

Data-Driven Mapping Using Local Patterns

Gayatri Mehta, *Member, IEEE*, Krunal Kumar Patel, Natalie Parde, and Nancy S. Pollard, *Member, IEEE*

Abstract—The problem of mapping a data flow graph onto a reconfigurable architecture has been difficult to solve quickly and optimally. Anytime algorithms have the potential to meet both goals by generating a good solution quickly and improving that solution over time, but they have not been shown to be practical for mapping. The key insight into this paper is that mapping algorithms based on search trees can be accelerated using a database of examples of high quality mappings. The depth of the search tree is reduced by placing patterns of nodes rather than single nodes at each level. The branching factor is reduced by placing patterns only in arrangements present in a dictionary constructed from examples. We present two anytime algorithms that make use of patterns and dictionaries: Anytime A* and Anytime Multiline Tree Rollup. We compare these algorithms to simulated annealing and to results from human mappers playing the online game UNTANGLED. The anytime algorithms outperform simulated annealing and the best game players in the majority of cases, and the combined results from all algorithms provide an informative comparison between architecture choices.

Index Terms—Design automation, placement, reconfigurable architectures.

I. INTRODUCTION

THE PROBLEM of mapping a data flow graph (DFG) onto a reconfigurable architecture has been difficult to solve in a satisfying manner. Fast solutions are needed for design space exploration and rapid development. Optimal solutions are needed to extract the best performance from any given design. Difficulties in meeting these objectives are exacerbated with highly customized architectures, such as coarse grained reconfigurable architectures (CGRAs). The added constraints of these architectures (e.g., limited interconnect) make the problem of mapping a DFG onto an architecture in an efficient manner—or even to find a feasible solution at all—that much more difficult. In fact, the difficulty of mapping onto CGRAs may be one bottleneck to widespread adoption [1]–[4].

Fig. 1 shows an example of the problem. A very simple DFG is shown with four inputs and two outputs. This DFG may be the core of a computationally intensive application, which we wish to run at high efficiency. Suppose that a designer chooses to place this DFG onto a reconfigurable mesh architecture with

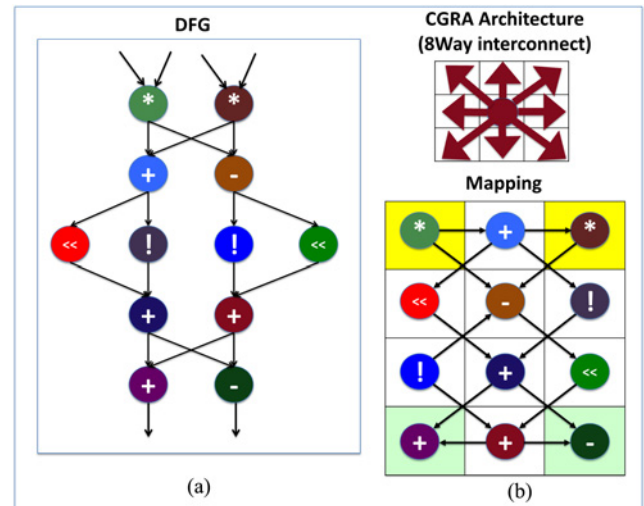


Fig. 1. Problem considered here is to map DFG onto CGRA architecture (top) in an efficient manner. In the example mapping (bottom), inputs are delivered to the yellow shaded elements, and outputs are extracted from the green shaded elements.

eight-way connectivity, as shown in the top of Fig. 1. In this case, each mesh element is a functional unit, which may be programmed as needed for addition, subtraction, etc., and each functional unit is connected only to its eight direct neighbors. The DFG must now be mapped onto this architecture, meaning that each node of the DFG must be assigned to a functional unit such that data can be routed within the constraints of the architecture. This mapping problem can be very difficult due to the limited number of connections between functional units, other constraints such as a limited number of multipliers, and the desire to fit the kernels into very small mesh sizes.

Many CGRA architectures have been proposed. For example, even within a simple mesh design as in Fig. 1, many variations on interconnect have been considered, including connecting to four direct neighbors (4Way), and connecting to neighbors in the same row or column with different numbers of hops (Fig. 5). There is no one clear solution for optimal design, and optimal architecture design will depend on the application domain. Good tools are needed to facilitate exploration of architectural choices [5]–[9].

A typical design flow is to propose an architecture, map benchmark DFGs to that architecture, and collect and analyze results for energy, area, performance, etc. We propose an alternative design flow, where results are made available rapidly and improve over time. In this way, the designer may reach a decision long before the mapping algorithms reach an optimal result. To support this design flow, we propose the use of anytime algorithms [10]–[12].

Manuscript received January 22, 2013; revised April 29, 2013; accepted June 10, 2013. Date of current version October 16, 2013. This work was supported by the National Science Foundation under Grant CCF-1117800 and Grant CCF-1218656. This paper was recommended by Associate Editor D. Chen.

G. Mehta, K. K. Patel, and N. Parde are with the University of North Texas, Denton, TX 76207 USA (e-mail: gayatri.mehta@unt.edu; krunalkumarpatel@my.unt.edu; natalieparde@my.unt.edu).

N. S. Pollard is with the School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213 USA (e-mail: nsp@cs.cmu.edu).

Color versions of one or more of the figures in this paper are available online at <http://ieeexplore.ieee.org>.

Digital Object Identifier 10.1109/TCAD.2013.2272541

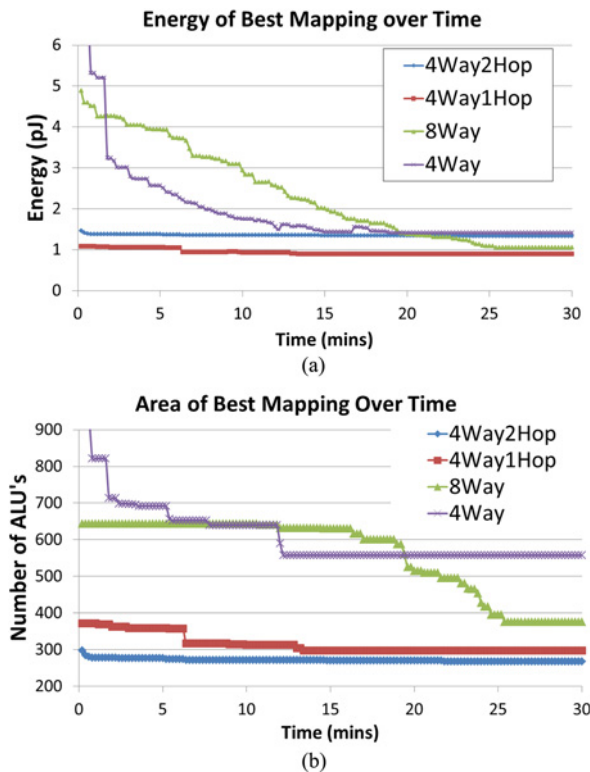


Fig. 2. Visualization of progress of our anytime algorithms. The sum of energy and area over all benchmarks tested are shown for four different architectures.

The goal of an Anytime Algorithm is to generate a good solution quickly and improve that solution over time. In the context of our problem, we seek high quality mappings for a collection of domain benchmarks that improve in quality as more time is available. We present two anytime algorithms to solve the problem of mapping a DFG onto a CGRA architecture—Anytime A*, a greedy algorithm that provides a solution within guaranteed bounds; and Anytime Multiline Tree Rollup, which explores many solution paths and attempts to keep those solution paths diverse to more uniformly explore the space of solutions.

Fig. 2 shows an example of our desired output, and a preview of our results. Here, a user is comparing results from four mesh architectures with different types of interconnects. Results from our algorithms are coming in moment by moment, displayed as cumulative energy and area costs plotted versus time. The most flexible architectures—4Way2Hop and 4Way1Hop—provide near optimal results within seconds, with results improving incrementally over time. The more challenging architectures—8Way and 4Way—provide results that improve more slowly. Even after a few short minutes a user would be able to observe the evolution of results and draw conclusions, perhaps deciding to stop the optimization early, adjust design parameters, and start again. The early availability of results and improvement over time are the desired features of anytime algorithms.

Anytime algorithms have scarcely been explored in the context of mapping. One possible reason is that they can be slow to make progress for search problems having many dead ends

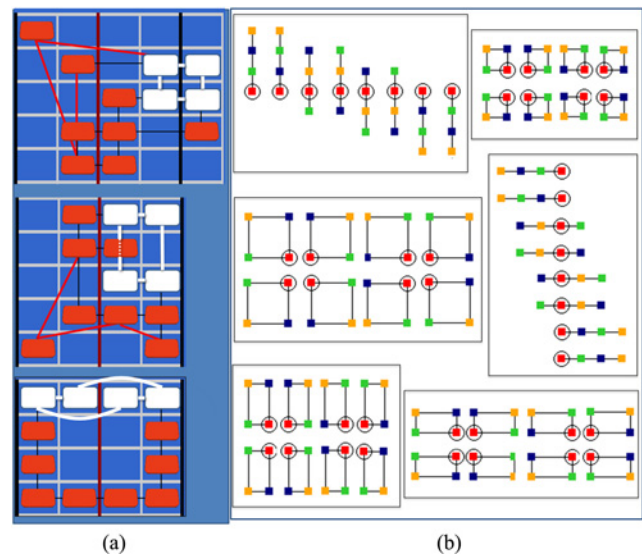


Fig. 3. Left: player is experimenting to find good arrangements of the configuration of nodes shown in white. Right: low cost arrangements for this configuration of nodes from the 4Way1Hop dictionary.

or local minima. We find that this situation is commonplace in mapping. To overcome this problem, we design anytime algorithms that make use of a dictionary of useful patterns. The key result in this paper is that anytime algorithms can be made fast and practical by restricting their search based on a dictionary of desirable arrangements of nodes connected in common local patterns. If data flow graphs can be mapped to an architecture one pattern at a time instead of a single node at a time, we obtain two benefits that complement each other. First, any tree representing a search for a good mapping will have fewer levels, as multiple nodes are placed at every step of the exploration. Second, it will have a relatively small branching factor, as we can take good advantage of our *a priori* knowledge of what relative node placements are known to be useful in creating desirable layouts. We show that these benefits make anytime algorithms practical and open the door to results such as those shown in Fig. 2.

