these standard benchmarks. None of the heuristics used for comparison in Table VI is suitable for real-time application, and none of them can cope with extremely large MMKP instances such as those occurring in global routing. Both aspects are further investigated in the next two sections.

In the columns reporting the pure Cplex results, results that match or outperform best values found by any of the heuristics are given in bold. Unique best values are underlined. With a time budget of one hour, Cplex finds two unique best values that are not found by any of the heuristics. Based on the 1200s Cplex results, we may conclude though that the heuristics outperform pure Cplex when given equal time budgets.

We can draw several other conclusions that are specific for CPH. First, the results in columns 8–10 confirm that postprocessing is useful, also in the version of CPH tuned for solution quality. Second, although we did not report results without the intermediate compositional improvement steps, we can confirm that these results show that also the intermediate improvements are useful. Finally, the results confirm the versatility of CPH. The parameterization allows tuning of the approach. The compositional nature allows parallelization of the approach, which improves solution quality when compared to a sequential analysis with an equal time budget.

We conclude this section by giving the results of CPH for the large irregular benchmarks INST21-INST30 of Table II. Table VII shows the results. We use CPH in the same configuration as earlier in this section, that is, with Cplex postprocessing and with Pareto minimization in the $(R+1)$-d value-resource usage space, but with the preprocessing that sorts groups on cardinality enabled. We compare CPH with both the default resource usage aggregation and with the scarcity-aware resource aggregation (with, as before, $\alpha = 2$) against pure Cplex as a reference, with a time budget of 1200s. CPH with default aggregation gives better results than pure Cplex in 7 of 10 cases, marked in bold in the table; CPH with scarcity-aware resource aggregation gives better results than Cplex in 9 of 10 cases. Scarcity-aware resource aggregation does sometimes but not always give better results than the default aggregation. The effect of scarcity-aware resource aggregation is smaller for these large benchmarks than for the real-time benchmarks (see Table IV) because Pareto minimization during the compositional steps is done in the nonaggregated $(R+1)$-d value-resource usage space and

Table VII.
Results for large irregular benchmarks.

		t = 1200s			t = 3600s		
Bench.	Relaxed value	Cplex	CPH +*OptPP* default aggr.	CPH +*OptPP* scarc.-aw. aggr.	Cplex	CPH +*OptPP* default aggr.	CPH +*OptPP* scarc.-aw. aggr.
INST21	44315	44246	**44262**	**44262**	44262	**44270**	**44270**
INST22	42076	41930	**41976**	**44976**	41976	41976	41976
INST23	42763	42488	42488	**42502**	42550	**42560**	**42562**
INST24	42252	41776	**41918**	**41910**	41918	41918	41918
INST25	44201	44128	**44138**	44128	44146	**44156**	**44152**
INST26	45011	44806	44806	**44832**	44835	**44869**	**44869**
INST27	87650	87588	**87616**	**87600**	87600	**87616**	**87616**
INST28	134672	134594	**134634**	**134634**	134610	**134634**	**134634**
INST29	179245	179168	**179186**	**179186**	179202	**179204**	**179206**
INST30	214257	214132	214132	**214198**	214198	214198	214198
Improv.			7	9		7	7

because the *OptPP* Cplex postprocessing, which takes only the highest-value solution candidate resulting from the compositional steps into account. The results confirm that CPH can cope with irregularity in MMKP instances. To see whether the positive effect of the compositional phase of CPH lasts when the time budget is increased, we also ran experiments with a time budget of one hour. We see that also then, CPH outperforms pure Cplex. The effect of scarcity-aware resource aggregation becomes even smaller because of the more extensive postprocessing. In several cases, underlined in the table, the 1200s results for CPH improve upon the 3600s results of pure Cplex.

6. CMP RUNTIME MANAGEMENT

In this section, we consider CPH in a chip-multiprocessor (CMP) runtime management scenario, in which computation time of the heuristic is critical. The CMP runtime management approach is described in detail in Shojaei et al. [2009]. This scenario motivates the need for real-time analysis. Complex applications with data- and context-dependent dynamic behavior running on CMP platforms are becoming dominant in embedded systems. When (parts of) applications may start and stop over time, selecting at runtime a resource-effective mapping of the applications on the CMP platforms is challenging.

