Received 21 January 2023, accepted 16 February 2023, date of publication 23 February 2023, date of current version 1 March 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3248283

RESEARCH ARTICLE

Packing Multiple Types of Cores for Energy-Optimized Heterogeneous Hardware-Software Co-Design of Moldable Streaming Computations

SEBASTIAN LITZINGER[1], JÖRG KELLER[1], AND CHRISTOPH KESSLER[2]
[1]Faculty of Mathematics and Computer Science, FernUniversität in Hagen, 58084 Hagen, Germany
[2]Department of Computer and Information Science, Linköping University, 581 83 Linköping, Sweden

Corresponding author: Sebastian Litzinger (sebastian.litzinger@fernuni-hagen.de)

The work of Christoph Kessler was supported by ELLIIT (Excellence Center at Linköping - Lund in Information Technology), through the project C05 GPAI (General-Purpose AI Computing).

ABSTRACT For fixed-application scenarios in embedded soft-realtime computing, the ideal (w.r.t. energy consumption) heterogeneous multi-core CPU design within given chip dimensions can be configured by composing it from given pre-layouted, rectangular chip submodules for each of a number $K > 1$ of core types, where K in practice is a small constant. For example, $K = 2$ in traditional ARM big.LITTLE designs. Nevertheless, even better solutions might be achieved for $K > 2$, and many feasible combinations can exist. For this purpose, we investigate finding all combinations of instances of $K > 1$ different types of given axis-parallel rectangles that can be packed within a given fixed-size 2D rectangle, and we propose two new packing heuristics: the corner heuristic for $K \leq 4$, and the onion heuristic for larger K. Both heuristics strive to pack cores of the same type close together, to simplify implementation of on-chip bus and shared cache structures. The core combinations can be used in co-optimizing chip configuration, task mapping and scheduling for stream processing applications. We evaluate the corner heuristic for a number of different types of ARM softcores and chip dimensions, and show that it outperforms strip packing techniques from the literature and yields similar results to an advanced rectpack heuristic allowing rotation, though these do not try to pack similar cores closely.

INDEX TERMS Packing rectangles, heterogeneous multi-core CPU, design space exploration, hardware-software co-design.

I. INTRODUCTION

In embedded systems, applications such as streaming computations and platform are often developed simultaneously, so that the platform is targeted towards the application at hand [1]. Within the platform, we focus on the configuration of a processor chip with multiple cores of different types, as computer architecture follows a strong trend towards higher core-level parallelism and towards increased heterogeneity, and will continue to do so for the foreseeable future [2], [3].

In addition to Application Specific Integrated Circuits (ASICs), which first come to mind for such a platform to implement the necessary number of IP cores, also Field Programmable Gate Arrays (FPGAs) implementing soft-cores may be used, in order to allow for a configuration update at a later time. The three main configuration parameters of such a system are application throughput, platform power consumption, and platform area, which all shall be optimized (throughput maximized, power consumption and area minimized), but which are interdependent. The design space of possible solutions is large, as also the application must be configured: we assume a streaming application which consists of a number of parallelizable tasks of known workload that execute and communicate in rounds. A scheduler assigns a degree of parallelism and an operating frequency to each task, maps the tasks onto the different (types of) cores, and sets start times for all tasks. For a fixed application, a static

The associate editor coordinating the review of this manuscript and approving it for publication was Jie Tang.

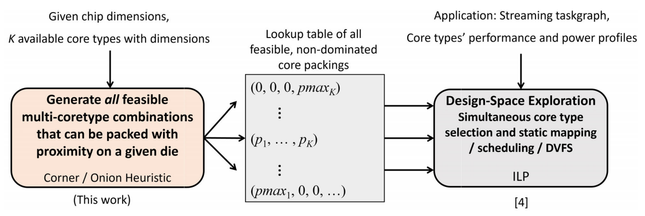
Given chip dimensions,
K available core types with dimensions
Generate all feasible multi-coretype combinations that can be packed with proximity on a given die
Corner / Onion Heuristic
(This work)
Lookup table of all feasible, non-dominated core packings
(0, 0, 0, pmax_K)
(p_1, … , p_K)
(pmax_1, 0, 0, …)
Application: Streaming taskgraph,
Core types' performance and power profiles
Design-Space Exploration
Simultaneous core type selection and static mapping / scheduling / DVFS
ILP
[4]

FIGURE 1. Our approach generates a table of all different feasible non-dominated multi-type core combinations, which will be considered by the ILP solver [4] in the subsequent design space exploration phase.

scheduler performs these actions prior to actual execution. Thus, scheduling can be combined with the question of core provisioning in the platform, leading to integration of scheduling and design space exploration.

In order to reduce the solution space, one can constrain one parameter, such as giving a maximum area, and find the Pareto front of all optimal combinations of throughput and power consumption, or one can constrain two parameters, such as maximum area and minimum throughput, and optimize the third. Overheating the chip can be avoided by constraints on maximum frequency or task placement.

In a previous work [4], we have presented an integer linear program (ILP) that combines this design space exploration with the actual task allocation, scheduling and frequency scaling to obtain a Pareto front of solutions that show minimum power consumption compared to throughput, given an area constraint. Yet, computing the area need of a collection of processor cores of different types poses a problem in itself, even in the absence of different integration technology generations. A simple solution might rely on knowledge of approximate area consumption for each core type to compute total area as a weighted sum, leaving out geometry considerations, and thus overestimating the number of possible cores. An exact solution might layout the necessary cores to see if they can be placed (and routed) within the die area. While this solution is already cumbersome for pre-layouted IP cores, the design space exploration needs many combinations of different cores, which would have to be checked for feasibility, i.e. such an approach is impractical.

As an intermediate, we follow a geometric approach: if there are K different core types with known rectangular dimensions, and the dimension of the die rectangle is fixed, then for any combination of $(p_1, \ldots, p_{K-1})$ cores of types 1 to $K - 1$, that can be packed within the die rectangle, we compute a layout that fits the maximum number p_K of cores of type K that we can still place on the die. Thus, we compute all possible, non-dominated combinations of core counts $(p_1, \ldots, p_K)$, which can be used as a reference table in the above ILP (see also Figure 1). This solution is much faster than providing an exact layout, and it yields more realistic numbers than the simple area calculation. The *corner heuristic* that we present and employ works for $K \leq 4$ core types, and can be extended to more core types via the *onion heuristic*. As an additional advantage, the heuristic tries to place cores of one type in several rows, so that structures like buses are as short as possible and the regular geometry supports bus layout. Similarly, resources like shared caches can be placed easier in such a geometry. We call this feature *proximity property*. A potential concern with a layout placing all cores of a type in close proximity is that the hardware may be damaged due to thermal issues. However, in addition to frequency scaling and specifying different maximum operating frequencies depending on cumulative core load, modern CPUs will employ throttling to avert imminent damage to the chip caused by overheating. Of course, it is still advisable to apply an adequate cooling solution to prevent throttling from happening under normal operating conditions. What is more, the scheduler may consider thermal issues when placing tasks onto different cores (see [5] for how this can be achieved on the application level).

We formalize the *proximity* property and compare our heuristic with state-of-the-art placement approaches both with respect to packing density and with respect to proximity, and find that it is competitive with respect to density and superior with respect to proximity.