To make this concept more concrete, consider Fig. 3, which shows an example from a player participating in a mapping game called UNTANGLED [13], [14]. This figure shows the player experimenting to find good arrangements for a collection of connected nodes painted white. The nodes are operators in a data flow graph. In fact, they are the top four operators in the DFG of Fig. 1. The player has recognized the connectivity pattern of these four nodes and is attempting various low cost arrangements. Fig. 3 shows the complete collection of 48 frequently observed arrangements for this node configuration, taken from one of our dictionaries (4Way1Hop). Notice how relatively few arrangements are observed to be common in high quality mappings, in contrast to the many arrangements that are possible. We observe that even the use of very simple patterns of a limited number of nodes, such as shown in Fig. 3, can be extremely effective, reducing search times by orders of magnitude.

The following sections of this paper provide details and evaluation of our two anytime algorithms. We compare

performance of our algorithms with and without dictionaries. We evaluate them on a range of architectures and benchmarks by comparing them to a custom version of simulated annealing. We also include results from the top scoring players of the game UNTANGLED as a second basis of comparison. We find that in most cases our algorithm outperforms both simulated annealing and UNTANGLED game players in quality of solution for any computation time, as well as providing an evolving lower bound on quality of the solution.

II. RELATED WORK

The difficulty of the mapping problem has been well discussed in the literature, and most nontrivial formulations are NP complete [15]. Most existing algorithms fall into one of several styles, including greedy algorithms, randomized algorithms, clustering algorithms, integer linear programming, and analytical placement. Comprehensive discussion and further references can be found in the following surveys [16]–[19]. We provide a brief summary here.

Many greedy algorithms have been explored, including deterministic place and route [20], heuristic depth first placement [21], and priority order placement with backtracking [22]. Greedy or heuristic mapping is the option of choice for many mapping problems due to its speed and determinism, but for difficult problems it may perform poorly.

Randomized algorithms are pervasive, successful, and popular, due to their generality and ease of implementation. simulated annealing is frequently used for placement; see VPR [23], SPR [24], RaPiD [25], Dragon [26], and TimberWolf [27]. The main drawbacks of most randomized algorithms are computation time and indeterminism.

Integer linear programming (ILP) has received attention due to its clean representation and the possibility of obtaining an optimal solution. ILP has not been shown to be feasible for large scale mapping problems [28], [29], but it is frequently used as a component of mapping algorithms.

Partitioning or clustering algorithms promise scalability and efficiency even in large graphs, cf SPKM [29], Capo [30], FengShui [31], and NTUplace [32], [33]. Such algorithms can be very fast. However, solution quality may be suboptimal compared to algorithms that do not use divide and conquer [34].

Analytical placers are efficient and scalable, and can achieve good quality results [35], [36]. Analytical placers typically work in a two-step process. A straightforward optimization is solved while allowing placements to overlap. Then, an iterative procedure is used to separate overlapping elements. As with hierarchical clustering, solution quality can suffer due to the local nature of final adjustments. Bian *et al.* [37] investigated the use of the FastPlace analytical placement algorithm [38] for benchmarks having 100 s of thousands of LUTs and concluded that “simulated annealing based placement would still be in dominant use for a few more device generations.” As such, we use simulated annealing as the basis for comparison.

The concept of patterns has been used by numerous researchers in the context of instruction set extension and efficient hardware reuse [39], [40]. Patterns are defined as

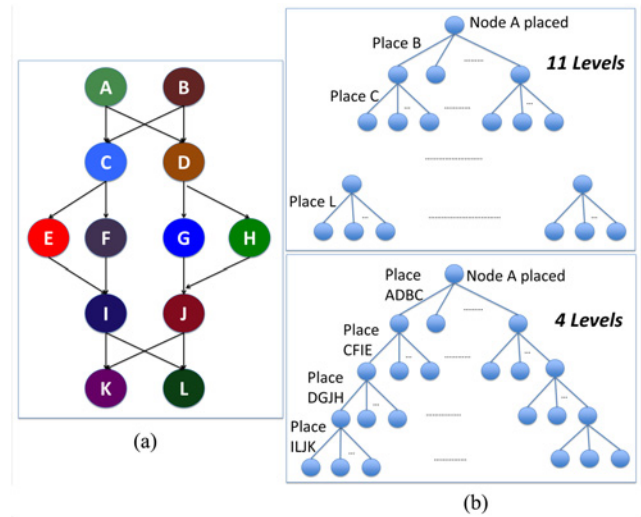


Fig. 4. Simple DFG and associated search trees. Left: test DFG. Top right: search tree that places a single node at a time has 11 levels. Bottom right: search tree that places a complete pattern at a time has only four levels.

subgraphs of a DFG that are identical or nearly identical so that they can easily be mapped onto the same hardware. Early research along these lines made use of component libraries or templates specified by the designer [41]. More recently, automatic pattern identification has been addressed by numerous researchers ([40] and the references therein). Of particular interest are techniques that take into account frequency of pattern occurrence [42], [43] and algorithms for identifying frequent patterns in graphs [44], [45]. Cong and his colleagues consider identification of patterns that are not identical, but may be able to use the same hardware using the concept of edit distance [39].

In this paper, we use patterns to index into a dictionary that provides suggestions for good spatial arrangements of nodes onto relatively homogeneous hardware (i.e., a CGRA with a mesh interconnect structure). We avoid the problem of pattern identification by using a simple definition of patterns such that many instances may be found in any DFG. However, our approach may also perform well for a different selection of patterns.

In the AI community, the use of pattern databases to accelerate A* search has a long history [46]–[49]. In this line of research, one constructs a lookup table recording admissible estimates of cost to go. To make this technique feasible, the lookup table is created using an abstraction of the search space that can be enumerated exhaustively (e.g., corner cube state for the Rubik’s cube puzzle). This technique has been applied to solving permutation style problems, such as Rubik’s cube and the Tower of Hanoi puzzle [46], [50], to action planning [51], and to model checking [52]. However, to our knowledge, pattern databases have not been applied to problems in electronic design automation (EDA) such as placement and routing.

Applying the pattern database approach in its original form is difficult for the mapping problem because the set of goal states is not easily enumerated, making it difficult to construct

the lookup table used in this approach. However, it may be possible to employ techniques from pattern databases to trade off memory for computation time in an approach complementary to our own.

Many authors formulate the mapping problem as we do—as a search through a tree representing potential placements of functional units or clusters. Our search trees and search process differ from previous research as follows. First, we do not do any *a priori* clustering of the graph, but instead treat every node as the center of its own cluster. Second, we present Anytime search algorithms, which are capable of returning a good solution quickly, refine the solution over time, and are complete given sufficient resources (i.e., guaranteed to terminate with a globally optimal solution).

We are the first to our knowledge to create and use a dictionary of frequent node arrangements harvested automatically from a database of high-quality mappings. We show that using a dictionary to place patterns of nodes results in a large decrease in mapping run time and results that are less costly than those of simulated annealing or manual place and route.

III. PROBLEM STATEMENT

The problem that we are solving is defined as follows. We are given a dataflow graph (DFG) and an architectural specification. The dataflow graph is expressed as a set of nodes connected with directed edges. The architecture is expressed as an array of functional units and a lookup table expressing a cost for placing an edge between any pair of functional units. Our objective is to find the lowest cost mapping of the DFG onto the architecture such that each node in the DFG is mapped to a unique functional unit.

We use the cost function from the game UNTANGLED [53] to allow direct comparison of our algorithm results to those from UNTANGLED game play

$$C = \sum_{i=1}^{n_n} \left(\sum_{j=1}^{n_p} I_{i,j} \right) + (N_{op} * 2000) + (N_{pg} * 800) + (N_{nop} * 400) \quad (1)$$

where n_n is the number of nodes, n_p is the number of parents of a node, $I_{i,j}$ is the interconnect cost for all edges of the mapping, N_{op} is the number of ALUs performing an operation, N_{pg} is the number of ALUs used as passgates, and N_{nop} is the number of empty ALUs. Cost factors used in 1 were obtained based on power simulations run on a 90-nm ASIC process from Synopsys using the Synopsys PrimeTime-PX tool. Interconnect cost $I_{i,j}$ reflects power requirements of the interconnect for any edge that is a legal edge within the architecture. For edges that are not supported by the architecture, interconnect cost is a heuristic cost, approximately quadratic in edge length, that is used as a guide toward a valid solution. Area is handled indirectly in 1 as follows. Parameter N_{nop} is determined by counting the number of empty ALUs in the smallest rectangular area that contains the ALUs that have already been assigned operations. A larger area solution may have more empty ALUs and more ALUs used as passgates. All else being equal, this solution will thus incur a greater

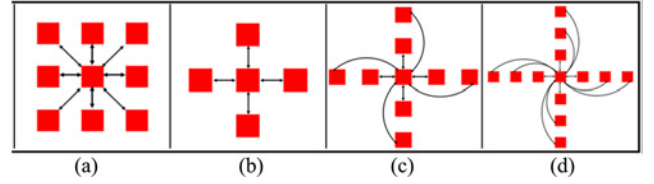


Fig. 5. Interconnect patterns for the architectures considered in this paper. Left to right: 8Way, 4Way, 4Way1Hop, 4Way2Hop.

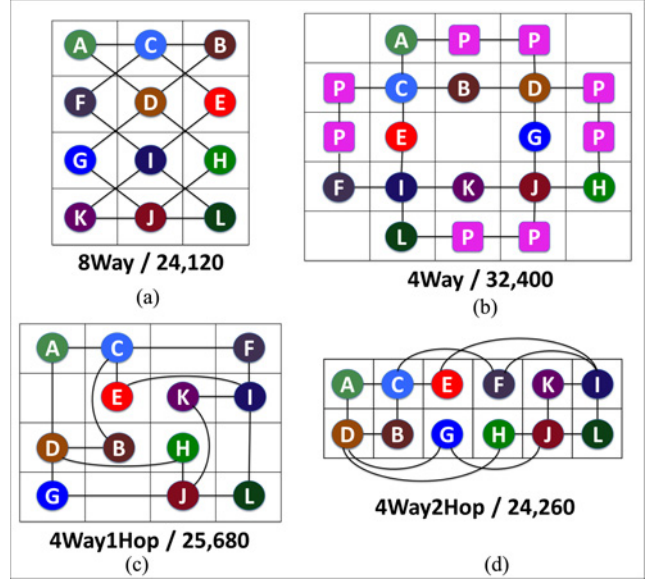


Fig. 6. Test DFG mapped onto all four architectures. The cost of each mapping according to (1) is shown next to the architecture name.