Figure 6 gives an illustrative example. Consider a system with three applications A, B, and C and 6 processing elements (PEs) allowing two clock speeds (ck1 and ck2, ck1<ck2). Potential application configurations (4-d: energy, speed, number of processors, clock speed) are determined offline. Some configurations are infeasible because of a latency constraint. The objective is to map applications onto the CMP so that latency and resource constraints are met and energy consumption is minimized. Applications may start and stop at arbitrary times. When application A starts at time 0, 4-PE configuration $\bar{c}_1$ is chosen with low clock speed ck1. At time t1, application B starts and among all possible combinations, configuration $\bar{c}_2$ with fast clock ck2 and 2 PEs assigned to each application is selected. When C starts at t2, it is assigned two additional PEs, giving configuration $\bar{c}_3$. In each step, the configurations of the newly arriving application are combined with the configurations of the already active applications, making sure that clock speeds match (Join in the figure). Infeasible configurations (denoted in grey) and dominated configurations (crossed out) are removed (Const resp. Min in the figure).

In general, application configurations are assumed to have a *value* (that takes into account the optimization objectives) and a *multidimensional resource usage* (where resource usage is assumed to be additive). We need to choose exactly one feasible configuration per application, maximizing the total value of the selected configurations,

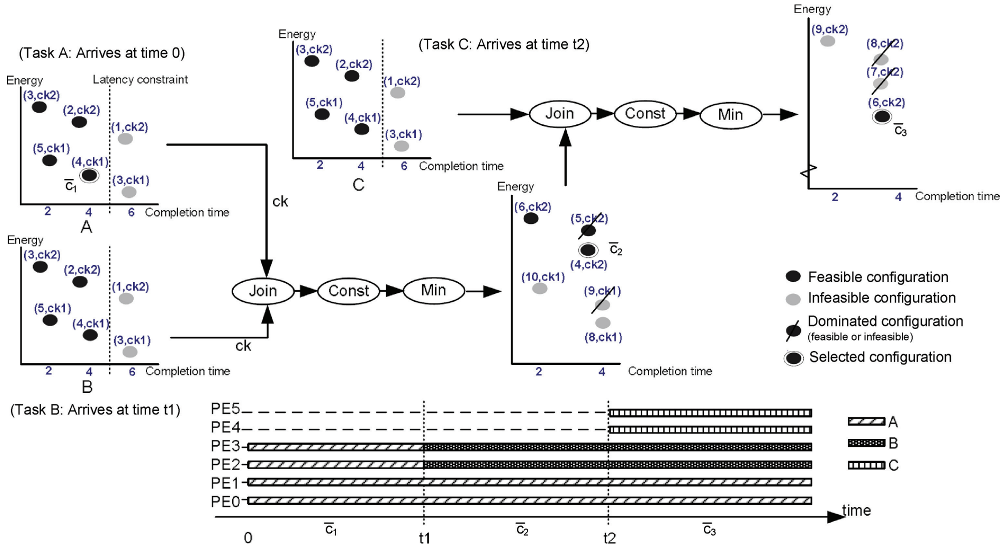


Fig. 6. Configuration selection in CMP runtime management (adapted from Shojaei et al. [2009]).

without exceeding the resource constraints in any dimension. Essentially, an application thus is represented by a group of items in MMKP terminology. Other constraints, however, like in the example the latency constraint and the requirement that clock speeds match, need to be taken into account as well. This leads to a variant of MMKP where some of the items from different groups are incompatible with each other. Enforcing such incompatibility constraints can in fact easily be added to CPH in Procedure *ProductSum-Bound-Min*; this is not always possible in other heuristics. Note that this example of integrating operations in *ProductSum-Bound-Min* furthermore illustrates once again the advantage of using Simple Cull as a basis for Pareto minimization (see the discussion in Section 3.3). Also note that latency constraints can be treated in the same way as resource constraints, which is therefore possible in all existing MMKP heuristics. The resulting optimization problem needs to be solved each time something changes in the set of active applications, that is, each time an application starts or stops executing. The analysis needs to be performed on the embedded platform, within a time budget of at most a few milliseconds.

