Physical Resource Binding for a Coarse-Grain Reconfigurable Array Using Evolutionary Algorithms

Fred Ma, John P. Knight, and Calvin Plett

***Abstract*—One of the challenges of designing for coarse-grain reconfigurable arrays is the need for mature tools. This is especially important because of the heterogeneity of the larger, more predefined (and hence more specialized) array elements. This work describes the use of a genetic algorithm (GA) to automate the physical binding phase of kernel design. We identify the generalizable features of an example platform and discuss suitable ways to harness the binding problem to a GA search engine.**

***Index Terms*—Coarse grain reconfigurable array, evolutionary algorithm, genetic algorithm (GA), physical binding, placement.**

I. Introduction

COARSE-GRAIN reconfigurable arrays are field-programmable arrays in which many of the gates have been preformed into word-oriented data-path units (DPUs). Course-grained arrays are less flexible than field programmable gate arrays (FPGAs), but can yield higher speeds and densities in applications which are aligned with the architecture. They have been receiving widespread attention for DSP on streaming data [1]–[7].

In all reconfigurable arrays, the desired circuit function is first mapped to available array elements (DPUs, memory, control, etc.) as a netlist. These elements are then *bound to* location-specific array instances. These two steps are well defined for fine-grained FPGAs but under development for coarse-grained arrays. This paper focuses on the second problem: the physical binding of a design to array instances. We demonstrate how architectural constraints can be handled using a genetic algorithm (GA). Our methodology currently targets architectures like the Chameleon platform [3] (Fig. 1), but we identify many features that can be generalized to other architectures.

The problem of physically organizing kernel netlists onto a coarse-grain reconfigurable array has been addressed before. An early place-and route-algorithm for coarse-grain arrays used simulated annealing (SA), and is part of an integrated CAD system for the rDPA architecture [1], [8]. Much of the research in this area has been in placement for fine-grain FPGAs. Here, routing typically follows placement, and an empirical metric is used during placement to estimate routability. For the fine-grain FPGA Garp, [9] mixes relative placement with synthesis to exploit the regularity of bit slices in mapping datapaths to fine-grain arrays. Like regular fine-grain FPGAs, the coarse-grain linear array RaPiD [5] has a separate placement done by SA [10], followed by routing done with Pathfinder [11] or a pipelining-aware router [12]. For the two-dimensional (2-D) coarse-grain array Morphosys [2], [13] mixes relative placement with transformations of computational expression trees to maximize throughput in the face of limited memory bandwidth (memory is external to the array). The work in [14] compiles from a high-level language SA-C [15] to fully scheduled instructions on the Morphosys platform. Both Morphosys and SA-C are geared toward vectorizable 2-D image processing. [Morphosys operates in a single instruction multiple data (SIMD) manner]. At a higher abstraction level is Raw's 2-D processor array [7] for which a SystemC-like language, StreamIt [16], was developed. Stream processing is represented as a collection of filter-based processes connected by channels. Processes are bound to processors via simulated annealing, and the resulting latency affects the scheduling.

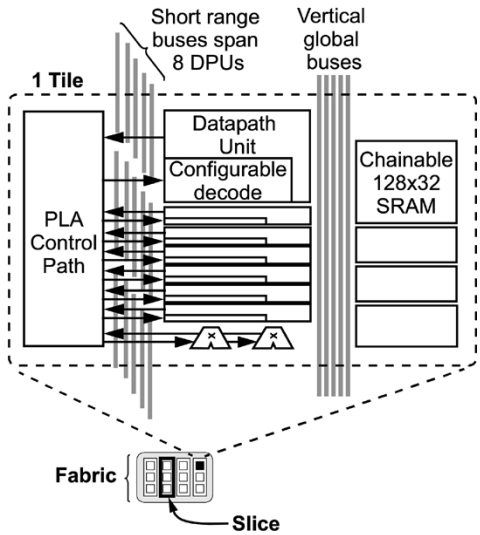


Fig. 1. Architecture of Chameleon array.

A. Chameleon Architecture

Chameleon's architecture [3], [17] is meant for stream data, but is not restricted to 2-D image processing. It is a 2-D array of DPUs, partitioned first into vertical slices, then into tiles (Fig. 1). Each DPU contains an ALU, a shifter, logic masking in both of its two operand inputs, and pipeline registers. The DPUs' behavior is controlled by finite state machines (FSMs) that are programmed into the PLA. In each cycle, control lines from the PLA select from eight "personalities" loaded into each DPU,

Manuscript received November 28, 2003; revised June 17, 2004 and November 11, 2004. This work was supported in part by Nortel Networks and Micronet, and presented in part at Engineering of Reconfigurable Systems and Algorithms (ERSA) 2004, Las Vegas, NV.

The authors are with the Department of Electronics, Carleton University, Ottawa, ON K1S 5B6, Canada (e-mail: fma@doe.carleton.ca; jknight@doe.carleton.ca; cp@doe.carleton.ca).

Digital Object Identifier 10.1109/TVLSI.2005.844286

not necessarily in SIMD fashion. Each personality consists of a set of configuration bits that determine how all the different components in the DPU operate. The local memories and PLAs allow kernels to be designed with some degree of distributed, locally controlled operation.