TABLE I
BASIC INFORMATION RELATED TO THE BENCHMARKS

	E1	E1	E3	M1	M2	H1	H2
Nodes	24	29	29	29	36	52	61
Edges	29	29	34	36	53	63	72

cost than a more compact solution. If a better cost function is available to replace 1, it can be substituted trivially.

Fig. 4 shows our simple test DFG from Fig. 1 with operations abstracted away as letters. We consider mesh architectures 8Way, 4Way, 4Way1Hop, and 4Way2Hop, as shown in Fig. 5. Fig. 6 shows mappings of our test graph onto all of these architectures, along with the cost of each mapping. The mappings onto 8Way and 4Way2Hop are the most compact, using the minimum 12 nodes, and have the lowest scores. The mapping onto 4Way is the least compact, has the highest score, and requires the use of eight passgates for routing.

We test our algorithms on seven actual benchmarks, identical to those used in the UNTANGLED game [53]. These are the Sobel (E1) and Laplace (E2) edge detection benchmarks, as well as GSM (E3), ADPCM decoder (M1), ADPCM encoder (M2), IDCT row (H1), and IDCT col (H2) benchmarks from the MediaBench suite. Basic information for the benchmarks is given in Table I.

IV. SIMPLE EXAMPLE

For an overview of our approach, consider again the simple DFG shown in Fig. 4. This DFG begins with operations A and B and concludes with operations K and L. The top part in Fig. 4 shows a standard search tree, which represents the complete set of mappings that can be generated by placing one node of the DFG at a time. Root node A is assigned to some viable functional unit, followed by node B, node C, etc. An edge in this tree represents a placement of a node relative to its parent in the tree. We must plan for routing at the same time as we place nodes, due to the severe interconnect constraints of the architecture [1], [3]. Thus, to traverse one level of the tree, we must not only select an edge, but also provide a routing solution to connect the newly placed node to all of its parents and children from the DFG who have already been placed.

We can estimate the size of the search tree for our architectures. For 4Way1Hop we consider the 32 closest placements of a node relative to its parent. This choice results in a branching factor of 32 at 11 levels for 10^{16} possibilities. Our Anytime A* algorithm finds the optimal solution in 57 s, but cannot prove that it is optimal even after 90 min of runtime.

Bottom in Fig. 4 is our proposed search tree, where every level represents placement and routing of a collection of nodes. For this sample DFG, our algorithms identify patterns of square topology, which are repeated four times to form a large loop, respectively, ADBC, CFIE, DGJH, and ILJK. Our dictionary for 4Way1Hop has 48 low cost arrangements for node configurations having a square topology [Fig. 3(b)]. The corresponding search tree has a branching factor of 48, giving 5.3 million possibilities. Compared to the original search tree, the search tree based on patterns has more than nine orders of magnitude fewer potential mappings. For the 4Way1Hop architecture, Anytime A* returns the optimal result in 0.06 s and proves that it is optimal within 3 s. This is a speedup of three orders of magnitude in each case and makes computation times practical for CGRA problems.

V. ANYTIME A*

The first algorithm we present is Anytime A*. A* algorithms are efficient in the sense that no algorithm can guarantee an optimal result by expanding fewer nodes of the search tree [54]. They are appealing because they can be designed to give results that are optimal within some fixed bound, allowing loosening of optimality guarantees in return for speed. This property is the basis for our anytime algorithm.

We begin by reviewing basic A* [54], [55]. A* is a tree search algorithm that expands nodes of a search tree in a best-first manner. It obtains its rigor through the use of admissible cost estimates, which are guaranteed to never overestimate the minimum cost of a complete solution built upon a partially expanded state.

In our case, the search tree consists of nodes representing partial mappings of a DFG to an architecture, and edges specifying potential row and column locations for the next nodes to be placed (Fig. 4). We define a state as a partial mapping, including placement and routing of some subset of nodes. A state is created by traversing a portion of the

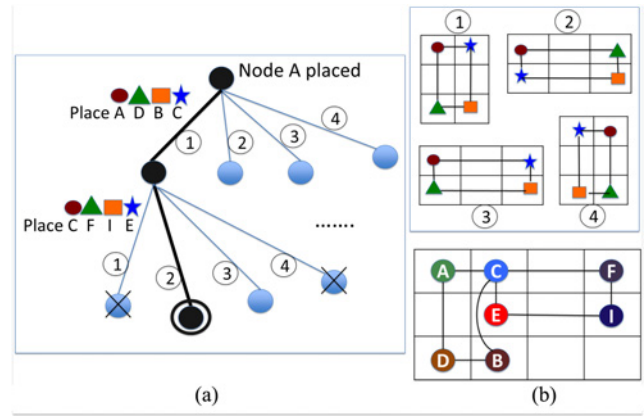


Fig. 7. Left: a portion of the search tree shown in the bottom, right portion of the figure. Top right: four of the 48 low-cost arrangements of nodes are shown. Bottom right: partial mapping that corresponds to the tree traversal highlighted in the left figure. This is an expansion state of the search tree and is represented by the circled node in the left figure.

search tree. For example, Fig. 7 shows a traversal of two levels of a search tree, selecting arrangement 1, followed by arrangement 2. The arrangements are shown with their labels in the top of Fig. 7. Bottom in Fig. 7 shows the partial mapping encoded by this tree traversal. This result is a successful partial mapping, the start of the complete mapping shown in Fig. 6 for 4Way1Hop. Note that after arrangement 1 is selected, a portion of the tree can be pruned. If ADBC is placed using arrangement 1, it is impossible to place CFIE using either arrangements 1 or 4. Both choices would result in two nodes being assigned to the same functional unit. In the first case, nodes B and F would be assigned the same row and column. In the second case, node pairs B,F and D,I and A,E would overlap. Because overlap of nodes is not allowed, these states need not be considered further.

For each state s , which is a partially expanded tree as shown in Fig. 7, the A* algorithm maintains the following three parameters.

- 1) $C(s)$: cost of placing and routing the DFG so far;
- 2) $G(s)$: cost-to-go, an admissible estimate of cost to complete the mapping;
- 3) $Q(s)$: cost estimate, an estimate of the minimum cost mapping that can be obtained from state s .

For basic A*, Q is the sum of the actual cost plus the estimated cost-to-go

$$Q(s) = C(s) + G(s). \quad (2)$$

Because $G(s)$ is admissible, expanding the node with the minimum $Q(s)$ at each step guarantees that the first complete mapping found will be a global optimum.

A. Weighted A*

A* can be slow to terminate, especially in the case where cost-to-go may be substantially underestimated. One way to address this problem is to relax the requirement to obtain a global optimum. Instead, we can search for a result that is some fraction of optimal, i.e., the actual cost of the solution

found is guaranteed to be not greater than the actual cost of the optimal mapping inflated by a factor $(1 + \epsilon)$

$$C(s_{\text{solution}}) \leq C(s_{\text{optimal}}) * (1 + \epsilon). \quad (3)$$

The Weighted A* algorithm achieves this result by inflating the cost-to-go estimate G by factor $(1 + \epsilon)$

$$Q(s) = C(s) + G(s) * (1 + \epsilon). \quad (4)$$

The result in (3) follows trivially [12]. The practical effect of increasing the inflation factor $(1 + \epsilon)$ is to encourage expanding nodes at a greater depth in the tree, thus enabling a solution to be found much more quickly.

Fig. 8 shows an example that illustrates weighted A*. A highly simplified search tree is shown in the center of the figure. The search consists of placing nodes ADBC using arrangement 1 or 3, then CFIE using either arrangement 2 or 3, followed by DGJH using 2 or 4, and finally ILJK using 2 or 4. The four arrangements are shown schematically at the top of Fig. 8.

The search begins by placing the single node A, which incurs a cost of $C = 2000$ according to the cost function in 1. Weighted A* search is performed with $\epsilon = 2$ so that $Q = C + 2G$. Traversing edges with lowest Q first, breaking ties to the left, results in expansion order a,c,e, then g, with Q values of 42K, 39.2K, 34.8K, and 36K, respectively. The suboptimal solution at leaf node g [Fig. 8(a), bottom] is reached very quickly, giving the user some early information as the search proceeds. The algorithm will then continue to explore the tree, returning the optimal solution at node h [Fig. 8(b), bottom] later in the search process. For $\epsilon = 1$ (basic A*), the traversal order would have been a,b,c,d,e,f,h. In other words, most of the search tree would have been traversed before finally reaching a global optimum.

B. Anytime A*

Our Anytime A* algorithm is constructed from Weighted A*. Initially, a large ϵ is used to encourage finding some solutions. After each solution is found, it is presented to the user, ϵ is reduced, and the search continues.

Algorithm 1 gives pseudocode for our Anytime A* mapping algorithm. The algorithm begins with the *OPEN* set initialized to a set of potential start states s_{start} . Each start state contains a row and column placement for the root node of the search tree. Start states should be created to place the root node in each viable grid location in the reconfigurable design. The algorithm is further initialized by setting ϵ to the *startCostFactor*.

The anytime algorithm continues until either it is stopped by the user or the optimal solution has been found (*OPEN* is the empty set). While *OPEN* is not empty, the algorithm iteratively processes the state s in *OPEN* with the smallest estimated cost $Q(s)$, removing s from *OPEN*.