To evaluate the applicability of CPH for CMP runtime management, we ran experiments on the cycle-accurate SimIt-ARM simulator [SimIt-ARM 2012] for the StrongARM architecture running at 206 MHz. CPH without preprocessing, value bound, intermediate optimization, and postprocessing, and with 2-d Pareto minimization and default resource aggregation was applied on MMKP benchmarks I1 through I6. The results given in Table VIII are for the worst case in which all applications start simultaneously. MMKP instances can be solved in appropriate time budgets of around 1 ms or less. Time budgeting (via Equation (1)) is particularly relevant in this context. For the reported experiment, the analysis time taken by the state-of-the-art RTM heuristic was taken as time budget for CPH. Within the time budget, the CPH result is better than the RTM result in 4 of 6 cases.

Another important aspect in this particular application of CPH is compositionality. Figure 7, taken from Shojaei et al. [2009], compares the performance of CPH with the RTM heuristic, on the StrongARM platform, for benchmark I5 with $L = 10$ and assuming that applications start 5 at a time (a) or 1 at a time (b). The results illustrate that CPH can be applied incrementally and that the performance does not depend on the number of active applications (i.e., the number of groups already covered in

Table VIII.
Comparison of the CPH and RTM heuristics on a StrongARM 206 Mhz platform. The table reports for both heuristics the obtained absolute value, the relative value with respect to the optimum, and the needed analysis time.

	RTM			CPH		
Bench.	val	%opt	t(ms)	val	%opt	t(ms)
I1	148	85	0.1	154	89	0.1
I2	327	93	0.2	344	98	0.2
I3	1407	87	0.6	1266	80	0.6
I4	3326	92	0.9	2911	81	0.9
I5	3705	95	1.0	3900	100	1.0
I6	4313	90	1.2	4408	92	1.2

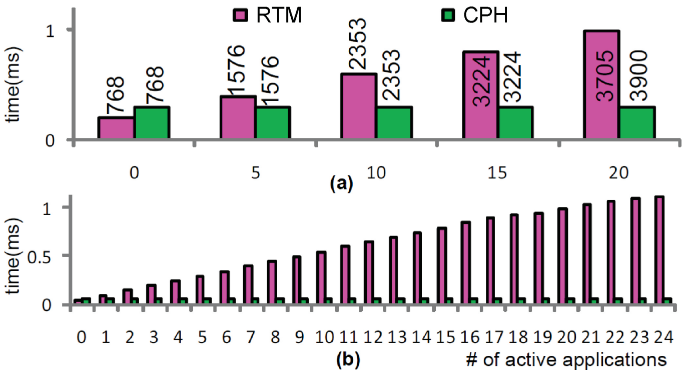


Fig. 7. An illustration of compositional MMKP solving, using benchmark I5, with (a) 5 applications starting at a time, and (b) 1 application starting at a time (from Shojaei et al. [2009]).

the compositional computation). RTM needs to solve the complete MMKP instance for all applications each time, which results in increasing execution times when more applications are active. Compositionality only applies when applications start. When applications stop, that is, when an item from one group in the MMKP has to be removed from the solution, both RTM and CPH need to recompute the full remaining MMKP instance to optimize the mapping. In practice, this is not a problem. If an application stops, it is guaranteed that the remaining configuration is still feasible, so the reconfiguration process is not time critical.

7. GLOBAL ROUTING

Global routing is the step in the physical implementation of Integrated Circuits (ICs) that plans the wiring between nanoscale components. Today's ICs contain millions of nets that should be routed. Each route is a complex 3-d structure, spanning over many metal layers. Global routing thus is a very large-sized combinatorial optimization problem. Scalability of solution heuristics with respect to the size of an MMKP instance is essential. This case study illustrates the application of CPH in a design-time optimization problem.