Overall, this article makes the following contributions:

- We present two heuristic algorithms: the corner heuristic for $K \leq 4$, and as its generalization the onion heuristic for arbitrary number K of core types, to solve the problem of generating *all* possible non-dominated optimized combinations of different numbers of pre-layouted cores of K different types that can be packed with proximity within a die rectangle with given dimensions. We also suggest a generalization to consider restricted rotation in the packings.
- We experimentally evaluate and compare our heuristics with respect to optimization time and aggregated solution quality with state-of-the-art rectangle packing algorithms, using core type parameters taken from real-world ARM cores and for different die sizes.

The remainder of this article is organized as follows. Section II discusses related work. Section III formalizes the problem, introduces the corner heuristic and sketches how to extend it to an onion heuristic. Section IV presents experimental results to compare the placement performance of the corner heuristic to other heuristics and to an upper bound. Section V summarizes and proposes future work.

II. RELATED WORK

Floorplanning as the placement of general cells on a 2-dimensional area has been applied as one step in the VLSI design chain for decades, see e.g. Lengauer's book [6]. While those cells typically are much smaller than a complete core, also floorplanning for a multitude of (possibly different) cores has been considered in design space exploration [7], [8]. Yet, the design goal there is often not to place the maximum number of cores, but a fixed combination of different cores that is optimal with respect to a required throughput or power consumption. This is also true for [9] where a placement heuristic for two different core types is given, which we generalize to $K > 2$ core types in this paper. While [9], by design,[1] places similar cores as closely as possible, it does not formalize proximity, only considers two core types, and does not use that placement, but the achieved core counts serve as a lower bound in a design space exploration.

The decision problem of placing a *complete* set of m rectangles with fixed orientation[2] into a bounding rectangle of given dimension $B \times H$ is NP-complete [10], and likewise placing m such rectangles in a rectangle of given area but variable dimensions [11]. The corresponding optimization problem is to find the minimum area rectangle that can accommodate all m rectangles. The problem can be generalized to 2D bin packing by minimizing the number of fixed-size/area bounding rectangles needed to accommodate all m objects. Note that we consider here a related but different problem: Given an arbitrarily large supply of rectangular cores of K different types, i.e. sizes, find all feasible, non-dominated (see Section III) combinations of these that can be packed within a given $B \times H$ rectangle. A related problem is strip packing [12], where a given sequence of rectangles is packed into a strip of given width, to minimize strip length.

Huang and Korf [13] consider an inverse problem: given a number of rectangles, find the smallest enclosing rectangle. However, even if that enclosing rectangle has less area than the chip rectangle, this does not indicate that the rectangles fit into a chip, i.e. a feasible solution.

Du et al. [14] use a system with CPU cores, GPU and FPGA, i.e. $K = 3$ core types, thus going beyond ARMs big.LITTLE with $K = 2$.

In addition to the above goals, we require that rectangles of same type be packed close together, which existing techniques do not have as a constraint or objective. Huang et al. [15] investigate rectangle packing, but without considering to pack equal-sized rectangles close together. While they mention a "corner-occupying action", their algorithm does not show the concept of arranging similar cores in successive rows in a corner as in our corner heuristic. Jylänki [16] provides a survey of rectangle packing algorithms (some of which allow rotation by 90 degrees). The `rectpack` library contains a selection of these. Yet, no algorithm used in the code tries to place equal-sized rectangles close to each other.

All previously mentioned approaches only provide one packing, while our approach provides all possible packings, so that a user can select the most suitable solution among those of similar core counts according to further needs.

III. CORNER AND ONION PACKING HEURISTICS

A. PROBLEM FORMULATION

A *(system) configuration* $(p_1, \dots, p_K) \in \mathbb{N}_0^K$ describes a heterogeneous multicore system with K core types that is composed of $p_1 \geq 0$ cores of type $1, \dots,$ and $p_K \geq 0$ cores of type K.

Each core type $i = 1, \dots, K$ comes with a given rectangular layout of dimensions $b_i \times h_i$, and we assume that the b_i (and likewise the h_i) sizes are pairwise general, i.e., not integer multiples of each other or of some base size unit, so that a discretization of the two-dimensional chip area for the core placement is not applicable. Even in the case of a possible discretization, sub-solutions will differ by different forms of their border which prevents re-use in an approach such as dynamic programming.

A configuration $(p_1, \dots, p_K)$ is *feasible* if p_i instances of rectangle type i, for all $1 \leq i \leq K$, can be packed together in the given rectangle so that they do not overlap.

Each feasible configuration can be associated with one[3] feasible *placement* as proof of its feasibility. A *feasible placement* is defined by a vector containing the positions of all $\sum_{i=1}^{K} p_i$ cores' rectangles such that these are placed within the given $B \times H$ rectangle and do not overlap.

A configuration $p_1, \dots, p_K$ is *dominated* by some other feasible configuration $(p'_1, \dots, p'_K)$, if there exists some core type $q \leq K$ such that $p'_q > p_q$ and for all $r \neq q$, $p'_r \geq p_r$. For example, for $K = 2$, $(1, 1)$ is dominated by $(1, 2)$ and both by $(2, 2)$, while $(1, 2)$ is not dominated by $(2, 1)$. Dominated configurations are uninteresting, because they are not Pareto-optimal with respect to the number of cores that can be accommodated. Hence, we will only keep the non-dominated configurations, and it will be up to the optimized software [9] to decide if the additional core(s) are used or powered down.

A feasible placement is called *contiguous* if the rectangles of each core type are placed contiguously, i.e., any two rectangles of the same type are either direct neighbors sharing one edge or are connected by a path of such pair-wise directly neighbored rectangles of the same type. We call a feasible configuration *contiguous* if it has a feasible, contiguous placement associated with it.

The problem that we solve (*ALL-ND-F-C-CONFs*) is defined as follows: Given chip rectangle dimensions $B \times H$ and $K > 1$ different core types with their pre-layouted core dimensions $b_1 \times h_1, \dots, b_K \times h_K$, we want to generate the set of *all* feasible, contiguous, non-dominated configurations $(p_1, \dots, p_K)$, and annotate each such configuration with one feasible contiguous placement, e.g., the one with best proximity score, cf. Sect. IV-D. As all reported configurations are non-dominated, we compute a K-dimensional Pareto front.

Note that, in itself, ALL-ND-F-C-CONFs is not an optimization problem, but a set of many decision problems to be solved. It will calculate the entries in the configuration table for parameterization of the actual optimization algorithm for hardware-software co-design described in [9].

[1] The placement heuristic in [9] is actually a variant of the special case of the corner heuristic (Sect. III) for $K = 2$, where in [9] the placement of the two different core types instead starts from *opposite* corners of the chip rectangle, but otherwise proceeds in the same way.

[2] Rotations further increase the solution space, unless all rectangles are squares.

[3] In general, there will exist multiple such placements for the same system configuration.

B. CORNER PLACEMENTS AND CORNER HEURISTIC

In order to constrain the number of solutions of the feasible placement problem and to enforce feasible placements, we arrange all rectangles of the same type into one corner of the boundary rectangle, as illustrated in Fig. 2 (left). This seems to constrain our approach to $K \leq 4$, yet can be overcome.