Chameleon's interconnect is primarily vertical, though horizontal buses span slices. Hence, slices form a natural partitioning point for large kernels. Within each slice, connections are mostly made by local wires; they can connect DPUs within eight positions of each other, even across tiles. In addition, a set of global buses span the whole slice. Their use requires registered destination ports to meet timing. Two DPUs at the bottom of each tile do multiplication, and DPUs in even rows can read memories while DPUs in odd rows can write to them.

The following features of the platform are general principles that can be applied to other platforms: 1) partitioning of DPUs (into slices, in this case) to limit complexity of the interconnect network; 2) lots of interconnect along one axis (vertical in this case), with limited global interconnect along the other axis; 3) partitioning of DPUs into tiles to limit complexity of the control PLA; 4) division of interconnect into local and global wires to limit complexity; 5) latency incurred with global wires; 6) gross location-specific resources that constrain the placement of DPUs to which they connect, e.g., multipliers and memories; and 7) interleaving DPUs with complementary capabilities to reduce complexity, e.g., even (odd) memory read (write).

B. Organization of the Rest of the Paper

This paper is organized as follows. In Section II, we describe the place-and-route problem in terms of the Chameleon architecture, approaches to its solution, and reasons for choosing GAs. In Section III, fundamental choices in the GA implementation are made based on a simplified placement problem. In Section IV, heterogeneous constraints are incorporated using penalization and repair of violations. Section V summarizes the schedule of steps in our GA. Section VI discusses results from several kernels, as well as variations of the algorithm parameters and iteration scheme.

II. Problem Details and Solution Approaches

The resource binding problem is simplified by a key feature of the Chameleon architecture: The output of each DPU is connected to the inputs muxes of all DPUs within eight rows of it. Unlike FGPAs, this local wiring is not shared. Therefore, a net's routability by local wires is automatically determined by the placement of source and destination DPUs. There is no "estimation" of routability during placement, and no separate step after placement to find routes between source and destination. If a DPU arrangement contains a net over eight rows from source to destination, it is simply and immediately found to be infeasible; there is no varying degree of acceptable delay.

Fig. 2 illustrates this reduced, simultaneous "place-and-route" for a hypothetical architecture with only 3 DPUs/tile, and a local wire reach of three rows up and two rows down. The placement problem at this point is: find an arrangement of the DPUs in the net list such that no nets exceed the local wire reach. This simplified placement problem is a starting point for the full placement algorithm. A realistic architecture will have a multitude of additional conditions, or *constraints*, that must be satisfied for a solution to be "feasible."

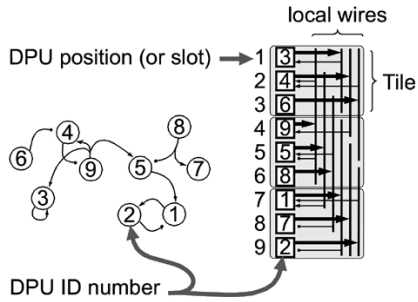


Fig. 2. DPUs in a netlist are arranged onto a slice. The required connections should be mostly realizable with the limited reach of local wires. Example architecture has 3 DPUs/tile and 3 tiles/slice. Local wires extend up three rows and down two rows, except near the top and bottom.

The method used to find a solution should be able to accommodate highly varied constraints. In Chameleon, for example, DPUs that access the local memories must be adjacent to those memories, and should occupy sites that have the correct memory read[write] capability. Other special conditions are fixed multiplier placement, and the availability (with possible retiming) of a few long lines for overlength nets.

Mathematically, the problem is as follows. A DPU netlist is a directed hypergraph (V, E), where each net $e_i \in E$ consists of source DPU $v_i^S \in V$ and one or more destinations $\{v_i^{D1}, v_i^{D2} \ldots\} \subseteq V$. The coarse-grain array is a vector of DPU positions $(p_1, p_2, \ldots, p_{N_{\mathrm{DPUs/slice}}})$. To allow gross position constraints, contiguous p_m are further grouped into abutted tiles $T = \{t_1, t_2, \ldots\}$ such that $t_k = \{p_{l+1}, p_{l+2}, \ldots, p_{kN_{\mathrm{DPUs/tile}}}\}$, where $l = (k-1)N_{\mathrm{DPUs/tile}}$. Each v_i may be required to reside within a subset of adjacent tiles (1-tile subsets in this case). We assume that the array interleaves N_{Classes} classes of DPUs with varying capabilities ($N_{\mathrm{Classes}} = 2$ for Chameleon). A v_i may also be required to reside in certain classes; hence, p_m are also grouped into classes $C = \{c_1, c_2, \ldots, c_{N_{\mathrm{Classes}}}\}$, where $c_k = \{p_{l \cdot N_{\mathrm{Classes}}+k} \,|\, 0 \le l < N_{\mathrm{DPUs/slice}}/N_{\mathrm{Classes}}\}$. Under this notation, the problem is then to arrange the v_i's onto the p_m's such that: 1) there are no more than $N_{\mathrm{GlobalWires/tile}}$ nets being sourced from each tile to destinations more than $l_{\mathrm{LocalWire}}$ away and 2) any memberships of each v_i in T and C are met.