If s is a complete mapping, a solution has been found that is within factor $(1 + \epsilon)$ of optimal. In this case, the algorithm notifies the user, adds state s to the solution set, and prepares for the next iteration. The next two steps are the critical ones, which make this algorithm an effective anytime algorithm. First, ϵ is reduced to give a tighter bound by multiplying it

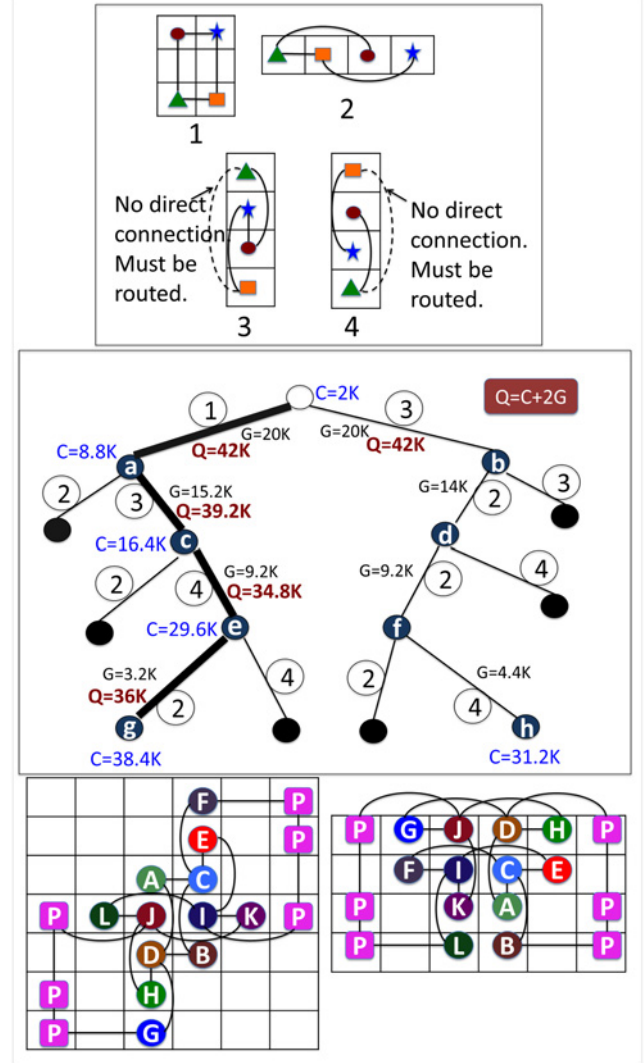


Fig. 8. Full worked-out example that illustrates Weighted A*, whereas a standard A* search tree would be nearly fully expanded to reveal the optimal solution shown on the bottom right, the weighted A* search tree (center) is traversed in a more depth-first manner to reveal a suboptimal solution (bottom left) early in the search process.

with factor *decay*. The estimated cost $Q(s)$ is then recomputed for every state s in the *OPEN* set, and the algorithm proceeds.

If s is not a complete mapping, it is expanded to the next level. First, successor states s' are created by placing the next unplaced pattern in all arrangements listed in the dictionary. Each successor s' is considered in turn. If s' can be successfully placed and routed, its actual cost $C(s')$ is computed, estimated cost $Q(s')$ is updated, state s' is added to the *OPEN* set, and the algorithm proceeds.

The practical effect of this algorithm is to return many solutions early on, while parameter ϵ is high, some of which are quite good or even optimal, and then to slow at points where it becomes more difficult to prove that the solution is within a factor of $(1 + \epsilon)$ of the optimal mapping. The algorithm is guaranteed to terminate either with an optimal solution or to inform the user that no solution is possible. However, sufficient information to proceed may be available well before that time.

Algorithm 1: Anytime A*

Data: *startCostFactor*, *decay*, s_{start}
Result: set of mappings, contained in *solutions*
 $OPEN = \{s_{start}\};$
 $solutions = \{\};$
 $\epsilon = startCostFactor;$
while $OPEN \neq \{\}$ **do**
 remove s with smallest $Q(s)$ from $OPEN$;
 if s is a complete mapping **then**
 $solutions = solutions \cup \{s\};$
 notify user of new solution;
 $\epsilon = \epsilon * decay;$
 foreach s_i in $OPEN$ **do**
 $Q(s_i) = C(s_i) + G(s_i) * (1 + \epsilon);$
 end
 else
 foreach successor s' of s **do**
 place and route graph through the level of s' ;
 compute actual cost $C(s')$;
 if place and route was successful **then**
 update $G(s')$;
 $Q(s') = C(s') + G(s') * (1 + \epsilon);$
 $OPEN = OPEN \cup \{s'\};$
 end
 end
 end
end
return *solutions*;

In practice, we perform Anytime A* search beginning with ϵ at 10 and using a decay factor *decay* of 0.98.

C. Admissible Estimate of Cost-to-Go

An admissible estimate of cost-to-go $G(s)$ is obtained by summing the minimum possible cost for each level over all remaining levels of the search tree as follows:

$$G(s) = \sum_{i=l+1}^L (\min_{a \in A_i} (g(i, a))) \quad (5)$$

where l is the level of the search tree that has been traversed to reach state s , L is the leaf level of the search tree, A_i is the set of arrangements available at level i , and $g(i, a)$ is a lower bound on the cost of placing the new nodes at level i in arrangement a and doing the minimal routing required to connect them to each other and to the graph. Parameter $g(i, a)$ can further be expressed as follows:

$$g(i, a) = N_{newOps}(i) * 1600 + N_{newPG}(i, a) * 400 \quad (6)$$

where $N_{newOps}(l)$ is the number of new nodes that must be placed at level l , and $N_{newPG}(l, a)$ is the minimum number of passgates that will be needed to connect these nodes to each other and to the nodes that have already been placed. The cost values can be explained as follows. The cost of an operator is 2000, based on (1). In the lowest cost case, this operator will be placed where there was a no-op before, which would have had a cost of 400. Thus, the minimum

additional cost of placing this operator is 1600. The same argument holds for each passgate, which has a cost of 800 and may replace a no-op having cost 400. Many different cost functions are possible. As long as the cost is guaranteed to be a lower bound on the actual cost added by placing and routing the new nodes, the estimate will be admissible.¹

VI. MULTILINE TREE ROLLUP

Anytime A* is effective for most of the scenarios that we tested. However, it is sometimes slow to return a first solution, even in the case where many solutions exist. A typical case is when the A* algorithm would have to prove that all of the best looking solutions are going to fail before attempting a second class of solutions with higher estimated cost, but which were able to succeed. To provide more rapid results in cases such as this one, we designed the Multiline Tree Rollup algorithm.

The goal of this algorithm is to simultaneously explore multiple paths through the search tree. This approach differs from a pure randomized search, because when the collection of solutions becomes too large, we do a downselect in a principled manner to attempt to maintain diversity.

Pseudocode for the basic Multiline Tree Rollup algorithm is shown in Algorithm 2 (MTRollup). At each step, the algorithm fully expands all of the states in $OPEN$ to the next level of the tree. If the expanded $OPEN$ set is larger than the maximum population size, it is pruned to that population size in a manner that is designed to retain diversity.

More precisely, the algorithm begins with the same $OPEN$ set as Anytime A*. It then proceeds level by level until the tree has been expanded to its complete depth. At each level, each state s in $OPEN$ is considered in turn. All successors s' of s are identified by looking up in the dictionary all arrangements for the node pattern at the next level of the search tree. We attempt to place and route each successor s' in turn. If the place and route is successful, the actual cost of the result, $C(s')$, is computed, and state s' is added to set $NEXT$.

Once all states s in $OPEN$ have been processed in this manner, we check the size of $NEXT$. If it is greater than the population size, it is pruned to the population size in a manner to be discussed shortly. Following this pruning, $OPEN$ is set to $NEXT$, and the algorithm proceeds to the next level.

A. Pruning

When pruning, our goal is to maintain a diverse population of solutions. For some problems, finding the optimal solution can be very much like finding a needle in a haystack in the sense that the best solutions may be rare, and they may not first appear to be the most promising. For this reason, and to also avoid sampling many solutions that may be very close to

¹One improvement to $g(i, a)$ as given in (6) would be to add a lower bound estimate of the cost of no-ops that will be added due to any unavoidable introduction of a new row or column. Implementing this solution, however, requires great care to correctly estimate the minimal number of new no-ops that must be added in this fashion without undertaking the substantial work of expanding the entire search tree to determine exactly which rows and columns are and are not occupied.

Algorithm 2: MTRollup

```

Data:  $P, s_{start}, discardFraction, \alpha$ 
Result: set of complete solutions contained in  $OPEN$ 
 $OPEN = \{s_{start}\};$ 
 $solutions = \{\};$ 
for  $i$  from 1 to the number of search tree levels do
     $NEXT = \{\};$ 
    foreach  $s \in OPEN$  do
        foreach successor  $s'$  of  $s$  do
            place and route graph through level of  $s'$ ;
            if place and route was successful then
                compute actual cost  $C(s')$ ;
                 $NEXT = NEXT \cup s'$ ;
            end
        end
    end
    if  $NEXT.Count > P$  then
        PruneMappings( $NEXT, P, discardFraction, \alpha$ );
    end
     $OPEN = NEXT;$ 
end
return  $OPEN;$ 

```

(or symmetrical versions of) each other, we wish to give the algorithm a sampling of solutions that are different in cost, and thus perhaps diverse in character.

Our pruning algorithm follows that in [56], and proceeds as follows. We begin with a set of states *OPEN* that we wish to prune to a population size of P . If there are more than P states in *OPEN*, we first sort them based on cost $C(s)$, and then discard a fraction *discardFraction* of the highest cost solutions. We then generate a set of P numbers $\{x_1, \dots, x_P\}$, which are evenly spaced in the range from 0 to 1 and obtain P target cost values V as follows:

$$V_i = C_{min} + (C_{max} - C_{min}) * x_i^\alpha \quad (7)$$

where C_{min} and C_{max} express the range of cost values remaining. The value of α determines how much pruning favors lower cost solutions. Higher α means that more of the lowest cost solutions are taken, and α tending toward 1 maintains a more equal distribution of scores. In our experiments, we use a *discardFraction* of 0.4 and an α value of 3. The algorithm proceeds from V_1 through V_P , in each case removing from *OPEN* the state with the cost closest to V_i and placing that state in a new set *NEW*. Finally, when states have been selected based on all cost targets, the resulting set *NEW* is returned.