Before presenting the complete heuristic, we start with a slightly simplified problem. We are given K different types of cores, each with rectangular shape, to be placed on a rectangular chip. For core types $i = 1, \ldots, K-1$, the number c_i of cores of type i is given (and all those cores fit onto the chip). We place the cores of each type so that they are close together. Afterwards, we place as many cores of type K as possible. For $K \leq 4$, we place the cores of each type in one corner of the chip rectangle. For type $i = 1$, the complete rectangle is still available. If all c_1 cores of type 1 fit into one row or one column of the chip rectangle, we have the following possibilities. As a first possibility, we place all c_i cores in a row. Alternatively, we can place $c_i - 1$ cores in a row, and one core in the next row. Thus, we can explore all combinations of cores placed onto rows such that *each subsequent row contains at most as many cores as the previous row*. The last possibility is that all c_1 cores are placed in one column, i.e. one core per row. Fig. 2 (left) shows all possibilities to place $c_1 = 4$ cores. If only $t < c_1$ cores fit into the first row, the number of possibilities is reduced.

The total number $n(c, t)$ of possibilities of placing c cores with at most t cores in the first row can be computed as the sum of all possibilities when j cores are placed in the first row, for $j \leq \min(c, t)$. The recursion ends as $n(c, 1) = 1$ (all cores in one column) and $n(0, t) = 1$ (no more cores to be placed).

$$n(c, t) = \sum_{j=1}^{\min(c,t)} n(c - j, j). \tag{1}$$

When placing cores of type $i = 2$, we proceed similarly for an adjacent corner, only the rectangle is not empty anymore, i.e. the first row or column might be shortened. By construction, the placement of a further core type is only restricted by the maximum number of cores within the first row. Some placements might not be possible, for example it could be that not all c_i cores can be placed into one column because of insufficient height of the chip rectangle. Therefore, strictly speaking, the equal sign in (1) must be a less than or equal to.

Fig. 2 (mid) shows the only possible placement of a fourth core type for one possible placement of 2, 1 and 1 cores of the first three core types, respectively. The figure also illustrates that the last core type can (and should) be treated differently: We can allow any placement even if not complete rows can be filled, if rows do not start at the chip border, and even if subsequent row (parts) have a greater length than the previous row. As a nice property, the cores will still cover a connected area by the construction of the placements for the previous core types. Fig. 2 (right) shows 27 copies of core type 4. The special treatment of the last core type forbids to argue on the different bounding rectangles containing the c_i cores of core type i. While this is sufficient to determine the placements of the next core type as a succeeding row/column cannot be longer than a previous one, Fig. 3 (top) illustrates for $K = 2$ that different placements of core type 1 (in blue) with identical bounding rectangles allow different numbers of cores of the last core type 2 (in yellow). Similar examples exist for larger K. If $K > 4$, different placements within the same bounding box may lead to different opportunities for placing type $i > 4$ in the onion heuristic, see Section III-C.

To find all possible placements of all core types, our *corner heuristic* (cf. Algorithm 1) computes upper bounds on the number of cores of each type, and finds all possible placements for all combinations of core types 1 to $K-1$, extended by as many cores as possible for core type K. By default, we do this in decreasing order of core area and using corners in clockwise order; cf. Section IV for the impact of core type order. The corner heuristic nests loops per core type to enumerate all feasible different allocations of numbers of cores of the K different types [4] (including 0 for some but not all types). As soon as infeasibility is detected by an enumerating loop, it can break because adding more cores will not lead to any further feasible solutions.[5] Among all reported solutions, a postprocessing step (as part of the reporting function) can eliminate those that are dominated, i.e., not Pareto-optimal.

For the simplified case that core types are treated in order of descending size, cores are quadratic with side length ratio 2:1 between successive core types, and the chip is quadratic with k times the side length of the largest core type, we provide a complexity analysis.

For $K = 2$, we first place the largest core type $i = 1$. If the number of possibilities to place at most $w \leq k$ cores into one row and and use at most $h \leq k$ rows is $N_i(w, h)$, then we seek $N_1(k, k)$ which is

$$N_1(k, k) = \sum_{j=1}^{k} N_1(j, k-1) = O(k!)$$

with the usual constraints $N_1(j, 1) = j + 1$, $N_1(i, 0) = 1$, $N_1(0, *) = 1$ and $N_1(1, j) = j + 1$.

For each number of cores of type $i = 1$, the overall configuration with placement computed by the corner heuristic is a non-dominated configuration, as all convex possible corner-adjacent placements of the type-1 cores are enumerated and, for each one, the largest possible number of cores of type 2 that can be placed in the remaining area can immediately be computed. Thus, the formula above defines the complexity. For $K = 3$, for each placement of cores of type 1, all possible placements for cores of type 2 (starting from 1 to the maximum number possible in the given shape) must be computed. For each such placement, the number of

[4]We note that completely different approaches to the problem, such as greedy strategies, may be possible, but consider them to be outside the scope of our current research.

[5]Note that the loop for the last core type (here, the $p4$ loop) is only conceptual; a more efficient and more effective implementation that replaces the $p4$ loop is described in Section IV-A.

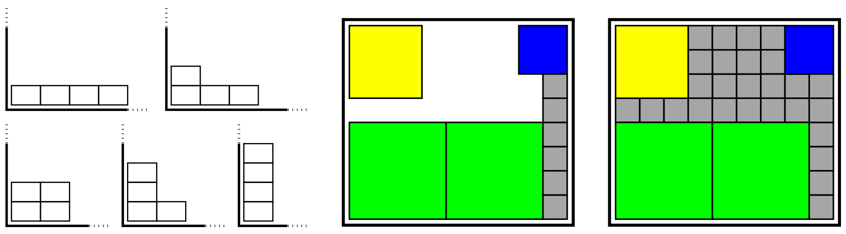

FIGURE 2. Left: all possibilities to place 4 cores in a corner. Mid: sole possibility for placing 6 copies of core type 4 (grey) after 2, 1 and 1 copies of core type 1 (green), 2 (yellow) and 3 (blue), respectively. Right: treating core type 4 as last core type allows 27 copies.

Algorithm 1 Corner Heuristic Algorithm for Packing $K = 4$ Core Types

```
Input: Set $R$ of $K = 4$ rectangle types of size $(w_i, h_i)$, $i = 1, \dots, K$
Result: Set $S$ of different feasible core allocations $(p1, p2, p3, p4)$, initially empty
int $pmax[K], p1, p2, p3, p4$;
Permute core type indexes to establish desired placement order (e.g., random)
for $i = 1; i \leq K; i++$ do
    $pmax[i] \leftarrow$ max. feasible number of cores of type $i$ (if no cores of other types)
end for
for $p1 = 0; p1 \leq pmax[1]; p1++$ do
    create scene with $p1$ cores of type 1 in all possible placements lower left corner
    for $p2 = 0; p2 \leq pmax[2]; p2++$ do
        add to scene $p2$ cores of core type 2 in all possible placements upper left corner
        if no placement possible then
            break loop
        end if
        for $p3 = 0; p3 \leq pmax[3]; p3++$ do
            add to scene $p3$ cores of core type 3 in all placements at upper right corner
            if no placement possible then
                break loop
            end if
            for $p4 = pmax[4]; p4 \geq 0; p4--$ do
                if any placement of $p4$ additional cores of type 4 is possible in lower right corner then
                                        ▷ ...Extend here for Onion heuristic with more loops of "inner" type
                    add $(p1, p2, p3, p4)$ and their placement to set $S$ of solutions
                    break loop
                end if
            end for
        end for
    end for
end for
```

cores of type 3 that can be placed can be determined immediately. Thus, also here the number of possibilities defines the complexity. As the number of rows for cores of type 2 is at most $2k$, and at most $2k$ cores of type 2 can be placed in one row (if no core of type 1 shortens this row), by an argument similar to the above the complexity is $O((2k)!)$ so that the total complexity can be bounded by $O(k! \cdot (2k)!)$. However, this bound might not be sharp anymore.