The problem is currently formulated for a single-slice kernel. It does not consider the retiming that may be necessary from using global interconnect. At this time, we expect the designer to perform the global partitioning of the kernel into slices, and any retiming that may be needed from the use of global interconnect by the placement algorithm.

The manually placed kernel in [17] took days. As coarse-grain reconfigurable arrays become larger, this will become more demanding. In automating this, our goal is to provide quick enough feedback on design feasibility to maintain continuity of the designer's work.

The following problem features motivate our approach for the methodology. Because of the *heterogeneous constraints*, this ordering problem differs from the classical *travelling salesman problem* (TSP), in which a tour of cities must be organized to minimize travel costs [18], [19]. Since the problem is primarily

driven by constraint satisfaction, we are not seeking "the optimum" solution; any solution is good so long as it complies with the architecturally imposed constraints. Additionally, we seek a method that can *easily adapt* to changing constraints as a platform's architecture evolves. Because of these two considerations, we chose a heuristic search through the solution space rather than laboriously trying to recast the constraints for an optimum-finding method. Fitness-based searches in particular ease the problem encoding for a multitude of variations on a platform's architecture.

This effort focuses on a working solution; we do not address theoretical complexity here, though empirical complexity is explored in Section VI-B. Serious consideration (including hand routing attempts) indicate that if there is a polynomial time algorithm, it would be very hard to find. For such potentially NP-hard problems, heuristic approaches are often used. Just as important is the fact a problem's difficulty can change significantly with minor changes in constraints. A general methodology should have performance that is robust in the face of variations in the exact feature set, which again suggests a heuristic approach.

Since candidate solutions are permutations of the DPU indices, the search space method must be suitable for *combinatorial* problems, e.g., simulated annealing, tabu search, and GAs. One of the motivating reasons for choosing GAs was its amenability to parallel processing. Other iteration schemes are may also perform well, as discussed in Section VI-B.

III. Baseline GA Based on Interconnect

The conceptual foundations of GAs may be found in [18] and [20], while the myriad of issues regarding its application can be found in [21] and [22]. We freely borrow from, and modify, the more liberal population replacement strategies of *evolutionary strategies* (ES) [23]. GAs and ESs are collectively known as *evolutionary algorithms*; we will use the term GAs to emphasize the discrete nature of the problem parameters.

The basic operation of the GA is to maintain a population of (initially random) solution points, or *chromosomes*, or *genomes*. In each generation, better *offspring* solutions are formed by combining parameters, or *genes*, from existing *parent* chromosomes in a process known as *mating*. All new solutions are evaluated and assigned a *fitness*. The population improves because fitter chromosomes are more likely to be *selected* as parents. Their offspring conditionally displace population members through various schemes.

A chromosome is not exactly the same as a solution. A *phenotype* is a set of problem parameters that comprise a candidate solution, e.g., a list of which DPUs go into which position (Fig. 2). Its encoding as a chromosome is called a *genotype*. The two can be very different, depending on the mapping from chromosome to solution [for example, Fig. 3(b), next section]. The population can be viewed as a set of sample points in the genotype space. Through the generations, sample points tend to get denser in regions of good fitness, since the responsible gene values proliferate through the population.

For the DPU binding of Fig. 2, the DPU ordering naturally forms a simple chromosome consisting of an integer vector. The vector element is the gene, the element position corresponds to the DPU position, and the gene value is the index of the DPU to be located there. We will call this *vanilla* encoding.

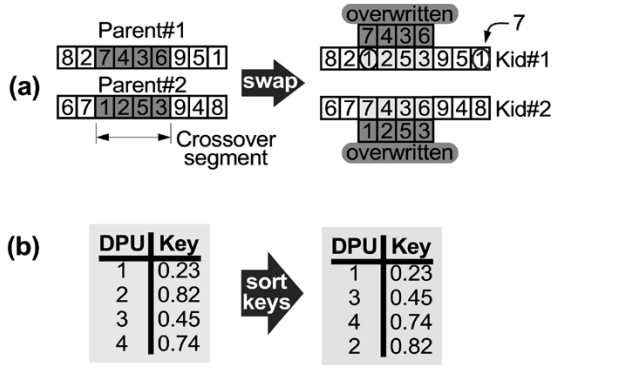


Fig. 3. (a) PMX: 2-point crossover that results in collisions, e.g., gene value 1 in the first offspring. PMX replaces the outside 1 with 7 (overwritten by 1 inside). If the new 7 outside had conflicted with a 7 inside, the process is repeated. (b) RKR: Obtaining a 4-slot ordering from a chromosome.

A. Crossover

In classical GAs, genes from two parents are often combined by *2-point crossover*: Two randomly determined *crossover points* in the chromosome delineate a segment of contiguous genes to be swapped between parents. Whether or not a favorable attribute gets propagated to the offspring depends on whether the associated genes, referred to as a *building block*, are kept together in the crossover. If these genes are far apart, a building block is less likely to fall entirely inside or outside of the crossover segment and remain intact; the building block is then *disrupted*.

Disruption is reduced by ordering the genes so that key building blocks are compact; however, it is difficult to recognize all ways that genes can combine to form building blocks, or how influential are various building blocks in the search. One way to address this is to explore different gene orderings by *inverting* the order of a random gene segment in a chromosome. This requires 1) that the genes be tagged to disassociate their meaning from their position and 2) a means to have a common order between mating parents. In this work, an alternate approach, *uniform crossover* [21], [22], is discussed below.