In global routing, we are given a grid-graph G. Each net is represented by a subset of vertices that denote the locations of its terminals on the grid. To route a net, its terminals should be connected using the edges in G. For a given route of a net, its wirelength is the length of the Steiner tree connecting its terminals. Each edge in G has a capacity representing the maximum number of routes that can pass the edge without overutilizing the routing resource associated with the edge. The goal of global

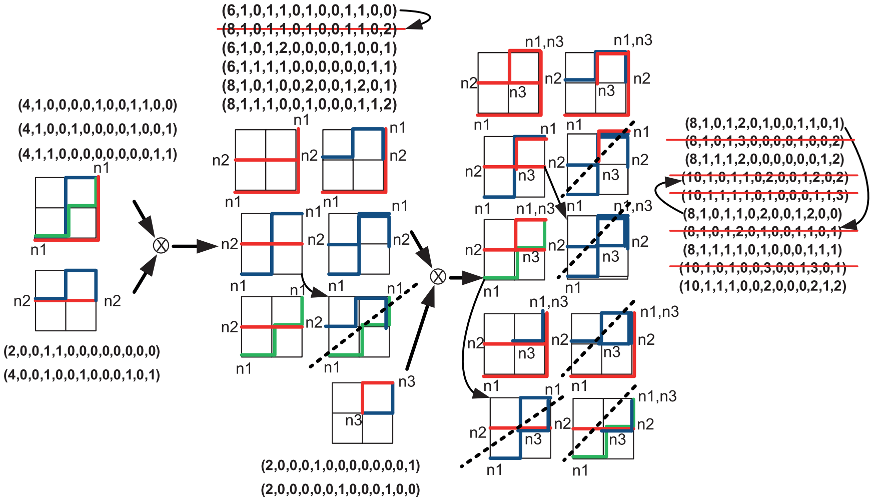


Fig. 8. Collaborative global routing.

routing is to route all the nets subject to the edge capacity constraints, while minimizing the summed wirelength.

Global routing problem variants can be modeled as MMKP instances [Shojaei et al. 2010]. To illustrate scalability of CPH, we discuss collaborative global routing in which wirelength is minimized by reusing good routes found by different existing routing tools. Each net is represented as a group of items in which an item is a candidate route for the net. The wirelength corresponding to a candidate route is its value (smaller values being better). Each edge in the grid-graph G corresponds to a resource. The resource usage of a candidate route is equal to the number of edges that it uses in G. We exclude the edges in the z-axis in computing resource usage; these edges correspond to 'infinite-capacity' vias connecting multiple metal layers. The value of each candidate route, which takes into account via usage, may thus be larger than its (aggregate) resource usage. The result is an MMKP instance for which value needs to be minimized instead of maximized. Given the typical number of nets to be routed (several hundred thousands, several millions) and the size of the 3-d grid-graph (e.g., $300 \times 300 \times 6$ vertices), the MMKP instances are extremely large. We need a very scalable heuristic to solve them.

Figure 8 shows a snapshot of the CPH execution for a small 2×2 grid. The nets to be routed, n_1, n_2, n_3, have 3, 2, and 2 candidate routes, respectively. For example, the top leftmost grid shows the three routes for n_1. Terminal vertices are marked in the grid. The three candidate routes define a group of three items, each item given as a 13-tuple. The first entry of an item is its value (wirelength). In this example, all n_1 candidate routes have a wirelength of four units. The other 12 entries correspond to the edges in the grid-graph. A 1-entry indicates that the candidate route passes the edge. Edges have a capacity of two in this example; at most two routes can pass an edge in a valid solution. Figure 8 shows two steps of the computation required in solving the MMKP instance for this example. First, the candidate routes for n_1 and n_2 are combined, resulting in six options. The second option is dominated by the first one and crossed-out because of higher wirelength and resource usage. The remaining five options are then combined with the two candidate routes for net n_3, resulting in 10 new options of which five are either infeasible or dominated. Once all the nets are processed, the option with the smallest value (total wirelength) is the final solution.

Table IX.
Results for collaborative global routing. Wirelength is scaled to 10^5 units.