B. Anytime Multiline Algorithm

The Multiline algorithm can be slow to run for large population sizes. However, it can be made into an anytime algorithm in a straightforward manner. To do this we begin with small population sizes, expanding the population size by a given factor at each iteration. The Anytime version of this algorithm is given in Algorithm 4. In practice, we use an initial *PopulationSize* of 5 and a *populationScaleFactor* of 1.5. For efficiency, care can be taken with bookkeeping to ensure that results from prior iterations are cached to avoid repetition and to avoid traversing paths that are known to be infeasible.

Algorithm 3: PruneMappings

```

Data:  $OPEN, P, discardFraction, \alpha$ 
Result: pruned solutions in  $OPEN$ 
if  $OPEN.Count \leq P$  then
    return  $OPEN$  ; // no need to prune
else
     $NEW = \{\}$ ;
    sort  $OPEN$  based on actual cost so far  $C(s)$ ;
     $D = discardFraction * OPEN.Count$ ;
    discard the  $D$  highest cost solutions;
    Let  $C_{min}$  be min cost in  $OPEN$ ;
    Let  $C_{max}$  be max cost still in  $OPEN$ ;
    Let  $X = x_1, \dots, x_P$  equally spaced from 0 to 1;
    for  $i$  from 1 to  $P$  do
         $V_i = C_{min} + (C_{max} - C_{min}) * x_i^\alpha$ ;
        get state  $s$  in  $OPEN$  with cost closest to  $V_i$ ;
        remove  $s$  from  $OPEN$ ;
         $NEW = NEW \cup \{s\}$ ;
    end
    return  $NEW$  ;
end

```

Algorithm 4: Anytime Multiline Tree Rollup

```

Data:  $startP$ ,  $scaleP$ ,  $maxTime$ ,  $s_{start}$ ,  $discardFraction$ ,  $\alpha$ 
Result: solution set in  $s$ 
 $P = startP$ ;
 $s = \{\}$ ;
while  $elapsedTime < maxTime$  do
     $s = s \cup MTRollup(P, s_{start}, discardFraction, \alpha)$ ;
    notify user of best solution so far;
     $P = P * scaleP$ ;
end

```

VII. PATTERNS AND DICTIONARIES

As the results will show, neither Anytime A* nor Multiline Tree Rollup is most effective when placing nodes one by one. Instead, we place entire collections of connected nodes at once according to a dictionary of desirable arrangements. This section explains how we build our search tree and how we create dictionaries from a database of good mappings.

We first provide a precise characterization of a node’s local neighborhood. Every node within a DFG has a local neighborhood that includes other nodes connected to it by traversing some number of edges. For all of the examples and results in this paper, we consider as the local neighborhood all nodes that are within distance 2. This distance is just sufficient to capture the square topology node configuration in our test DFG (Fig. 4) and many of our benchmarks. It is small enough that different DFGs share many node configurations with identical topologies. We emphasize that we do not divide the DFG into clusters. Instead, we consider the local neighborhood around every node.

Fig. 9(a) shows an example of a node and its local neighborhood. The central node is shown, along with all nodes that can be reached within distance 2. We divide this local neighborhood into a collection of strongly connected components (SCCs) and isolated edges [Fig. 9(b)]. The SCCs are defined as any collection of nodes where there is a path P from any node n_i to any other node n_j and back to n_i without backtracking along the same edge. Specifically,

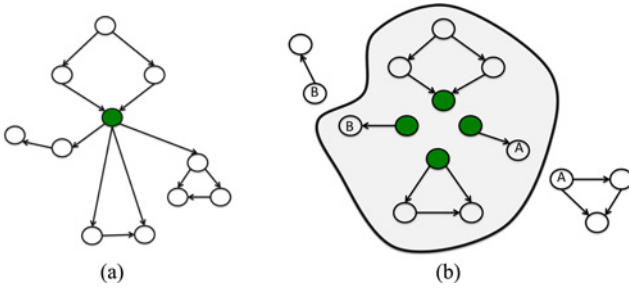


Fig. 9. (a) Example of the local neighborhood surrounding the node in green. All nodes within distance 2 in any direction or combination of directions are considered. (b) This same local neighborhood broken into SCCs and direct connections. We are interested in all components that contain the green node. Each of these components will be arranged relative to the green node according to the arrangements found in our dictionary.

$\forall n_i, n_j \in SCC, \exists P = Path(n_i, n_j, n_i)$ such that consecutive nodes (n_k, n_{k+1}) satisfy either $(n_k, n_{k+1}) \in E$ or $(n_{k+1}, n_k) \in E$, where E is the set of edges in the DFG and if consecutive nodes (n_a, n_b) are found in path P , then path P does not contain consecutive nodes (n_b, n_a) (i.e., no backtracking). Fig. 9 shows three such SCCs. All remaining edges are broken out as isolated edges. For the node's local environment, we retain only SCCs and isolated edges that contain the central node, as shown in Fig. 9(b). In our examples, SCCs identified in this manner ranged from three to six nodes in size.

A. Building Dictionary

We construct a dictionary from a sample mapping by parsing the local neighborhood of every node in the DFG into its components (the SCCs and direct connections), and entering the arrangements of each component into the dictionary. In order to maximize the information obtained from each arrangement and avoid biases, we enter into the dictionary the actual arrangement and the entries that result from permuting it in the following ways: 1) reflect about the x -axis; 2) reflect about the y -axis; 3) reflect about the x - y diagonal; 4) rotate by 90° ; 5) rotate by -90° ; 6) rotate by 90° and mirror about the x -axis; and 7) rotate by -90° and mirror about the x -axis.

We create a separate dictionary for every architecture, as good node arrangements will vary substantially by architecture. For our experiments, we use the results of simulated annealing runs to generate mappings for our dictionary. We note, however, that for most of our test cases, our algorithms can outperform simulated annealing using the dictionary created from simulated annealing results.

We use leave-one-out cross validation for all of our results. The dictionary that is used for benchmark E3, for example, is constructed from simulated annealing runs on all of the benchmarks except E3. This process demonstrates the generality of the method. There is no guarantee that there would be any legal mapping for E3 based on dictionary entries taken from the other benchmarks. We also prune the resulting dictionary for efficiency. For our experiments, we remove from each dictionary arrangements that appear less than one time out of 1000. Creating dictionaries is relatively fast. In our examples, the entire process requires just a few seconds.

The left column of Fig. 10 shows all of the patterns in one of our actual dictionaries (the dictionary created for benchmark































Patterns	Sample Arrangements			
				
				
				
				
				
				
				

Fig. 10. Patterns and sample arrangements from the dictionary used for the M2 benchmark and the 4Way1Hop architecture. The dictionary was constructed from multiple runs of simulated annealing on benchmarks excluding M2. In the sample arrangements, pink squares indicate one possible placement for passgates that are needed for routing in this architecture.

TABLE II
INSTANCES AND ARRANGEMENTS: 4WAY1HOP, M2 DICTIONARY

	A	B	C	D	E	F	G
Inst.	72,000	10,336	18,432	864	864	784	1152
Arr.	28	100	48	200	200	104	208

M2 for 4Way1Hop). Several arrangements are shown for each pattern, along with the number of times this arrangement was observed. The total number of observed instances and arrangements are shown in Table II. Referring to Fig. 10, note that arrangements that require no additional passgates for routing are much more common than those that do require additional routing (patterns A, B, D, and E). Also note that arrangements that leave gaps are sometimes more common than those that do not, perhaps because arrangements with gaps make it easier to fit other nodes that may be connected to this pattern (patterns C and F).

B. Building the Search Tree

Given any new DFG to map to an architecture, we must construct a search tree for that DFG. In general, any node in the DFG could be used as a root, and the graph could be built out in any manner from that root until the search tree specifies placements for all nodes. Our goal, however, is to introduce constraints on the mapping as quickly as possible in order to reduce the number of options (i.e., branches in the tree) that must be explored. We accomplish this goal using an algorithm that adds patterns to the search tree in a greedy manner that closes the next loop as quickly as possible. Each time we close a loop it creates a constraint that removes many possibilities from the search tree, focusing our search.

The largest pattern in the DFG, i.e., the pattern with the greatest number of nodes is selected as the first pattern to be placed. We build from that root one pattern at a time, until all

nodes of the DFG have been covered. At each step, we give priority to: 1) the largest pattern that connects to two different nodes in the existing graph; 2) the shortest sequence of patterns that results in closing a loop; and 3) the pattern connecting to the earliest/oldest part of the graph, breaking ties by adding the largest patterns first. Building out a DFG in this order has empirically been more effective than either a random order or a greedy order that simply takes the largest patterns first.

VIII. RESULTS

We compare results of our two anytime algorithms to simulated annealing and to the maximum scores of UNTANGLED players. In each case, the simulated annealing algorithm was allowed to run to completion, using the settings specified in [23]. We ran simulated annealing 30 times for each of the 28 cases (seven benchmarks \times four architectures), and plot the mean cost from simulated annealing with error bars showing one standard deviation. To make the other algorithms consistent with simulated annealing running times, we terminated them after 1 h. Notably, Anytime A* did not run to completion, and the resulting solution is not guaranteed to be optimal. However, we expect that in many cases the results are optimal or very close.

There is a baseline cost required simply to place the operators in each architecture. Fig. 11 shows the costs incurred for each solution above that baseline. If an algorithm did not return a solution in the allotted time, this is noted with an X on the zero cost line.

Considering the architectures in turn, for 4Way2Hop, one or both of the anytime algorithms returned a near optimal solution for all benchmarks, giving results much better than simulated annealing in all cases except two (Anytime A* on the E2 and M2 benchmarks). The anytime algorithms also outperformed the UNTANGLED players for the more difficult benchmarks. Anytime A* did not return a solution in the allotted time for benchmark M2. However, Multiline Layout uncovered a near-optimal solution almost immediately. Taking the anytime algorithm results cumulatively, on average the best Anytime result was 13% of the mean simulated annealing result and 73% of that of the best player.

For 4Way1Hop, at least one of the anytime algorithms outperformed simulated annealing by a significant margin for all benchmarks and outperformed players for the more difficult benchmarks. On average, the best Anytime result was 27% of the mean simulated annealing result and 68% of that of the best player.