As with many complex problems, one may wonder whether metaheuristics could pose a promising alternative approach. Metaheuristics are generally applied to optimization problems, whereas we consider a set of decision problems. What is more, metaheuristics are usually adopted for huge search spaces where enumerating all possible solutions is hardly conceivable in a finite amount of time. We on the other hand are interested in generating all non-dominated feasible configurations for given chip size and core dimensions, which cannot be guaranteed by a metaheuristic approach. As our algorithm can inspect the entire search space in a reasonable amount of time for relevant problem sizes, we furthermore fail to see the necessity to employ heuristic techniques on this level. That being said, when checking the feasibility of a given configuration by attempting to generate a feasible example placement, one could pursue a metaheuristic approach. It is to be expected though that this will require a much larger computational effort than our current strategy.

C. ONION HEURISTIC

To compute placements for $K > 4$ core types, we extend the corner heuristic into the *onion heuristic*. We apply the corner heuristic on the first 4 core types, but with the difference that we treat core type 4 similar to the core types 1 to 3, as it is not the last core type. Then the chip rectangle is filled from the borders, but contains an empty space in its interior. We find the largest rectangle fitting into that empty space, and use this rectangle as our new chip border into which to place $K' = K - 4$ core types recursively by the onion heuristic (which reduces to the corner heuristic once $K' \leq 4$). Fig. 3 (bottom), starting from a variant of Fig. 2 (mid), shows two rectangles of equal area that could serve as boundary

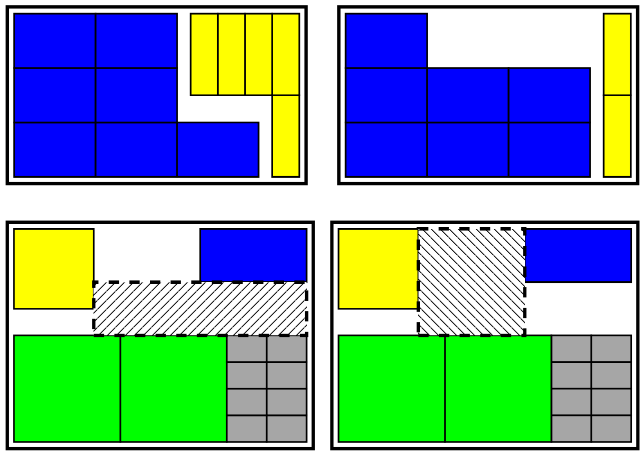

FIGURE 3. Top: different placements of 7 copies of core type 1 (blue) in 3 rows and 3 columns leading to different core counts for last core type $K = 2$ (yellow). Bottom: possibilities for new rectangular boundary to place further core types after placing 2, 1, 1 and 8 copies of core types 1 to 4, resp., when $K > 4$.

for placing further core types. Thus, implementing the onion heuristic necessitates further decision, such as choosing the rectangle of largest size and aspect ratio closest to 1.

Considering Restricted Rotation Until now, the corner and onion packing heuristics do not consider rotation of core rectangles by 90 degrees, in contrast to some other rectangle packing heuristics from the literature (cf. Section II). Not considering rotation leads to less flexibility in packing and might thus miss some good solutions that might be feasible if rotation were allowed. Selecting a fixed orientation for all cores of the same type (and trying both options) is not much more flexible. Instead, mixing both orientations in the same packing will maximize flexibility. However, an exhaustive enumeration of the two possible orientations (portrait and landscape) for each placed rectangle is out of the question, as it would lead to a time complexity that is exponential in the maximum possible number of cores. However, for corner placement we make the following observations that can simplify the mixing of portrait and landscape orientations:

First, in case of multiple rows of one core type in a corner, it makes sense to pack cores of equal orientation on top of each other; hence, we get straight columns (consisting either of portrait cores only or of landscape cores only). This restriction will avoid internal fragmentation in a core type's chip region due to rotation, and will simplify further steps.

Second, due to the non-increasing row lengths property in corner placement, we cannot become infeasible by shifting, among all these columns for the same core type, all portrait columns to the "left" and keeping the landscape columns to their "right" (directions referring to the base side of the rectangle with the filled corner to its left). Likewise, such shifting does not change the packing density, and it re-establishes the non-increasing row lengths property even in the case of rotation. Moreover, it helps to reduce mixed-rotation-caused external fragmentation towards core placements for the next "inner" core type for the onion heuristic, cf. Fig. 4 (left).

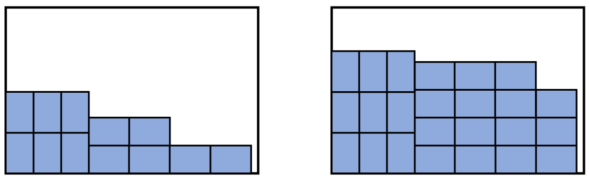

FIGURE 4. Restricted mixing of core orientations, here for flattest corner packings. Left: keeping portrait columns to the left, $pc_1 = 3, p_1 = 12$. Right: inserting a "mezzanine" row of landscape-oriented cores, $pc_1 = 3$ and $p_1 = 24$.

With these considerations we can generalize the corner heuristic and onion heuristic to consider restricted mixed rotation as follows: we replace each of the four nested loops (Algorithm 1) by two nested loops for each core type $i = 1, \ldots, 4$ (i.e., 8 nested loops in total for corner packing): an outer loop for core type i iterates over the number of portrait columns pc_i from 0, 1, 2, ...up to $pmax[i]$, but only as long as these can be accommodated in the bottom row. An inner loop for core type i iterates over the numbers of all type-i instances p_i (regardless of orientation) from pc_i upwards until at most $pmax[i]$, and breaks if no cores of type i can be placed anymore, similarly to the previous loop for core type i in Algorithm 1. These p_i cores are placed row-wise, i.e., first pc_i portrait cores, then landscape cores until the row is filled or no more cores are to be placed; if there are further cores, they are entered in the second row, again first in portrait format and then in landscape format, and so on. As soon as the portrait columns are more than one landscape core higher than the landscape columns, we add an extra row ("mezzanine row", see Fig. 4 (right)) to the landscape columns to balance the height difference between portrait and landscape columns as far as possible while preserving the non-increasing row lengths property.