Disruption is further reduced by using an integer gene for a DPU index rather than several binary genes. This prevents the crossover points from falling between the bits of the DPU index, thus disrupting that information. However, it also suppresses a beneficial role, which is to create new DPU index values. This role is essential because gene values get less diverse with evolution. If this happens too fast, the population may *prematurely converge* on a suboptimal region of the search space (local minimum). To maintain diversity, regular *mutation* of a few chromosomes is needed to compensate for the lack of gene disruption. In mutation, one gene is randomly changed.

Ordering problems form a special subset of combinatorial problems because the only valid solutions are permutations of a set of indices. Swapping a random set of genes between valid solutions often yields invalid solutions, since a gene value might show up both inside and outside crossover segment [Fig. 3(a)].

We call this a *collision*. Avoiding this requires specialized crossovers that create offspring which are also permutations [18], [21], [22]. Different permutation-preserving crossovers propagate different kinds of order information, e.g., absolute position, adjacency, and/or relative ordering of gene values. Many schemes target TSP-like problems, for which solutions are rotationally invariant, i.e., $\{2,1,3\}$ is the same as $\{1,3,2\}$. These schemes are not suitable for DPU binding, since there are no local wires wrapping around from the top of the slice to the bottom.

We initially tried *partially matched crossover* (PMX), because it propagates both order and position information; a DPUs absolute position is important when it connects to a site-specific resource, while interconnecting wire length depends on both relative order and absolute position. In PMX, a 2-point crossover is followed by a *repair phase* that resolves collisions [Fig. 3(a)]. The conflicting value *outside* the crossover segment is changed to a value that is unlikely to cause a collision. Thus, building blocks inside the crossover segment are preserved, but the probability of preserving building blocks outside are worse than for a normal 2-point crossover. An apparent bias toward preservation of building blocks near the center of the chromosome is avoided by treating the chromosome as circular. This allows the crossover segment to straddle the end points.

To avoid the disruption in PMX, we explored the use of *random key representation* (RKR) [Fig. 3(b)], [21], [22], which encodes an ordering problem in a way that avpids collisions. Each gene in the chromosome corresponds to a DPU—or equivalently, its index—rather than a DPU position. Each gene value is a random *real* number in the range [0.0, 1.0], which is used as a sort key. The DPU ordering is obtained by sorting the DPUs according to the sort key. The higher the sort key value for a DPU, the more likely it will end up toward the bottom of the slice. The sort key acts as a sloppy indicator of DPU position; the final position of any one DPU depends on the sort keys for all the DPUs. Intuitively, each collision-free point in the vanilla genotype space maps to a region of RKR space.

Under RKR, we tried *uniform crossover* because our problem has different building blocks from the TSP. In the TSP, the adjacency of cities to be visited is important, since the cost is determined by the edges between consecutive cities. Since 2-point crossover and PMX preserve information represented by close-together genes, adjacency of cities is propagated. In the DPU binding problem, however, interconnections must not exceed the length of local wires. Since these nets do not necessarily connect adjacent DPUs, we dispensed with crossovers that favor compact building blocks. In uniform crossover, genes to be swapped are randomly determined.

B. Cost and Fitness Scaling

The GA search is driven by the fitnesses of the solutions, which must reflect how compliant they are to the various constraints in Section I. Constraint driven searches differ from optimization searches in that constraint violations are first combined into a cost, and then converted to fitness. If fitness is calculated as 1/cost, the fitness explodes toward infinity as the violations approach zero. This means a few better-than-average solutions will be selected for mating most of the time, crowding out diversity and prematurely converging. Zero cost can be avoided by adding a constant "baseline" cost to the overall cost. The selection of this offset cost, however, is an open question, and its effect on selective pressure depends on the distribution of costs.