For 8Way, all algorithms performed well for the easier benchmarks, but the anytime algorithms had difficulty on the more difficult benchmarks, in four cases not returning a solution after 1 h. No algorithms outperformed the UNTANGLED players for any of the benchmarks in 8Way, except for simulated annealing on benchmark H1. When at least one anytime algorithm returned a result (all but one case), the anytime algorithms performed, on average, at 149% the value of the mean simulated annealing result and 413% the value of the best players.

For 4Way, all solutions were costly compared to the theoretical minimum, and no one algorithm dominates. anytime

algorithms did not return a solution in the allotted time for four cases. When at least one anytime algorithm returned a result (all but one case), on average the best Anytime result was 130% of the mean simulated annealing result and 145% of that of the best player.

One feature of the anytime algorithms is that they can often return a good solution extremely quickly. Fig. 12 shows the time required by each algorithm to reach a solution that is not more than 110% of optimal. For 4Way2Hop, one or both anytime algorithms returned a near optimal solution in less than 5 s for all benchmarks. For this architecture, we would have an excellent idea of the final cost of this benchmark suite after just a few seconds. Obtaining good results from simulated annealing would require a much longer wait. For 4Way1Hop, one or both of the anytime algorithms returns a near optimal solution in less than 10 s in five cases. The remaining two cases required 7 min (H2) and 10 min (H1) to return a solution of this quality. simulated annealing in contrast is able to beat the 10-min mark in only two cases. For 8Way and 4Way, the anytime algorithms return near optimal solutions very quickly for the easier benchmarks, but perform poorly for the more difficult benchmarks.

Cost values from the three algorithms (Anytime A*, Anytime Multiline Tree Rollup, and simulated annealing) and from all of the benchmarks can be combined to provide an aggregate summary of cost that evolves over time, stock ticker style. Fig. 2 shows these results after interconnect has been included and cost values have been converted into common units of picoJoules.² At each timestep, we have taken the minimum cost identified by any of the three algorithms and summed these cost values over all benchmarks. From the plots in the figure, we can see that results are clear for 4Way2Hop in seconds, for 4Way1Hop in minutes, and require longer for 8Way and 4Way. Energy required for 8Way approaches that of 4Way1Hop. However, the 8Way architecture requires more area than 4Way1Hop, based on the best mappings identified by our algorithms. We see very quickly that 4Way2Hop, despite allowing for more compact mappings, has too much energy overhead to be competitive. We also see that 4Way comes in at an energy cost similar to 4Way2Hop and vastly increased area requirements. We also see that for our algorithms, 4Way2Hop, followed by 4Way1Hop represent easy mapping problems, and 4Way and 8Way are substantially more difficult. The qualitative results we see in this comparison of different architectures are consistent with those found in previous research [6], [57]–[59], notably that 4Way1Hop is a substantial improvement over 4Way, 4Way2Hop has too much overhead to justify its ability to more compactly fit many DFGs, and 8Way performs more poorly than one might expect, given its similarity to the 4Way1Hop architecture. Regarding the 8Way architecture, we note that UNTANGLED players substantially outperformed our anytime algorithms and simulated annealing in a number of cases. Specifically, for the E1, M2, and H2 benchmarks,

²Energy overhead related to interconnect was obtained based on power simulations run on a 90-nm ASIC process from Synopsys using Synopsys PrimeTime-PX tool.

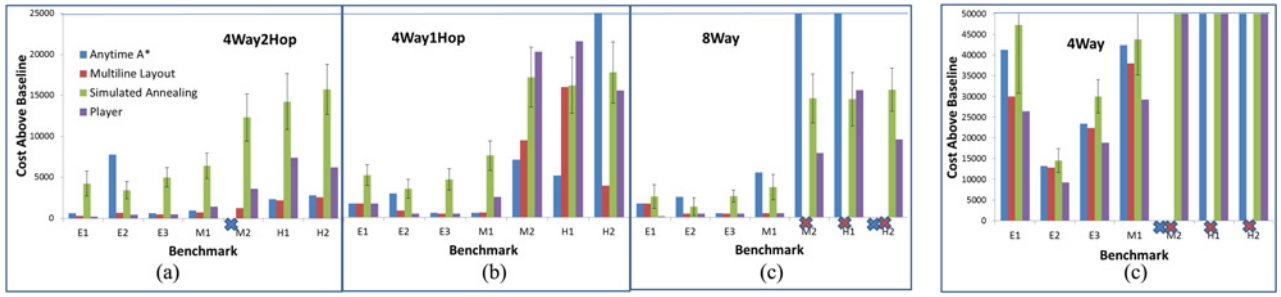


Fig. 11. Cost values for all algorithms, benchmarks, and architectures tested. If an algorithm did not return a solution after 1 h, it is marked with an X. The best UNTANGLED player's cost for each architecture and benchmark is provided for comparison. Note the different scale for the 4Way architecture, which results in substantially higher cost values than the other architectures.

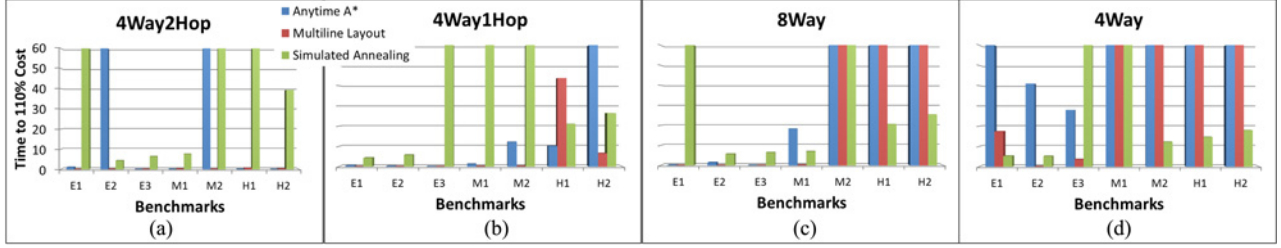


Fig. 12. Time required before we have a solution within 110% of the known optimal solution for all algorithms, benchmarks, and architectures tested. Times of 60 min indicate that the algorithm did not reach a solution within 110% after 1 h of runtime.

player costs are 8%, 54%, and 61% of the costs of the best algorithm result (Fig. 11). Perhaps, there is room for improved algorithms to make the 8Way architecture more attractive.

IX. DISCUSSION

In summary, we have presented two anytime algorithms that can be applied to the problem of mapping a DFG onto an architecture. The first is Anytime A*. A* algorithms are appealing in part because they have been shown to be optimal in identifying the best possible solution for a graph search given all of the information we have in the form of the search tree itself and our estimates of minimum cost to go [54]. Our Anytime A* algorithm makes this approach practical as well, by giving a user information to work with early in the search process and by ensuring that that information will improve over time. In addition, Anytime A* provides a guaranteed lower bound on the cost of solution that can improve with each new solution identified. We have shown that Anytime A* can often produce results within seconds that have less cost than results from simulated annealing and than our UNTANGLED players without losing the guarantees that come from having a complete algorithm.

In some cases, however, our Anytime A* algorithm falls into a local minimum that must be filled before it can identify an improved (or any) solution. Anytime Multiline Tree Rollup was presented as an alternative to protect against this situation. Our Anytime Multiline Tree Rollup algorithm strives to maintain a distribution of low cost solutions to avoid being stopped by any single apparently attractive local minimum. As with Anytime A*, more potential solutions are explored as more time is available, and the algorithm is complete in the sense that given sufficient time and memory the entire search tree will eventually be expanded. Our Anytime Multiline Tree

Rollup algorithm worked well in conjunction with Anytime A*, with one algorithm often providing quick results in situations where the other could not. All three algorithms (Anytime A*, Multiline Tree Rollup, and simulated annealing) can be combined to obtain best available results that improve over time, as shown in Fig. 2. As such, users can run the algorithms only until they have gathered sufficient information to move forward.

A. Value of Patterns and Dictionary

We have stated that the key to making both of our anytime algorithms practical is two-fold. First, we place entire patterns of nodes at each level of the search tree, instead of placing nodes one at a time. This alteration reduces the height of our search tree. Second, we consider only commonly occurring arrangements of such patterns, stored in a dictionary, which reduces the branching factor of the search tree.

In this section, we test the impact of these decisions for our architecture and benchmark suite. To do so, we consider three variations of our two anytime algorithms. The baseline algorithms are the ones described in this paper, which use patterns and a dictionary. The single algorithms place nodes one by one, rather than placing multinode configurations. However, we still use a dictionary for each of these node placements. Practically, this variation was implemented by constructing and using a dictionary that recognized only Pattern A in Fig. 10. The single, No Dictionary algorithms do not use patterns and do not use a dictionary. Instead, all node placements are considered. In our example, we consider all placements within distance 10, for a branching factor of 440. Clearly, this creates a much larger search tree than when a dictionary is used. However, most of the solutions are poor ones, and thus may rarely be explored by our algorithms.

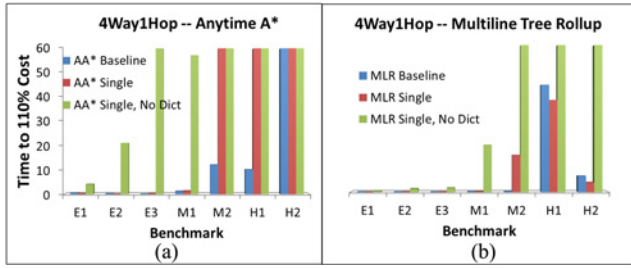


Fig. 13. Comparison of anytime algorithms with and without dictionaries and patterns in terms of time required before reaching a solution within 110% of the known optimal. Times of 60 min indicate that the algorithm did not reach a solution within 110% after 1 h of computation time.

Fig. 13 shows the results of this comparison for 4Way1Hop. This figure shows computation time required to reach a solution within 10% of the best known solution. Anytime A* results are shown in the left figure, and Anytime Multiline Tree Rollup results are shown on the right. We can see from these figures that removing the dictionary is devastating, with half of all cases returning no solution after one hour. Regarding the use of patterns, the single algorithm performs nearly as well as baseline in many cases, and slightly outperforms it in two instances, but in three instances it encounters difficulties in finding a near-optimal solution, requiring longer runtime or being unable to find a near-optimal solution after 1 h of computation time. We hypothesize that the difference between the two treatments (single and baseline) would be even greater if our dictionaries were improved to include a better selection of arrangements.