IV. EXPERIMENTAL RESULTS

A. IMPLEMENTATION OF THE HEURISTIC

In order to evaluate our approach presented in Section III, we have created a Python implementation[6] of the corner heuristic. In doing so, it does not suffice to know the total number of possible placements for a given configuration as specified in (1) but these placements will have to be computed. This problem is equivalent to the problem of providing all partitions of a given integer (sorted in descending order), where the number of summands as well as their absolute value is bounded. The integer then represents the total number of cores of a certain type to be placed, while the limit on the number of terms corresponds to the maximum number of rows, and the maximum absolute value of each term is the maximum number of columns. Cores are always placed in the bottom left corner, and prior to beginning placement of the next core type, the scene is rotated clockwise by 90°. After all cores of a type have been placed, the current partial solution

[6]The source code of our implementation, including the other heuristics used for the evaluation, is available at https://github.com/sglitzinger/corepacking

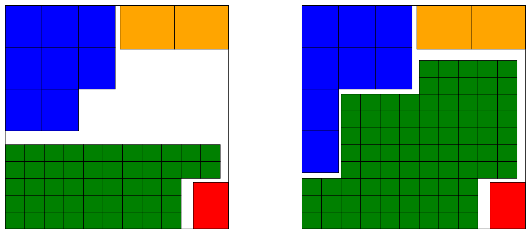

FIGURE 5. **Resulting placements for different methods of maximizing fourth type core count: row-based (left), grid-based (right).**

is checked for feasibility. This is done by computing the coordinates of a polygon which covers the chip area allocated to the cores of a type, for each core type already placed. With the help of the `shapely` library,[7] it can be determined whether any of the polygons intersect, in which case the partial solution is not feasible, and further core placement will be foregone. Obviously, this procedure is invoked only for ≥ 2 polygons, i.e. only after two or more core types have been placed on the chip. When filling the remaining free area on the chip, one has to account for the possibility that there can be rows in which the free space does not extend to the edge of the chip (this is specific for compositions of ≥ 2 core types). Beginning placement on the left may therefore lead to large unused areas when a collision is detected for the initial leftmost placement, and the rest of the row is subsequently ignored, cf. Fig. 5 (left). In contrast, one may check for each possible position of a core in a grid anchored in the bottom left corner whether a collision occurs, and if it does not, place a core in the respective position. This way, waste can be dramatically reduced, see Fig. 5 (right). In the example, this even leads to a different optimal placement of the blue core type, which is now 3-3-1-1 instead of 3-3-2.

B. COMPETING APPROACHES

As we are dealing with a restricted version of an otherwise well-studied problem, there is no shortage of approaches to tackle the general problem of rectangle packing, and it will be interesting to see how well our method fares in comparison. In particular, we have computed solutions with the `rectpack` library[8] implementing various algorithms presented in [16]. There are three categories of algorithms (MaxRects, Guillotine, and Skyline), and we have picked one from each category. Furthermore, we have created a Python implementation of the strip packing heuristic in [17], Algorithm 1 (BestFitPack). Although strip packing solves a slightly different problem (minimizing the height of a strip of given width), it can be applied to our problem.

C. EXPERIMENTAL SETTINGS

In our experiments, we seek to place four different types of cores on a chip of given width and height. Naturally, it is desirable to operate with realistic data, thus we have obtained areas and aspect ratios of various real-world core types: ARM A7 and A15,[9] ARM A72,[10] and a Mali T760 GPU.[11] For each core type, we have assumed a rectangular shape, and as data for individual cores was not available in all cases, we have used values for whole clusters, i.e. including caches, which would have to be placed on the chip in any case. Although these core types originate from different eras and are produced in different technology nodes, which makes it unlikely to encounter a combination of them on a single chip, utilizing real-world data serves to get the problem's size and structure right. We have computed solutions for three chip sizes (16 mm $\times$ 16 mm, 24 mm $\times$ 24 mm, and 32 mm $\times$ 32 mm). As larger chips are not realistic by today's standards, there seems to be no need to consider even larger problem instances.

In each experiment, we have attempted to place a given number of A15, A72, and Mali cores on the chip. If a feasible solution could be computed, A7 cores were subsequently placed until no further placement was possible. For the 24 mm $\times$ 24 mm chip, the maximum number of A15, A72, and Mali cores were 24, 20, and 30, respectively. These figures were computed as $\lfloor \text{width}_{chip}/\text{width}_{core} \rfloor \cdot \lfloor \text{height}_{chip}/\text{height}_{core} \rfloor$ for each core type. Consequently, $25 \cdot 21 \cdot 31 = 16275$ different configurations had to be investigated. Evidently, a feasible solution is only conceivable if the combined area of A15, A72, and Mali cores does not exceed the total chip area. This holds for 4027 of the 16275 configurations, which may therefore serve as an upper bound for the total number of feasible solutions. The 32 mm $\times$ 32 mm chip can accommodate at most 48 A15, or 30 A72, or 56 Mali cores resulting in $49 \cdot 31 \cdot 57 = 86583$ different configurations, with 21150 as an upper bound for the number of feasible solutions as per the consideration above. For the 16 mm $\times$ 16 mm chip, we have 12 A15, 6 A72, or 12 Mali as maximum core counts, which yields 1183 different configurations, 412 being an upper bound here. For the three chip sizes and each configuration, we have attempted to compute a solution in ten different ways:

- MaxRectsBssf (rotation allowed/not allowed),
- GuillotineBssfSas (rotation allowed/not allowed),
- SkylineBl (rotation allowed/not allowed),
- BestFitPack, i.e. strip packing,
- corner heuristic (random order, descending total area, descending core area).

The first three algorithms are part of the `rectpack` library and are detailed in [16]. For the corner heuristic, we have considered the order in which the core types are placed on the chip a parameter. One set of experiments was based on a randomized order, which was individually determined for each configuration. For the other experiments, either the area occupied by a single core of the respective type or the area covered by all cores of a given type for a given configuration

[7]https://github.com/Toblerity/Shapely
[8]https://github.com/secnot/rectpack
[9]https://www.anandtech.com/show/6768/samsung-details-exynos-5-octa-architecture-power-at-isscc-13
[10]https://en.wikichip.org/wiki/arm_holdings/microarchitectures/cortex-a72
[11]https://www.anandtech.com/show/9330/exynos-7420-deep-dive/2

yielded the placement order. There is good reason to expect that the placement order has a notable influence on solution quality for the corner heuristic: as the first and third core types to be placed on the chip are rotated by 90° in the final solution (and the second and fourth core types are not), solutions for the same configuration but based on a permuted placement order may differ substantially.

D. MEASURING SOLUTION QUALITY

We have judged the performance of each method by three characteristics: the number of feasible solutions produced, solution quality in case a feasible solution is available, and proximity of cores of the same type. A measure of solution quality will have to reflect the ultimate goal to place as many A7 cores on the chip as possible for a given feasible placement of A15, A72, and Mali cores. We have thus opted to construe solution quality as the distance (in # A7 cores) to an upper bound, which is the remaining free area on the chip after placing the A15, A72, and Mali cores, divided by the area an A7 core occupies.