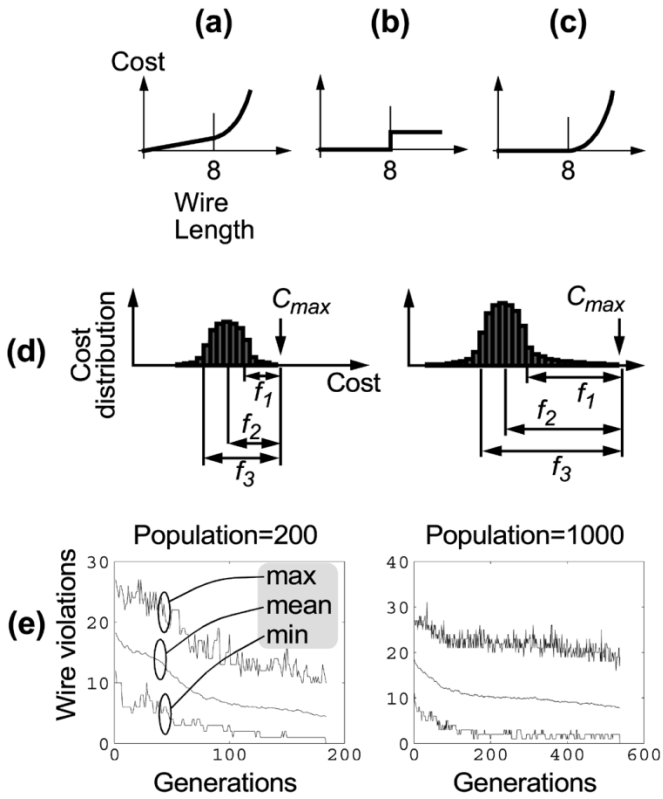


Fig. 4. (a)–(c) Wire length cost schemes for local wires that span eight DPUs. (d) Fitness as the cost difference from maximum cost. Small $C_{\max}$ (left) accentuates *relative* differences between fitnesses, while large $C_{\max}$ (right; e.g., from more extreme outliers in larger populations) yields relatively similar fitnesses. (e) Delayed convergence from reduced selective pressure. Since the entire population is replaced in each generation, the larger population took almost 600×10^3 matings; the smaller population took less than 40×10^3 matings.

We first establish a cost structure based on the wire lengths needed to connect a DPU arrangement for a chromosome [Fig. 4(a)–(c)]. Costs for violating any location constraints are then incorporated. Finally, individual costs are summed into overall chromosome cost, which is converted to fitness.

In wire cost scheme Fig. 4(a), the quadratic cost for overlength wires is typical of schemes that penalize a solution based on how much it encroaches into an infeasible region in the search space. However, we also impose a shallow linear ramped cost for wires of *legal* length to encourage connected DPUs to be closer together. Hence, it is possible for a feasible solution to have nonzero cost in this scheme.

The premise for Fig. 4(b) was that all legal wire lengths were equally good and all overlength wires were equally bad. This scheme often converged prematurely.

The best performing cost scheme was Fig. 4(c), a combination of (a) and (b). Compared to (a), its search time was more consistent. This was attributed to greater diversity, due to absence of selective pressure from the linear cost ramp.

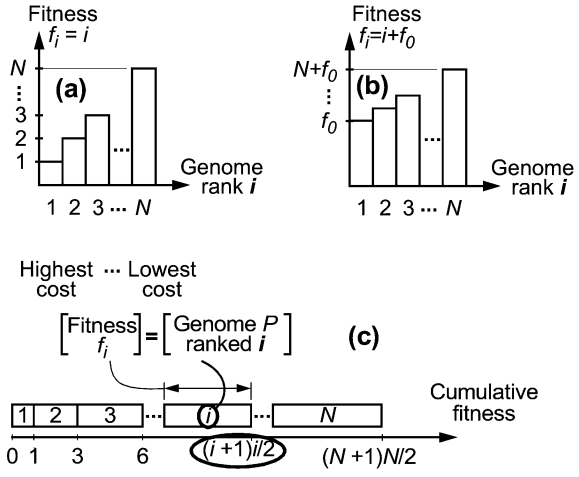


Fig. 5. (a) Ranked fitness. (b) Adding offset to rank to reduce selective pressure. (c) Unrolled roulette wheel of fitnesses.

Global vertical wires in the architecture are easily accommodated by assigning them to the worst violators of local wire length, thus ensuring the most value for a scarce resource. Those penalties are then "pardoned" from the overall cost. In the Chameleon architecture, each tile can drive three global 32-bit buses, totalling nine buses for the slice. Hence, the wiring cost assessment will pardon the three longest overlength wires driven from each tile. To minimize retiming problems, nets that drive registered inputs can be favored in allocating these global wires.

We initially obtained fitness from cost by subtracting a chromosome's cost from the greatest cost in the population, $C_{\max}$ [Fig. 4(d)]. This guarantees that all chromosomes had nonzero fitness (except the worst one) and had some chance of mating, thus ensuring diversity. However, selective pressure is reduced for a large population size N_{Pop}. To see how, consider a probability distribution of chromosomes over cost. Consider any (large) cost threshold $C_0 < C_{\max}$ for which there is a 90% chance of a chromosome having lesser cost. For a population of N_{Pop}, the chances of having a cost greater than C_0 is, $(1 - 0.9)^{N_{\text{Pop}}}$ which increases with N_{Pop}. Therefore, for the same cost distribution and any given large threshold C_0, a larger population has a greater likelihood of containing an outlying genome with a greater cost. Hence, large a N_{Pop} typically yields large $C_{\max}$, which leads to fitnesses that relatively similar [Fig. 4(d)]. This reduces selective pressure and leads to long search times [Fig. 4(e)].

Sigma truncation avoids the modulation of selective pressure by population size. The trade-off is that it does not ensure the inclusion of all chromosomes for selection. Instead, $C_{\max}$ is taken to be some number of standard deviations above the mean cost; fitness is taken to be zero for any chromosomes with higher cost. This effectively reduces the population size, and diversity.

To avoid both modulation of selective pressure and exclusion of chromosomes from selection, *ranked fitness* was used. The chromosomes are ranked in order of decreasing cost [Fig. 5(a)]. The rank is then linearly mapped to fitness. Diversity is enhanced because even the poorest chromosomes have a probability of selection which is not almost zero. Furthermore, relative fitnesses and selective pressure are immune to $C_{\max}$. As well, isolated star performers remain close to the runners up and are prevented from crowding out diversity. Finally, a controlled degree of differentiation is maintained between all chromosomes even if some absolute costs are bunched together in the cost distribution. Effectively, ranked fitness provides locally adapting nonlinearity to maintain steady selective pressure, both in time, as the generations elapse, as well as across the population in any generation. The cost-fitness conversion has large differentiation in cost regions where there are many chromosomes, with costs bunched together; where costs are far apart, the differentiation is reduced.