B. Need for Bootstrapping

There is a bootstrapping process for this algorithm, as a dictionary must be constructed. We give results for a dictionary that was created based on results of running simulated annealing on graphs different from the one we are testing. To test an alternative means of constructing dictionaries, we also ran our algorithms using dictionaries created from graphs that were produced by the players of the game UNTANGLED [14]. What we found was that these dictionaries contained much more variety in how configurations of multiple nodes were arranged. In particular, they contained higher proportions of sub-optimal arrangements that kept connected nodes close to one another, but would require the addition of passgates for routing. First, we thought that these dictionaries with greater variety might be beneficial for finding solutions for difficult architecture/benchmark combinations. However, we found in early testing that the algorithms using the player dictionaries behaved slightly worse than those using simulated annealing dictionaries and player dictionaries were not considered further. On further reflection, we believe that the player dictionaries were not sufficiently complete, reflecting as they did individual player biases. simulated annealing, in contrast, due to its nature of random exploration, tended to have a fairly even distribution of solutions over the space of possibilities. It is possible that combining the player and simulated annealing dictionaries would give good results, but we did not attempt this option.

The dictionaries generated from results of simulated annealing are not very surprising in retrospect, and it is possible that

TABLE III
COMPARISON OF ARCHITECTURES BASED ON EXHAUSTIVE
ENUMERATION OF MAPPINGS OF THE TEST GRAPH OF FIG. 4

	4Way2Hop	4Way1Hop	8Way	4Way
Branching Factor	184	48	96	88
# Leaf Nodes	1.1B	5.3M	85M	60M
# Feasible Solns	1.0M	1536	3840	2048
% Feasible Solns	0.091	0.029	0.005	0.003
Scaled % Feas Solns	26.6	8.5	1.3	1

they could be generated automatically from a cost function on edges (e.g., return all arrangements below a certain cost). This direction would eliminate the need for bootstrapping. On the other hand, we speculate that better performance could be obtained by fine tuning our dictionaries to focus search on a minimal collection of most useful arrangements. Optimizing the dictionary is an interesting future research area.

Our current dictionary has useful information that we do not employ in our search—the relative frequency of each arrangement was recorded when creating the dictionaries. The most frequently observed arrangements from one typical dictionary are shown in Fig. 10. We considered using this information to focus the search for good mappings, but decided against it. On the one hand, exploring most common arrangements first could provide results more quickly in the easy cases. On the other hand, DFGs that require less popular arrangements may suffer from this bias. Exploring how frequency information may best be used is another interesting topic of future research.

C. Mappability of Architecture Plus Dictionary

Our results tell us something about how easy it is to map our benchmarks onto each test architecture. Fig. 2, for example, clearly suggests that finding good solutions is easiest for 4Way2Hop, followed by 4Way1Hop, followed by 4Way and 8Way. This information can potentially be very useful. In fact, we may be able to determine the mappability of an architecture plus dictionary without running any of the benchmarks.

We explore this idea with the simple test graph in Fig. 4. The search tree for this DFG consists of four levels, as shown in the bottom of Fig. 4. We can exhaustively expand this search tree and determine which solutions are feasible. Results for all four architectures are shown in Table III. This table shows the branching factor of the search tree for each architecture, which is based on the dictionary for that architecture. It shows the number of leaf nodes of the resulting search tree, the number of solutions, and the percentage of feasible solutions, both raw percentage and normalized relative to the 4Way architecture.

If we look at the last two rows of Table III, we get a quantitative assessment of the difficulty of mapping to each of the four architectures. For example, we can see that the density of solutions for 4Way2Hop is 26.6 times greater than the density of solutions for 4Way. By the numbers, 4Way2Hop is by far the easiest architecture, 4Way1Hop is relatively easy, and 8Way and 4Way are the most difficult.

We further note for this example that 4Way2Hop has solutions having many different costs in a roughly Gaussian distribution, 8Way has solutions with a handful of different

costs, and 4Way1Hop and 4Way architectures have many feasible solutions with identical cost values. 4Way, for example, has 2048, or 2^{11} solutions with the same cost, which are likely variations of the same mapping exhibiting many different symmetries. If we could manage these symmetries effectively, we could potentially achieve orders of magnitude speedups in completely exploring the search space. We are very interested in exploring this opportunity.

D. Scalability

Finally, we consider the issue of scalability. The Anytime A* algorithm is well known to consume large amounts of memory for some types of search problems. We did not encounter memory limitations for the problem sizes considered here, in part because the Anytime version of the algorithm encourages more depth first exploration, which is much less memory intensive. Determining the maximum problem size that our algorithms can handle is a topic of future research. If memory does become an issue, we note Anytime graph search algorithms that are designed specifically to trade off computational time for reduced memory usage [60], [61].

Scalability to much larger DFG sizes may involve the use of approaches complementary to our own. One could consider developing dictionaries designed specifically for placing larger clusters. The concept presented here of placing multinode configurations using dictionaries of common arrangements could be useful whether those patterns were collections of LUTs, ALUs, custom instructions, or clusters of any of these.

X. CONCLUSION

In conclusion, we showed in this paper that the use of patterns and dictionaries brought algorithms such as A* that were traditionally considered to be slow and memory intensive into the range of practically useful algorithms for CGRA sized problems. We showed that in many cases, near optimal results can be obtained in seconds or minutes, giving a designer the ability to stop a simulation early to fine tune a good design or abandon a poor one. We also showed that early results provided a good indication of mappability of an architecture, which can be further explored through small test graphs. Finally, algorithms such as A* that provide lower bounds on solution cost can give a designer confidence that the solutions obtained were within a specified percent of optimal.

REFERENCES

- [1] M. Hamzeh, A. Shrivastava, and S. Vrudhula, "EPIMap: Using epimorphism to map applications on CGRAs," in *Proc. DAC*, 2012, pp. 1280–1287.
- [2] J. Cong, "Era of customization and implications to EDA," in *Proc. 48th Annu. DAC Keynote Speech*, 2011.
- [3] H. Park, K. Fan, S. A. Mahlke, T. Oh, H. Kim, and H.-S. Kim, "Edge-centric modulo scheduling for coarse-grained reconfigurable architectures," in *Proc. Parallel Arch. Compilation Tech.*, 2008, pp. 166–176.
- [4] J. W. Yoon, A. Shrivastava, S. Park, M. Ahn, and R. Jeyapaul, "SPKM: A novel graph drawing based algorithm for application mapping onto coarse-grained reconfigurable architectures," in *Proc. ASP-DAC*, Jan. 2008, pp. 776–782.
- [5] R. Hartenstein, T. Hoffmann, and U. Nageldinger, *Design-Space Exploration of Low Power Coarse Grained Reconfigurable Datapath Array Architectures* (Lecture Notes in Computer Science, vol. 1918). Berlin, Germany: Springer, 2000.
- [6] M. V. da Silva, R. Ferreira, A. Garcia, and J. M. P. Cardoso, "Mesh mapping exploration for coarse-grained reconfigurable array architectures," in *Proc. ReConFig*, Sep. 2006, pp. 1–10.
- [7] K. Karuri, A. Chattopadhyay, X. Chen, D. Kammler, L. Hao, R. Leupers, H. Meyr, and G. Ascheid, "A design flow for architecture exploration and implementation of partially reconfigurable processors," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 16, no. 10, pp. 1281–1294, Oct. 2008.
- [8] L. Bauer, M. Shafique, and J. Henkel, "Cross-architectural design space exploration tool for reconfigurable processors," in *Proc. DATE*, Apr. 2009, pp. 958–963.
- [9] Y. Kim, R. Mahapatra, and K. Choi, "Design space exploration for efficient resource utilization in coarse-grained reconfigurable architecture," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 18, no. 10, pp. 1471–1482, Oct. 2010.
- [10] T. Dean and M. Boddy, "An analysis of time-dependent planning," in *Proc. 7th Nat. Conf. AI*, 1988, pp. 49–54.
- [11] E. Hansen and R. Zhou, "Anytime heuristic search," *J. AI Res.*, vol. 28, no. 1, pp. 267–297, Mar. 2007.
- [12] M. Likhachev, D. Ferguson, G. Gordon, A. Stentz, and S. Thrun, "Anytime search in dynamic graphs," *Artif. Intell. J.*, vol. 172, no. 14, pp. 1613–1643, 2008.
- [13] *International Science and Engineering Visualization Challenge*. (Feb. 2013) [Online]. Available: <http://www.sciencemag.org/site/special/vis2012/>
- [14] G. Mehta. (2012). *Untangled* [Online]. Available: <https://untangled.unt.edu>
- [15] M. Garey and D. Johnson, "Crossing number is NP-complete," *SIAM J. Algebraic Discrete Methods*, vol. 4, no. 3, p. 312, 1983.
- [16] V. Tehre and R. Kshirsagar, "Survey on coarse grained reconfigurable architectures," *Int. J. Comput. Appl.*, vol. 48, no. 16, pp. 1–7, Jun. 2012.
- [17] K. Choi, "Coarse-grained reconfigurable array: Architecture and application mapping," *IPSJ Trans. Syst. LSI Design Method.*, vol. 4, no. 0, pp. 31–46, Feb. 2011.
- [18] G. Theodoridis, D. Soudris, and S. Vassiliadis, "A survey of coarse-grain reconfigurable architectures and CAD tools," in *Fine- and Coarse-Grain Reconfigurable Computing*, S. Vassiliadis and D. Soudris, Eds. Springer Netherlands, 2008, pp. 89–149.
- [19] R. Hartenstein, "A decade of reconfigurable computing: A visionary retrospective," in *Proc. DATE*, 2001, pp. 642–649.
- [20] S. W. Gehring and S. H.-M. Ludwig, "Fast integrated tools for circuit design with FPGAs," in *Proc. FPGA*, 1998, pp. 133–139.
- [21] R. Ferreira, A. Garcia, and T. Teixeira, "A polynomial placement algorithm for data driven coarse-grained reconfigurable architectures," in *Proc. ISVLSI*, 2007, pp. 61–66.
- [22] G. Dimitroulakis, S. Georgiopoulos, M. D. Galanis, and C. E. Goutis, "Resource aware mapping on coarse grained reconfigurable arrays," *Microprocessors Microsyst.*, vol. 33, no. 2, pp. 91–105, Mar. 2009.
- [23] V. Betz and J. Rose, "VPR: A new packing, placement and routing tool for FPGA research," in *Proc. Int. Conf. Field-Prog. Logic Appl.*, 1997, pp. 213–222.
- [24] S. Friedman, A. Carroll, B. V. Essen, B. Ylvisaker, C. Ebeling, and S. Hauck, "SPR: An architecture-adaptive CGRA mapping tool," in *Proc. FPGA*, 2009, pp. 191–200.
- [25] C. Ebeling, D. Cronquist, and P. Franklin, "RaPiD: Reconfigurable pipelined datapath," in *Proc. Field-Programmable Logic Smart Appl. New Paradigms Compilers*, 1996, pp. 126–135.
- [26] T. Taghavi, X. Yang, and B.-K. Choi, "Dragon2005: Large-scale mixed-size placement tool," in *Proc. ACM Int. Sym. Phys. Des.*, 2005, pp. 245–247.
- [27] C. Sechen and A. Sangiovanni-Vincentelli, "The TimberWolf placement and routing package," *IEEE J. Solid-State Circuits*, vol. 20, no. 2, pp. 510–522, Apr. 1985.
- [28] G. Lee, S. Lee, K. Choi, and N. Dutt, "Routing-aware application mapping considering Steiner points for coarse-grained reconfigurable architecture," in *Proc. Reconfigurable Comput. Architectures Tools Appl.*, 2010, pp. 231–243.
- [29] J. Yoon, A. Shrivastava, S. Park, M. Ahn, and Y. Paek, "A graph drawing based spatial mapping algorithm for coarse-grained reconfigurable architectures," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 17, no. 11, pp. 1565–1579, Nov. 2009.