Number of Feasible Solutions In a first approach, we may compare the number of feasible solutions each method yields. This information is provided by Table 1. Nearly all `rectpack` methods produce a larger number of feasible solutions than the other approaches, the only exceptions being Guillotine without rotation for the large chip size, where strip packing delivers slightly more feasible solutions, and Skyline without rotation for the smallest chip. For all three chip sizes, the Skyline algorithm with rotation achieves the highest ratio of feasible solutions. It is by far the best option for the medium chip size with 70.3% feasible solutions relative to the upper bound (the second best, Guillotine with rotation, being at 62.1%), whereas for the large chip the gap narrows, and Skyline without rotation replaces Guillotine with rotation as the second best method. Interestingly, for the MaxRects algorithm, rotation only pays off in terms of the number of feasible solutions for the small chip, and strip packing performs much better for the large chip size. The corner heuristic cannot quite keep up with the rectpack algorithms, and for the small and large chips even falls behind strip packing. Although the placement order does have some effect, its extent is apparently rather marginal in terms of the total number of feasible solutions. It should be noted that the upper bound we have computed here represents only a rough estimation, and should by no means be regarded as the supremum; the number of configurations for which a feasible solution exists may well be significantly lower. Here, at least one feasible solution was discovered for 299 configurations for the small chip, 3086 for the medium chip size, and 16656 for the large chip. Another interesting aspect is the number of solutions which are exclusively provided by a particular method, i.e. the number of configurations for which no other method has delivered a feasible solution. For the medium chip size, Skyline with rotation unsurprisingly features the most exclusive solutions as it is far ahead of any other method in terms of the total number of feasible solutions. Moreover, the corner heuristic in all variations stands out here as it awards more exclusive solutions than the other methods aside from the single exception just mentioned when focusing on the medium and large chips. For the small chip, there are very few exclusive solutions overall, which is not surprising as there are not that many solutions in total due to the low number of possible configurations. Mind that exclusive solutions are solutions which are unique to the particular method, which includes variations such as rotation or placement order. In that sense, the placement order should not be neglected for the corner heuristic, as it may enable one to reach solutions unavailable to the other methods.

To get a better understanding of the underlying algorithms' capabilities, Table 2 displays the same results, this time grouped by algorithm. It becomes clear that the corner heuristic profits notably from varying the placement order (which could be expected considering the substantial number of exclusive solutions the corresponding methods offer). It can thus beat strip packing for all chip sizes, the Guillotine algorithm on the medium and large chips, and on the medium chip even outmatch the MaxRects algorithm, which is the default algorithm of the `rectpack` library. Although the Skyline heuristic produces feasible solutions for a slightly larger number of configurations, the ample number of exclusive solutions underlines the corner heuristic's relevance. As already pointed out in Section I, the corner heuristic's solutions feature a placement which assembles cores of the same type in spatial proximity. For our purposes, this is a very beneficial property that the solutions computed by the other algorithms do not inherently possess.

Quality of Feasible Solutions In addition to the number of feasible solutions an algorithm yields the solution quality when it produces a feasible solution is of major interest. To enable a fair comparison, we will focus on the configurations for which each algorithm has delivered a feasible solution. Table 3 shows maximum and average distances in # A7 cores to an upper bound based on the remaining free chip area after placing the A15, A72, and Mali cores (as discussed above). For the two larger chip sizes, we have an identical ranking of the examined algorithms: the Guillotine and MaxRects algorithms perform well when it comes to the maximum distance from the upper bound, followed by the corner heuristic and the Skyline algorithm. Ranked by the average distance to the upper bound the corner heuristic delivers the best results, while there are hardly any noticeable differences between the three algorithms from the `rectpack` library. For the small chip, the corner heuristic cannot match the `rectpack` algorithms' performance not only in terms of the maximum distance but also with regard to the average distance to upper bound. The strip packing algorithm trails behind in any aspect here. Very likely, this can be explained by the adaptation to our specific problem: in order to prioritize the development of a feasible solution, the A15, A72, and Mali cores are placed on the strip first, and afterwards A7 cores are added until the

TABLE 1. Total number of feasible solutions, number of feasible solutions relative to upper bound, number of solutions exclusive to the respective method, nonproximity, and runtime for all considered methods and all three examined chip sizes.

	$16\,\text{mm} \times 16\,\text{mm}$					$24\,\text{mm} \times 24\,\text{mm}$					$32\,\text{mm} \times 32\,\text{mm}$				
method	# feas.	vs. u.b.	# ex.	nonp.	t (s)	# feas.	vs. u.b.	# ex.	nonp.	t (s)	# feas.	vs. u.b.	# ex.	nonp.	t (s)
upper bound	412	100.0%				4027	100.0%				21150	100.0%			
rectpack, MaxRects, rot.	239	58.0%	1	0.251	2	2480	61.6%	9	0.431	81	14313	67.7%	53	0.796	882
rectpack, MaxRects, no rot.	234	56.8%	2	0.139	1	2496	62.0%	7	0.282	43	14730	69.6%	90	0.544	424
rectpack, Guillotine rot.	242	58.7%	1	0.406	1	2502	62.1%	7	0.400	46	14125	66.8%	61	0.473	516
rectpack, Guillotine no rot.	235	57.0%	5	0.224	1	2371	58.9%	3	0.213	33	13658	64.6%	53	0.166	297
rectpack, Skyline rot.	259	62.9%	0	0.205	6	2830	70.3%	102	0.150	218	15285	72.3%	50	0.137	2135
rectpack, Skyline no rot.	224	54.4%	0	0.136	3	2457	61.0%	0	0.134	100	14890	70.4%	8	0.148	1096
strip packing	230	55.8%	1	0.067	0	2244	55.7%	0	0.043	9	13709	64.8%	0	0.062	166
corner heuristic, random	204	49.5%	4	0.000	1	2304	57.2%	33	0.017	252	12892	61.0%	108	0.018	50332
corner heuristic, total area	212	51.5%	2	0.000	1	2331	57.9%	25	0.000	223	12916	61.1%	133	0.012	44073
corner heuristic, core area	186	45.1%	0	0.000	1	2374	59.0%	12	0.001	179	12603	59.6%	123	0.000	27660
total	299	72.6%				3086	76.6%				16656	78.8%			

strip's height reaches the chip height. Due to the nature of the algorithm, areas on the chip where an A7 core could be placed but none of the other cores types are regarded as waste and are not reprocessed at the final stage of A7 placement. An alternative could be to insert several A7 cores into the list of cores to be placed initially. This approach however involves the risk of producing an infeasible solution where otherwise a feasible solution would have been within reach, as A7 cores might also be preferred by the algorithm in situations where cores of other types could be selected.

While the average distances to the upper bound on core count are very similar for all considered `rectpack` algorithms, the distributions differ to some extent, as Fig. 6 illustrates. For the MaxRects and Guillotine algorithms, the vast majority of cases lies within a rather narrow range. This does not hold for the Skyline algorithm, especially for the large chip size (cf. bottom row of Fig. 6). Concerning the corner heuristic, a large number of solutions which are fairly close to the upper bound is accompanied by a small fraction of solutions which are notably worse. The distribution for the strip packing algorithm is visibly wider than the other distributions, particularly for the medium chip size. As with the upper bound on the number of feasible solutions, it should be noted that the existence of any solution which places that many A7 cores is by no means guaranteed. On the whole, it can be considered unlikely as this would imply that a solution with close to zero waste could be constructed for each configuration.