Selective pressure can be controlled through the mapping between rank and fitness. For example, consider a population of $N_{\text{Pop}} = 200$ chromosomes. Suppose one wants to combat premature convergence. Selective pressure can be decreased by adding an offset f_0 to all the fitnesses [Fig. 5(b)], thus compressing relative fitness. Suppose that one wants a level of diversity such that the ratio of ***p***robabilities of selecting the ***b***est and ***w***orst chromosomes is $P_{b2w} = 10$. By adding f_0 to the tallest and shortest bars (and all other bars) in Fig. 5(a), we calculate $P_{b2w} = (f_0 + N)/(f_0 + 1)$, from which $f_0 = (N - P_{b2w})/(P_{b2w} - 1)$. For $P_{b2w} = 10$, $f_0 = 190/9 = 21$.

C. Selection of Parents

The *roulette wheel* method places all genomes on a pie chart, with the slice width proportional to fitness. A dial is spun to select a parent (or parents). Implementation is done by mapping the rim of the pie chart to the real number range $[0.0, \Sigma_j f_j]$ [Fig. 5(c)]. For ranked fitness, a parent is selected by generating a random number $x\varepsilon[0.0, \Sigma_j f_j]$; the selected parent P is the genome whose segment encompasses x. This conversion of x to P may be done by table lookup (LUT) via binary search. (For a multiarm roulette wheel, linearly scanning the LUT may be more efficient). As can be seen from Fig. 5(c), this table is trivial for ranked fitness, since the segment delineation points in $[0.0, \Sigma_j f_j]$ are the cumulative sum of linear unit-ramped fitnesses. Hence, P (or rather, its rank i) is related to its position in $[0.0, \Sigma_j f_j]$ by an arithmetic series. P can thus be algebraically determined from x.

For the case where f_0 is added to all the ranks, the arithmetic series of Fig. 5(c) is modified slightly. It is straightforward to show that x maps to P's rank i according to

$$i = \left\lceil \frac{\sqrt{8x + (2f_0 + 1)^2} - 1 - 2f_0}{2} \right\rceil .$$

For Fig. 5(a) and (c), $f_0 = 0$.

D. Population Replacement, Mutation, and Diversity

We define *kids* as any newcomers to the population, *offspring* as kids from mating, and *mutants* as kids from mutation. Our initial attempts used *generational* evolution. For a population of N_{Pop}, this scheme selects N_{Pop} pairs of parents at a time, with possible repetition, to generate N_{Pop} offspring; the entire population is replaced at once. Unfortunately, this kind of search threw out even the best chromosomes with the old generation. We next tried *steady-state* replacement, in which one mating was performed per "generation." Any offspring and mutants are

incorporated into the population before selecting parents for the next generation.

Initially, mutation was performed only on offspring, e.g., a probability of mutation $P_{\text{mut}} = 50\%$ generates an average of one mutant from two offspring, yielding three newcomers or *kids*. This method gave frequent premature convergence where the cost converged to a single unchanging value throughout most of the population. We simultaneously measured diversity using several metrics and found it decreased rapidly. To combat this more chromosomes from the population were mutated, e.g., for a population of N chromosomes, $P_{\text{mut}} = 50\%$ now generates $N/2$ mutants, selected with uniform probability, without repetition. Including offspring, this yields $N/2 + 2$ kids per generation. Because of the dramatic improvement that this gave, we explored a range of values for the fraction of the population to replace each generation, as well as the ratio of mutants to offspring. These ratios seemed more important for harder problems. 50% and 40%, respectively, worked well as discussed in Section VI.

The initial population replacement policy was to merge the kids with the population of N, sort all chromosomes by cost, and keep the best N. There is no guarantee of how many kids are kept. Another experiment, which improved diversity, was to unconditionally inject the kids into the population, displacing the worst members. Diversity was further enhanced by removing duplicates among all kids, as well as duplicates between kids and the population. Note this is much more meaningful with integer genes of vanilla encoding, since RKR's real valued genes can be very similar but still technically nonidentical.

IV. Heterogeneous Constraints

A. Tile Bound DPUs

The placement of a DPU may be constrained to a tile for a number of reasons. In [17], for example, the memories of a tile are chained into a contiguous memory, and the DPU must read from or write to the memory. Another reason may be that certain DPUs share the same PLA (Fig. 1), e.g., because of a common FSM for control, or FSMs with a lot of mutual dependence. In general, a placement algorithm that accommodates tile bound DPUs allows the array architecture to have DPUs with different capabilities in different tiles. We refer to the binding tile as the *home tile*.

Initially, we penalized solutions with tile binding violations; however this rendered much of the search space infeasible. Furthermore, since RKRs keys are a sloppy specification of position, this would discourage tile-bound DPUs from going near the tile boundary, even if the final solution requires this. In [17], for example, DPUs bound to the top/bottom tiles "strained" to be close to the center tile, since central positions were favorable for interconnection. A generalization of this is that valid solutions are likely to lie near infeasible regions of the search space for those problems that push resource limits. To avoid repulsion of tile bound DPUs from the tile boundary, we modified the genotype–phenotype decoding to allow a gentler penalization. For illustration, we assume a reduced architecture with $N_{\text{DPUs/tile}} = 4$ DPUs/tile and $N_{\text{tiles/slice}} = 3$ tiles/slice.