Benchm.	Grid	#Nets	MGR WL	CPH WL	%Impr.	t(min)
adaptec1	$324 \times 324 \times 6$	176715	52.86	52.50	0.68	6.13
adaptec2	$424 \times 424 \times 6$	207972	51.47	51.13	0.66	6.48
adaptec3	$774 \times 779 \times 6$	368494	128.91	127.79	0.87	15.99
adaptec4	$774 \times 779 \times 6$	401060	119.95	119.04	0.76	15.66
adaptec5	$465 \times 468 \times 6$	548073	153.23	151.15	1.36	18.89
newblue1	$399 \times 399 \times 6$	270713	45.63	45.37	0.57	6.34
newblue2	$557 \times 463 \times 6$	373790	74.49	73.62	1.17	10.62

To evaluate CPH for global routing, we used the routable global routing benchmarks from the ISPD 2007 and 2008 contests [ISPD contests 2007, 2008]. The benchmarks and results are given in Table IX. The first three columns give the name and size of the routing instances. For example, adaptec1 contains about 176 thousand nets to be routed on a $324 \times 324 \times 6$ grid. To get candidate routes, we took the results of five recent global routers, NTHU-Route 2.0 [Chang et al. 2008], FastRoute 4.0 [Xu et al. 2009], BFGR [Hu et al. 2010], NCTU-GR [Dai et al. 2012] and MGR [Xu and Chu 2011]. Each net thus has five candidate routes. Global routing tools are inherently different in the procedures to tackle the routing problem. Combining various routing procedures is very difficult, if not impossible. The selected routes that are reflective of these procedures, however, can be combined into better solutions.

Given the extremely large sizes of the MMKP instances, we applied greedy postprocessing and omitted preprocessing and compositional improvement; 2-d Pareto minimization was done and L was set to 10. None of the existing quality-driven MMKP approaches nor Cplex scales to problem sizes obtained from global routing. Also the efficiency-driven RTM heuristic is not applicable, because it takes the combination of items with the worst values in each of the groups (i.e., highest wirelength routes in this case) as a starting point, assuming this to be a valid solution to the MMKP instance. This is generally not the case for global routing MMKP instances.

Table IX shows the results for CPH. All our analyses ran on a machine with a 2.8GHz Intel CPU and 12GB of memory. MGR turned out to give the best wirelength (WL) for all benchmarks, which is reported in column 4. The wirelength obtained by CPH is given in column 5, and the relative improvement compared to the MGR wirelength in column 6. We always obtain improvement, although this cannot be guaranteed in general. Although the relative improvement may seem small, the gain in absolute wirelength is substantial. It is competitive when considering the typical improvement obtained by recent advances in global routing. Further, our solutions satisfy edge capacity constraints by construction, which is not always the case for other approaches. The runtime for CPH, reported in column 7, is always less than 20 minutes. The results show that postprocessing results of global routing tools is worthwhile. Even though a single tool gives the best WL for all benchmarks, using routes of other tools may provide a further improvement of those results. The results of this global routing case study confirm that the compositionality and parameterization of CPH provide scalability to extremely large MMKP instances.

8. CONCLUSIONS

This article presents CPH, a novel parameterized compositional heuristic to solve MMKP. In the embedded systems and design automation domains, MMKP instances appear in many different combinatorial optimization problems, both problems that need to be solved at runtime and those that can be addressed at design time. CPH provides a versatile and flexible framework that can be tuned to the problem at hand.

When tuned towards analysis efficiency, CPH can find good quality solutions in real time, within a predefined time bound. CPH outperforms the state-of-the-art heuristic targeting real-time performance. A chip-multiprocessor resource management case study shows that real-time performance is possible even on embedded platforms. The compositional nature of CPH is a strength that enables incremental problem solving and parallelization. When tuned to solution quality, CPH and its parallelizable variant pCPH find new best solutions in 2 resp. 5 out of 27 standard benchmarks. The quality of solutions for these benchmarks is competitive, although CPH needs more analysis time than existing state-of-the-art heuristics. When tuned for scalability with respect to the size of MMKP instances, CPH can be applied successfully in a case study on global routing of wires in integrated circuits. This proves that CPH scales to extremely large MMKP problem instances.