Proximity To assess if the cores of each type are positioned close together, and to compare different heuristics in this respect, we introduce the notion of *nonproximity*. If the intersection of two cores $u, v, u \neq v$ is a line (and not a single point), their distance is 0. Otherwise, their distance is the Manhattan distance between their centroids, normalized in the core dimension. The Manhattan distance seems preferable to the Euclidean distance, as routing of busses to connect cores is normally done along axes. For each core u of a type, the smallest distance to any other core of the same type, $closest(u)$, is determined. Nonproximity for a core type i in a specific configuration is $D_i = \max_{u \in C_i} closest(u)$, where C_i is the set containing all cores of type i. Nonproximity for a configuration is the average of D_i over all core types i.

Table 1 provides nonproximity values, averaged over all configurations for which each method yields a feasible solution. Nonproximity is indeed very low for the heuristic approach, and both ordering criteria pay off in this regard. When it comes to strip packing, the favorable nonproximity figures are due to the same feature which leads to the underwhelming solution quality: areas on the chip once marked as waste are not reprocessed later on and as such, the A7 cores will not be scattered over the chip to utilize every last bit of the die. For MaxRects and Guillotine, rotation leads to worse nonproximity, which is plausible as rotation should offer more possibilities to squeeze in additional cores. Interestingly, this does not always hold for Skyline, which also delivers the best overall nonproximity among the `rectpack` algorithms deployed in our investigation.

E. RUNTIMES

Finally, we will take a quick look at the runtimes for examining all configurations as measured by the `time` module's `process_time` function, which can be gathered from Table 1. Strip packing is the fastest method. The `rectpack` algorithms take longer to execute when rotation is allowed, although this does not guarantee a better outcome for each individual configuration, as we have already seen. On the small chip, all methods terminate within seconds. While for the medium chip the corner heuristic delivers its results only a little slower than the other approaches, the gap becomes substantially wider for the large chip. Fortunately, computations can still be handled in less than a day on a single CPU core (we used an AMD Ryzen 7 2700X consumer CPU), and if even larger chips were to be considered, one could easily distribute the work over several cores or machines, as each configuration can be treated independently. Interestingly, the

TABLE 2. Total number of feasible solutions, number of feasible solutions relative to upper bound, and number of solutions exclusive to the respective algorithm for all considered algorithms and all three examined chip sizes.

	16 mm × 16 mm chip			24 mm × 24 mm chip			32 mm × 32 mm chip		
algorithm	#feas.	vs. u.b.	#ex.	#feas.	vs. u.b.	#ex.	#feas.	vs. u.b.	#ex.
upper bound	412	100.0%		4027	100.0%		21150	100.0%	
MaxRects	272	66.0%	3	2670	66.3%	17	15279	72.2%	165
Guillotine	271	65.8%	6	2617	65.0%	10	14824	70.1%	120
Skyline	259	62.9%	0	2837	70.4%	104	15296	72.3%	211
strip packing	230	55.8%	1	2244	55.7%	0	13709	64.8%	0
corner heuristic	247	60.0%	7	2743	68.1%	114	14899	70.4%	468
total	299	72.6%		3086	76.6%		16656	78.8%	

TABLE 3. Maximum and average distances to upper bound (# A7 cores) for all configurations where each algorithm yields a feasible solution (all chip sizes).

	16 mm × 16 mm		24 mm × 24 mm		32 mm × 32 mm	
algorithm	max.	avg.	max.	avg.	max.	avg.
rectpack (MaxRects)	11	7.35	21	12.92	31	18.21
rectpack (Guillotine)	13	7.16	20	12.97	29	18.69
rectpack (Skyline)	12	7.21	26	12.36	37	18.05
strip packing	19	10.93	42	21.73	67	30.03
corner heuristic	16	8.39	22	10.37	37	13.93

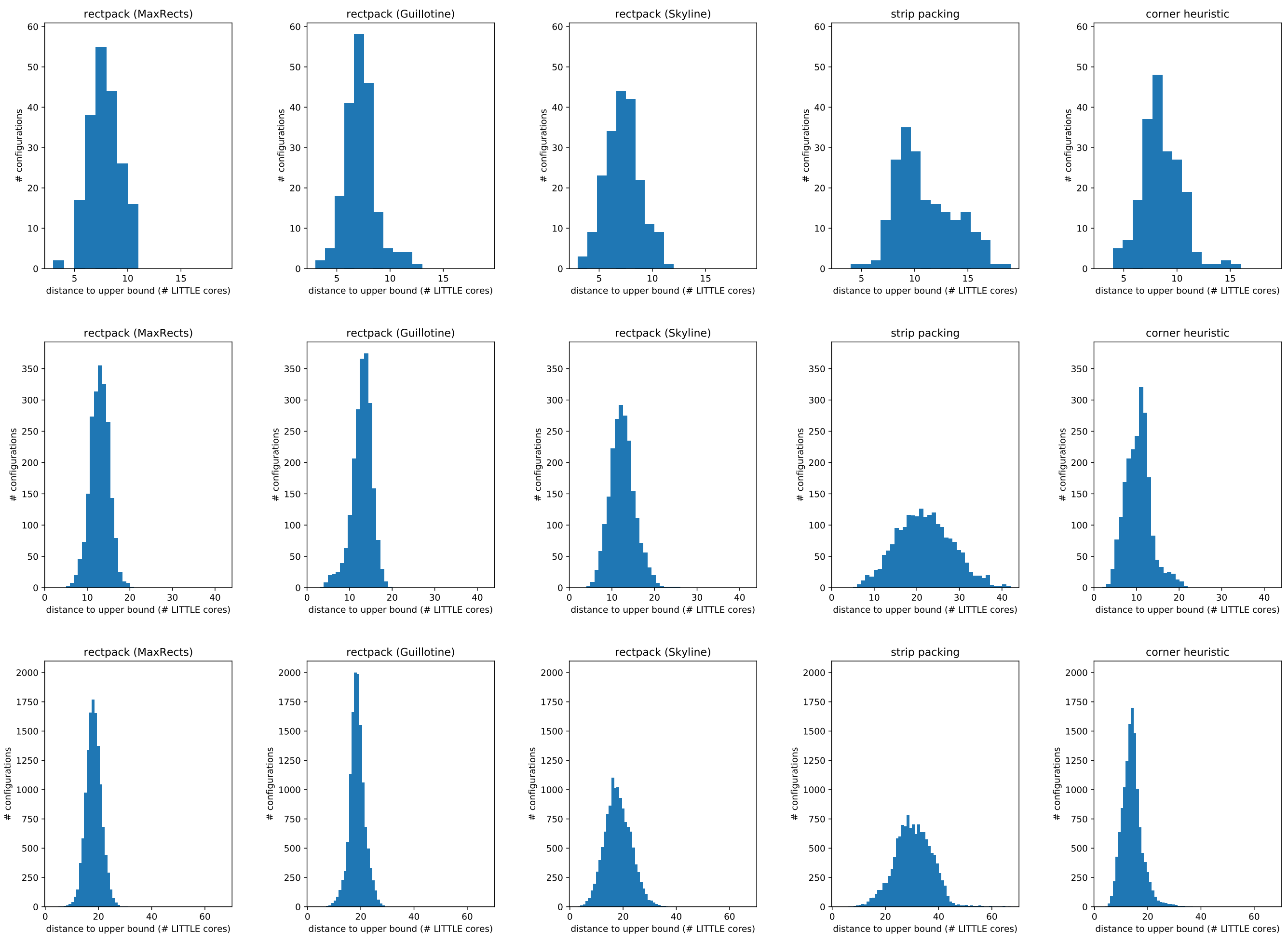

FIGURE 6. Distribution of distances to upper bound (# A7 cores) for all configurations where each algorithm yields a feasible solution. Top row: 16 mm × 16 mm chip, center row: 24 mm × 24 mm chip, bottom row: 32 mm × 32 mm chip.

corner heuristic's runtime is heavily influenced by the placement order as well, where employing a random placement order takes nearly twice the time to complete on the large chip than placing cores in descending order of core size.