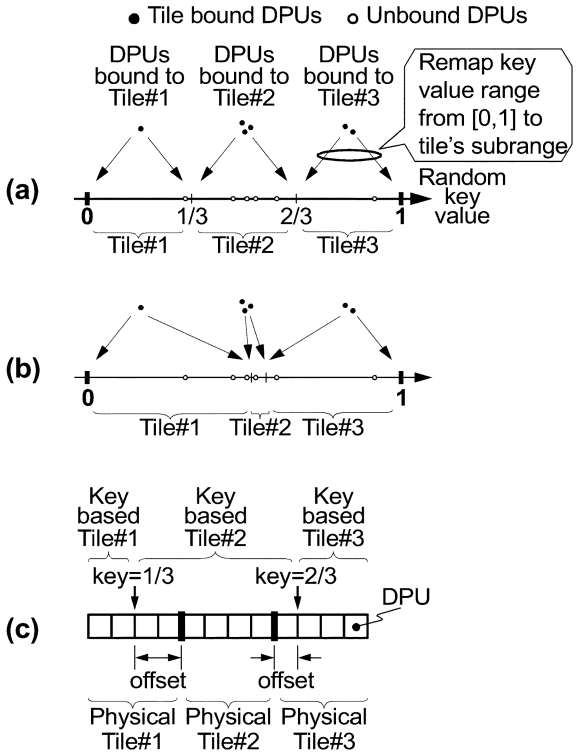


Fig. 6. Modifying the chromosome decoding to support tile binding for a 4 DPU/tile architecture.

Fig. 6(a) shows a starting point for linearly remapping the key values of tile bound DPUs to the subrange for their home tiles. A high key value then causes the DPU to be positioned toward the tail end of the *home* tile t_{home}. We need a way to define the home tile's subrange. To simply assume equal subranges $[(t_{\text{home}} - 1)/N_{\text{DPUs/tile}}, t_{\text{home}}/N_{\text{DPUs/tile}}]$ would cause some tiles to have more than $N_{\text{DPUs/tile}}$ DPUs, as in Fig. 6(a). The tile boundaries' key values must be defined in a way that ensures $N_{\text{DPUs/tile}}$ DPUs in each tile, assuming $N_{\text{DPUs/tile}} N_{\text{tiles/slice}}$ DPUs overall [Fig. 6(b)]. If tile t has $N^{(t)}_{\text{bound}}$ bound DPUs, it should have $N_{\text{DPUs/tile}} - N^{(t)}_{\text{bound}}$ unbound DPUs; thus, the unbound DPUs that straddle each boundary are easily determined by counting out the sorted unbound DPUs along the key number line. It remains to decide *exactly* where each boundary should sit between the straddling DPUs. The simplest choice of midway between straddling DPUs is both arbitrary and unjustified.

This uncertainty can be avoided by using both boundary definitions (a) and (b). Denote the boundaries in (a) as *key-based* boundaries [Fig. 6(c)]. After the tile bound DPUs are mapped to the home tile, each tile t can have the wrong number of DPUs $N^{(t)}_{\text{DPU}}$, which we refer to as the tile's *size*. The GA can be made to favor solutions with correct tile sizes of $N^{(t)}_{\text{DPU}} = N_{\text{DPUs/tile}}$. To do this, determine the *physical* boundaries by counting out the desired $N_{\text{DPUs/tile}}$ DPUs per tile; physical boundaries are designated by straddling DPU pairs rather than a specific key value. Any misalignment between the DPUs straddling a key-based boundary versus the corresponding physical boundary is penalized in the same manner as an overlength wire. One can imagine elastics connecting physical and key-based boundaries, pulling them together into alignment. The cost of misaligned

boundaries can be incurred even if no DPUs violate any tile bindings, i.e., a completely feasible solution can have nonzero cost. Since the problem is no longer one of constraint satisfaction alone, violations must be tallied independently from cost. Unlike optimization, however, improved fitness from aligned boundaries does not translate into a better phenotype; it only improves the search by enabling a good remapping scheme for the keys of tile bound DPUs. For example, we intuitively expect less repulsion of tile bound DPUs from the boundaries because of the "fuzziness" of misalignment, which has two contributions. First, the physical boundaries are based on DPU ordering, which depend on their key values in a sloppy way. Second, the straddling DPUs themselves define a range of key values which the key-based boundary can fall between and still be aligned.

The misalignment penalty encourages but does not guarantee alignment, particularly in the early generations of evolution. Bound DPUs that fall between physical and key-based boundaries may violate their tile bindings. To resolve this, we search for a feasible modification of the DPU ordering. If such a fix can be found, the effect of this heuristic (local) search is to remap some solutions from an infeasible regions to nearby points in feasible regions. As mentioned, solutions for resource-limited problems are likely to lie near the interface, making it worthwhile to fix almost-feasible solutions. Since the repair raises infeasible points near the interface to feasibility, repulsion effects from penalization are further mitigated, thus encouraging more search to take place there.