The parallelizable variant of our approach, pCPH, gives better results than CPH in the same time budget, by exploiting multiple processors (or cores). The parallelization scheme of pCPH does not fully exploit available processors. Given the trends towards multicore processors and cloud computing, decompositions of MMKP into subproblems and parallelization and load-balancing schemes that fully exploit the available processors provide promising directions to further improve the CPH approach.

Implementations. Implementations of the algorithms presented in this article are available through http://www.es.ele.tue.nl/pareto/mmkp/.

ACKNOWLEDGMENTS

We thank the associate editor and the reviewers for their constructive feedback, which helped to improve the content and presentation of the article.

REFERENCES

Akbar, M. M., Rahman, M. S., Kaykobad, M., Manning, E. G., and Shoja, G. C. 2006. Solving the multidimensional multiple-choice knapsack problem by constructing convex hulls. *Comput. Oper. Res. 33*, 5, 1259–1273.

Bentley, J. 1980. Multidimensional divide-and-conquer. *Comm. ACM 23*, 214–229.

Borzsonyi, S., Kossmann, D., and Stocker, K. 2001. The skyline operator. In *Proceedings of the IEEE Conference on Data Engineering*. IEEE, 421–430.

Chang, Y.-J., Lee, Y.-T., and Wang, T.-C. 2008. NTHU-Route 2.0: a fast and stable global router. In *Proceedings of the IEEE International Conference on Computer-Aided Design*. 338–343.

Cherfi, N. 2009. Hybrid algorithms for knapsack problems. Ph.D. thesis, University of Paris I, France.

Cherfi, N. and Hifi, M. 2008. A column generation method for the multiple-choice multi-dimensional knapsack problem. *Comput. Optim. Appl. 46*, 1, 51–73.

Cherfi, N. and Hifi, M. 2009. Hybrid algorithms for the multiple-choice multi-dimensional knapsack problem. *Int. J. Operational Research 5*, 1, 89–109.

Cplex 2012. IBM ILOG Cplex. http://www-01.ibm.com/software/websphere/products/optimization/academic-initiative/.

Crévits, I., Hanafi, S., Mansi, R., and Wilbaut, C. 2012. Iterative semi-continuous relaxation heuristics for the multiple-choice multidimensional knapsack problem. *Comput. Operations Research 39*, 32–41.

Dai, K.-R., Liu, W.-H., and Li, Y.-L. 2012. NCTU-GR: Efficient simulated evolution-based rerouting and congestion-relaxed layer assignment on 3-d global routing. *IEEE Trans. VLSI Syst. 20*, 3, 459–472.

Geilen, M. and Basten, T. 2007. A calculator for Pareto points. In *Proceedings of the Conference and Exhibition on Design, Automation and Test in Europe*. IEEE, 285–290.

Geilen, M., Basten, T., Theelen, B. D., and Otten, R. 2005. An algebra of Pareto points. In *Proceedings of the 5th International Conference on Application of Concurrency to System Design (ACSD'05)*. IEEE, 88–97.

Geilen, M., Basten, T., Theelen, B. D., and Otten, R. 2007. An algebra of Pareto points. *Fundamenta Informaticae 78*, 1, 35–74.

Godfrey, P., Shipley, R., and Gryz, J. 2005. Maximal vector computation in large data sets. In *Proceedings of the 31st International Conference on Very Large Databases*. VLDB Endowment, 229–240.

HANAFI, S., MANSI, R., AND WILBAUT, C. 2009. Iterative relaxation-based heuristics for the multiple-choice multidimensional knapsack problem. In *Proceedings of the 6th International Workshop on Hybrid Metaheuristics*, Lecture Notes in Computer Science, vol. 5818, 73–83.

HIFI, M., MICHRAFY, M., AND SBIHI, A. 2004. Heuristic algorithms for the multiple-choice multidimensional knapsack problem. *J. Operational Research Soc. 55*, 12, 1323–1332.

HIFI, M., MICHRAFY, M., AND SBIHI, A. 2006. A reactive local search-based algorithm for the multiple-choice multidimensional knapsack problem. *Comput. Optim. Appl. 33*, 2–3, 271–285.

HIREMATH, C. S. AND HILL, R. R. 2007. New greedy heuristics for the multiple-choice multi-dimensional knapsack problem. *Int. J. Operational Research 2*, 4, 495–512.