[30] S. Adya, S. Chaturvedi, J. Roy, D. Papa, and I. Markov, "Unification of partitioning, placement and floorplanning," in *Proc. ICCAD*, Nov. 2004, pp. 550–557.

[31] A. R. Agnihotri, S. Ono, and P. H. Madden, "Recursive bisection placement: Feng shui 5.0 implementation details," in *Proc. ACM Int. Sym. Phys. Des.*, 2005, pp. 230–232.

[32] T. C. Chen, T. Chang Hsu, Z. Wei Jiang, and Y. Wen Chang, "NTUplace: A ratio partitioning based placement algorithm for large-scale mixed-size designs," in *Proc. ACM Int. Sym. Phys. Des.*, 2005, pp. 236–238.

[33] T.-C. Chen, Z.-W. Jiang, T.-C. Hsu, H.-C. Chen, and Y.-W. Chang, "NTUplace3: An analytical placer for large-scale mixed-size designs with preplaced blocks and density constraints," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 27, no. 7, pp. 1228–1240, Jul. 2008.

[34] Y. Sankar and J. Rose, "Trading quality for compile time: Ultra-fast placement for FPGAs," in *Proc. FPGA*, 1999, pp. 157–166.

[35] A. Agnihotri and P. Madden, "Fast analytic placement using minimum cost flow," in *Proc. ASP-DAC*, Jan. 2007, pp. 128–134.

[36] T. Luo and D. Z. Pan, "DPlace2.0: A stable and efficient analytical placement based on diffusion," in *Proc. ASP-DAC*, 2008, pp. 346–351.

[37] H. Bian, A. C. Ling, A. Choong, and J. Zhu, "Toward scalable placement for FPGAs," in *Proc. FPGA*, 2010, p. 147.

[38] N. Viswanathan and C.-N. Chu, "FastPlace: Efficient analytical placement using cell shifting, iterative local refinement, and a hybrid net model," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 24, no. 5, pp. 722–733, May 2005.

[39] J. Cong, H. Huang, and W. Jiang, "Pattern-mining for behavioral synthesis," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 30, no. 6, pp. 939–944, Dec. 2011.

[40] K. Martin, C. Wolinski, K. Kuchcinski, A. Floch, and F. Charot, "Constraint programming approach to reconfigurable processor extension generation and application compilation," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 5, no. 2, pp. 1–38, Jun. 2012.

[41] O. Bringmann and W. Rosenstiel, "Resource sharing in hierarchical synthesis," in *Proc. ICCAD*, 1997, pp. 318–325.

[42] J. Cong, Y. Fan, G. Han, and Z. Zhang, "Application-specific instruction generation for configurable processor architectures," in *Proc. ISFPGA*, 2004, pp. 183–189.

[43] C. Wolinski, K. Kuchcinski, and E. Raffin, "Automatic design of application-specific reconfigurable processor extensions with UPaK synthesis kernel," *ACM ToDAES*, vol. 15, no. 1, pp. 1–36, Dec. 2009.

[44] Z. Zou, J. Li, H. Gao, and S. Zhang, "Mining frequent subgraph patterns from uncertain graph data," *IEEE Trans. Knowledge Data Eng.*, vol. 22, no. 9, pp. 1203–1218, Sep. 2010.

[45] V. Krishna, N. Suri, and G. Athithan, "A comparative survey of algorithms for frequent subgraph discovery," *Current Sci.*, vol. 100, no. 2, pp. 190–198, 2011.

[46] R. E. Korf, "Finding optimal solutions to Rubik's cube using pattern databases," in *Proc. Natl. Conf. Artif. Intell.*, 1997, pp. 700–705.

[47] R. E. Korf, "Research challenges in combinatorial search," in *Proc. 26th AAAI Conf. Artif. Intell.*, 2012, pp. 2129–2133.

[48] N. R. Sturtevant, A. Felner, M. Likhachev, and W. Ruml, "Heuristic Search Comes of Age," in *Proc. Natl. Conf. Artificial Intell. (AAAI)*, 2012.

[49] S. Edelkamp and S. Schroedl, *Heuristic Search: Theory and Applications*. San Francisco, CA, USA: Morgan Kaufmann, 2011.

[50] R. E. Korf and A. Felner, "Recent progress in heuristic search: A case study of the four-peg towers of Hanoi problem," in *Proc. 20th IJCAI*, 2007, pp. 2324–2329.

[51] S. Edelkamp, "Planning with pattern databases," in *Proc. 6th Eur. Conf. Planning*, 2001, pp. 13–34.

[52] S. Kupferschmid and M. Wehrle, "Abstractions and pattern databases: The quest for succinctness and accuracy," in *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Germany: Springer, 2011, pp. 276–290.

[53] G. Mehta, C. Crawford, X. Luo, N. Parde, K. Patel, B. Rodgers, A. K. Sistla, and A. Yadav. (2013). "Untangled: A game environment for discovery of creative mapping strategies." Dept. Electr. Eng., Univ. North Texas, Denton, TX, USA, Tech. Rep. UNT-EE-TR-02 [Online]. Available: <https://engineering.unt.edu/electrical/reconfigurable-computing-lab/publications-rcl>

[54] R. Dechter and J. Pearl, "Generalized best-first search strategies and the optimality of A*," *J. ACM*, vol. 32, no. 3, pp. 505–536, Jul. 1985.

[55] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Englewood Cliffs, NJ, USA: Prentice-Hall, 2009.

[56] Y. Ye and C. K. Liu, "Synthesis of detailed hand manipulations using contact sampling," *ACM Trans. Graph.*, vol. 31, no. 4, pp. 1–10, 2012.

[57] N. Bansal, S. Gupta, N. Dutt, A. Nicolau, and R. Gupta, "Network topology exploration of mesh-based coarse-grain reconfigurable architectures," in *Proc. DATE*, 2004, pp. 474–479.

[58] B. Mei, A. Lambrechts, J.-Y. Mignolet, D. Verkest, and R. Lauwereins, "Architecture exploration for a reconfigurable architecture template," *IEEE Design Test Comput.*, vol. 22, no. 2, pp. 90–101, Mar.–Apr. 2005.

[59] F. Bouwens, M. Berekovic, B. De Sutter, and G. Gaydadjiev, "Architecture enhancements for the adres coarse-grained reconfigurable array," in *Proc. HiPEAC*, 2008, pp. 66–81.

[60] W. Zhang, "Complete anytime beam search," in *Proc. Natl. Conf. Artif. Intell.*, 1998, pp. 425–430.

[61] R. Zhou and E. Hansen, "Beam-stack search: Integrating backtracking with beam search," in *Proc. Int. Conf. Automated Planning Scheduling*, 2005, pp. 90–98.



Gayatri Mehta (M'05) received the Ph.D. degree in electrical and computer engineering from the University of Pittsburgh, Pittsburgh, PA, USA, in 2009.

She is currently an Assistant Professor with the Department of Electrical Engineering, University of North Texas, Denton, TX, USA. Her current research interests include the areas of reconfigurable computing, low-power very large scale integration design, systems-on-a chip design, electronic design automation, embedded systems, portable/wearable computing, and energy harvesting.



Krunal Kumar Patel is currently pursuing the Master's degree in computer science and engineering at the University of North Texas (UNT), Denton, TX, USA.

He is at the Reconfigurable Computing Laboratory, UNT, under the mentorship of Dr. G. Mehta.



Natalie Parde is currently pursuing the bachelor's degree in computer science and engineering at the University of North Texas (UNT), Denton, TX, USA.

She is with the Reconfigurable Computing Laboratory, UNT, under the mentorship of Dr. G. Mehta. She has participated in the university's summer undergraduate program in engineering research. She is a member of UNT's Honors College and currently serves as an Engineering Ambassador for the College of Engineering.



Nancy S. Pollard (M'93) received the Ph.D. degree in electrical engineering and computer science from the Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, USA, in 1994.

She is currently an Associate Professor with the School of Computer Science, Carnegie Mellon University (CMU), Pittsburgh, PA, USA. Before joining CMU, she was an Assistant Professor and part of the Computer Graphics Group, Brown University, Providence, RI, USA.

Dr. Pollard received the NSF CAREER Award in 2001 and the Okawa Research Award in 2006.