All in all, the corner heuristic can compete with algorithms which are directed at the more general problem of rectangle packing in terms of the number of feasible solutions, and the solution quality it produces when a feasible solution is found surpasses that of the other algorithms considered here. Moreover, its solutions possess a property desirable in our context: placing cores of the same type next to each other.

V. CONCLUSION AND FUTURE WORK

For design space exploration in heterogeneous multi-core chip design for a given application, it is important to efficiently determine if a certain allocation of pre-layouted cores drawn from a small constant number K of different core types is feasible or not for accommodation on a chip of given dimensions, where cores of same type should be kept contiguous. For the underlying Pareto-optimal contiguous K-types rectangle packing problem, we have presented the corner heuristic for $K \leq 4$ core types to generate Pareto-optimal core allocations and their packings within a given rectangular chip area. We have shown that the heuristic is competitive with core-type agnostic state-of-the-art rectangle packing techniques while keeping cores of same type close together. We also have shown how to generalize the corner packing heuristic to $K > 4$ core types (onion packing heuristic) and to taking restricted rotation of core rectangles into account. Implementation and evaluation of these extensions will be the subject of future work. Furthermore, we will consider non-rectangular core layouts, non-convex chip areas available for placing cores, core packing taking un-core placement or maximum wire length constraints into account, or packing in 3 dimensions.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their valuable contributions to improve their article.

REFERENCES

[1] P. Marwedel, *Embedded System Design*. Cham, Switzerland: Springer, 2010.

[2] J. Dean, D. Patterson, and C. Young, "A new golden age in computer architecture: Empowering the machine-learning revolution," *IEEE Micro*, vol. 38, no. 2, pp. 21–29, Mar. 2018.

[3] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," *Commun. ACM*, vol. 62, no. 2, pp. 48–60, Jan. 2019.

[4] J. Keller, S. Litzinger, and C. Kessler, "Integrating energy-optimizing scheduling of moldable streaming tasks with design space exploration for multiple core types on configurable platforms," *J. Signal Process. Syst.*, vol. 94, no. 9, pp. 849–864, Sep. 2022.

[5] C. Kessler, J. Keller, and S. Litzinger, "Temperature-aware energy-optimal scheduling of moldable streaming tasks onto 2D-mesh-based many-core CPUs with DVFS," in *Job Scheduling Strategies for Parallel Processing*, D. Klusáček, W. Cirne, and G. P. Rodrigo, Eds. Cham, Switzerland: Springer, 2021, pp. 168–189.

[6] T. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout*. Leipzig, German: Teubner, 1990.

[7] N. Khan, J. Castro-Godinez, S. Xue, J. Henkel, and J. Becker, "Automatic floorplanning and standalone generation of bitstream-level IP cores," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 29, no. 1, pp. 38–50, Jan. 2021.

[8] I. Shallari, I. S. Leal, S. Krug, A. Jantsch, and M. O'Nils, "Design space exploration for an IoT node: Trade-offs in processing and communication," *IEEE Access*, vol. 9, pp. 65078–65090, 2021.

[9] J. Keller, S. Litzinger, and C. W. Kessler, "Combining design space exploration with task scheduling of moldable streaming tasks on reconfigurable platforms," in *Proc. 17th Int. Symp. Appl. Reconfigurable Comput.*, vol. 12700. Cham, Switzerland: Springer, 2021, pp. 93–107.

[10] B. S. Baker, E. G. Coffman Jr., and R. L. Rivest, "Orthogonal packings in two dimensions," *SIAM J. Comput.*, vol. 9, no. 4, pp. 846–855, 1980.

[11] H. Murata, K. Fujiyoshi, S. Nakatake, and Y. Kajitani, "VLSI module placement based on rectangle-packing by the sequence-pair," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 15, no. 12, pp. 1518–1524, Dec. 1996.

[12] C. Kenyon and E. Remila, "Approximate strip packing," in *Proc. 37th Conf. Found. Comput. Sci.*, Oct. 1996, pp. 31–36.

[13] E. Huang and R. E. Korf, "New improvements in optimal rectangle packing," in *Proc. 21st Int. Conf. Artif. Intell. (IJCAI)*, 2009, pp. 511–516.

[14] D. Du, Q. Liu, X. Jiang, Y. Xia, B. Zang, and H. Chen, "Serverless computing on heterogeneous computers," in *Proc. 27th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Feb. 2022, pp. 797–813.

[15] W. Huang, D. Chen, and R. Xu, "A new heuristic algorithm for rectangle packing," *Comput. Oper. Res.*, vol. 34, no. 11, pp. 3270–3280, Nov. 2007.

[16] J. Jylänki. (2010). *A Thousand Ways to Pack the Bin—A Practical Approach to Two-Dimensional Rectangle Bin Packing*. [Online]. Available: http://pds25.egloos.com/pds/201504/21/98/RectangleBinPack.pdf

[17] L. Wei, Q. Hu, S. C. H. Leung, and N. Zhang, "An improved skyline based heuristic for the 2D strip packing problem and its efficient implementation," *Comput. Oper. Res.*, vol. 80, pp. 113–127, Apr. 2017.

SEBASTIAN LITZINGER received the M.A. degree in philosophy from Eberhard Karls Universität Tübingen, Germany, and the M.Sc. degree in practical computer science from FernUniversität in Hagen, Germany, where he is currently pursuing the Ph.D. degree with the Parallelism and VLSI Group. His research interests include energy-efficient task scheduling for parallel systems, the application of machine learning techniques to scheduling problems, and neural architecture search.

JÖRG KELLER received the Ph.D. degree in computer science from Universität des Saarlandes, Saarbrücken, Germany, in 1992. He is a Professor with the Faculty of Mathematics and Computer Science, FernUniversität in Hagen, Germany. His research interests include energy-efficient parallel computing, security and cryptography, and fault tolerant computing.

CHRISTOPH KESSLER received the Ph.D. degree in computer science from Universität des Saarlandes, Saarbrücken, Germany, in 1994. He is a Professor with the Department of Computer and Information Science (IDA), Linköping University, Linköping, Sweden. His main research interests include parallel computing and compilers, especially models and frameworks for high-level parallel programming, program parallelization, optimization, and code generation.