HU, J., ROY, J., AND MARKOV, I. 2010. Completing high-quality global routes. In *Proceedings of the International Symposium on Physical Design*. ACM, 35–41.

ISPD CONTESTS 2007, 2008. http://www.ispd.cc/contests/.

KHAN, S. 1998. Quality adaptation in a multisession multimedia system: Model, algorithms and architecture. Ph.D. thesis, Univ. of Victoria, Victoria, B.C., Canada.

KHAN, S., LI, K. F., MANNING, E. G., AND AKBAR, M. M. 2002. Solving the knapsack problem for adaptive multimedia systems. *Stud. Inform. Univ. 2*, 1, 157–178.

KUNG, H., LUCCIO, F., AND PREPARATA, F. 1975. On finding the maxima of a set of vectors. *J. ACM 22*, 469–476.

MMKP BENCHMARKS. 2010. MMKP benchmarks. ftp://cermsem.univ-paris1.fr/pub/CERMSEM/hifi/MMKP/MMKP.html.

MMKP BENCHMARKS. 2012. MMKP benchmarks. http://www.es.ele.tue.nl/pareto/mmkp/.

MOSER, M., JOKANOVIC, D. P., AND SHIRATORI, N. 1997. An algorithm for the multidimensional multiple-choice knapsack problem. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci. 80*, 3, 582–589.

PARRA-HERNANDEZ, R. AND DIMOPOULOS, N. J. 2005. A new heuristic for solving the multichoice multidimensional knapsack problem. *IEEE Trans. Syst. Man Cybernetics, Part A 35*, 5, 708–717.

PISINGER, D. 1995. Algorithms for knapsack problems. Ph.D. thesis, DIKU, University of Copenhagen, Denmark.

PREPARATA, F. AND SHAMOS, M. 1985. *Computational Geometry: An Introduction*. Springer.

SBIHI, A. 2003. Hybrid methods in combinatorial optimization: Exact algorithms and heuristics. Ph.D. thesis, Univ. of Paris I, France.

SBIHI, A. 2007. A best first search exact algorithm for the multiple-choice multidimensional knapsack problem. *J. Comb. Optim. 13*, 4, 337–351.

SHOJAEI, H., GHAMARIAN, A., BASTEN, T., GEILEN, M., STUIJK, S., AND HOES, R. 2009. A parameterized compositional multi-dimensional multiple-choice knapsack heuristic for CMP run-time management. In *Proceedings of the IEEE/ACM Design Automation Conference*. ACM, 917–922.

SHOJAEI, H., WU, T.-H., DAVOODI, A., AND BASTEN, T. 2010. A Pareto-algebraic framework for signal power optimization in global routing. In *Proceedings of the International Symposium on Low-Power Electronics and Design*. ACM. 407–412.

SIMIT-ARM 2012. SimIt-ARM. http://simit-arm.sourceforge.net.

XU, Y. AND CHU, C. 2011. MGR: Multi-level global router. In *Proceedings of the IEEE International Conference on Computer-Aided Design*. IEEE, 250–255.

XU, Y., ZHANG, Y., AND CHU, C. 2009. Fastroute 4.0: global router with efficient via minimization. In *Proceedings of the Asia and South Pacific Design Automation Conference*. 576–581.

YKMAN-COUVREUR, C., NOLLET, V., CATTHOOR, F., AND CORPORAAL, H. 2011. Fast multidimension multichoice knapsack heuristic for MP-SoC runtime management. *ACM Trans. Embed. Comput. Syst. 10*, 3, Article 35.

YKMAN-COUVREUR, C., NOLLET, V., MARESCAUX, T., BROCKMEYER, E., CATTHOOR, F., AND CORPORAAL, H. 2006. Fast multi-dimension multi-choice knapsack heuristic for MP-SoC run-time management. In *Proceedings of the International Symposium on System-on-Chip*. IEEE, 1–4.

YUKISH, M. 2004. Algorithms to identify Pareto points in multi-dimensional data sets. Ph.D. thesis, Pennsylvania State University.

Received August 2012; revised March 2013; accepted April 2013