Fig. 7(a)–(c) shows the approach for repairing tile binding violations. Starting from the nearest edge, the home tile is searched for a slot in which the errant DPU, or *swapper*, can be put. The DPU occupying that slot, or *swappee*, can then be swapped with the swapper [Fig. 7(c)]. To be a swappee, the swap should not cause the candidate DPU to violate any tile bindings. If the candidate is already an errant DPU, the swap should not bring it farther away from its home tile. Such a swap could make the chromosome grossly infeasible, since the swappee would be a whole tile farther from its home.

Our GA does not penalize tile binding violations that can be repaired. The tile binding violations can only arise from misalignment of physical and key-based boundaries, which are already penalized. For the same reason, even *unrepairable* tile binding violations are not penalized. Another reason to avoid penalizing repairable violations is the importance of not discouraging search in the vicinity of almost-feasible points.

Unlike PMX repairs, our repairs are applied to the phenotype for the purpose of evaluation only. The repairs are not encoded back into the genotype because changing the chromosome to reflect improvements is like a life form being able to change its own genes due to things learned during its lifetime. Changing the chromosome would allow it to be perpetuated in the evolution. Such *Lamarckian* evolution has been disproved in the biological world, though local optimization of chromosomes has been used in GAs, with controversy [20]. It effectively pushes the chromosome back to the shores of an infeasible region [Fig. 7(d)]; this makes the search greedier and makes it harder to break out of local optima. Studies have shown Lamarckian evolution preventing the discovery of feasible points ([22, p. 321]). We avoid it and leave the solutions in the infeasible region. Penalization of tile edge misalignments will minimize tile violations.

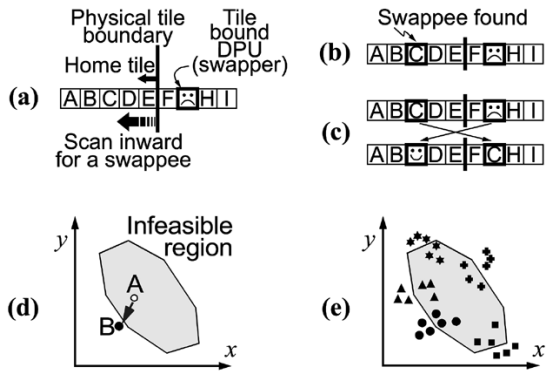


Fig. 7. (a)–(c) Repair of tile binding violations. (d) Lamarckian learning in a 2-D search space. The underlying chromosome is pushed out of an infeasible region by repairs from local search. (e) Non-Lamarckian local search changes the search space by replacing infeasible points with nearby feasible points.

Conceptually, restricting the effects of repair to cost evaluation is equivalent to embedding the repair into the chromosome decoding rather than shifting chromosomes in the search space. It does not change the rules governing how to choose new test points; it modifies the genotype–phenotype mapping to give some of the feasible solutions more representation in the search space [Fig. 7(e)]. Regions in the genotype space that use to be infeasible are replaced by replicas of nearby feasible points. Thus, the infeasible regions shrink and the feasible solutions are easier to find.

When a phenotype is changed by a tile repair, it creates the possibility that previous unrepairable violations become repairable. Therefore, after each repair, the remaining tile violators are rescanned for further repairability.

B. Polarity Bindings

One architectural approach to improve silicon usage is to interleave DPUs with complementary functionality. This can save resources over duplicating all functionality across all DPUs. An example of this is Chameleon's read/write access to memory, alternating between even and odd position DPUs. We call this the position's *polarity*.

Finely interleaved position constraints like polarity require different treatment from gross location constraints such as tile binding. Though *deterministic local* search was used to find nearby feasible variations of tile binding violators, the GAs *global* search was *stochastically* discouraged from putting too many points too deeply into infeasible regions, i.e., where not even local search can find feasible variations. This global evolution does not help much in complying with the fine position constraints of polarity. We assume that polarity constraints have 50% chance of violation, then rely completely on the deterministic local search to fix the violations. Thus, wherever evolution has placed a sampling point in the search space, a nearby polarity compliant solution is sought through a series of small interchanges. Because of the fragility of polarity compliance, it cannot be found by a stochastic search, e.g., if a chromosome c was compliant with many of its polarity

constraints, a single DPU movement can shatter this by shifting many DPUs by one position. The riskiness can be viewed as a random search step from c in a genotype space containing finely alternating regions of feasibility and infeasibility.

The repair heuristic is simple. A polarity violating DPU can reverse its polarity by swapping positions with a neighboring DPU. The following conditions are to be satisfied: 1) the neighbor with the closest key value is preferred; 2) the swap should not create a new polarity violation in the neighbor, though it may fix an existing one; 3) no new tile binding violations should be created, though existing ones may be fixed; and 4) physical tile edges are used to assess tile binding compliance, since key-based edges are fictitious.

Condition 1) minimizes the perturbation in the search space to maintain tight association between the new cost and actual costs available close to the underlying genome. We thought that mapping infeasible points to far-away variations might deceive the stochastic search. We did not worry about whether a tile binding violator is brought further away from its home tile, as the movement is small compared to migration between tiles.

As with tile bindings, if any violations are repaired, the remaining violators are rescanned for further repairability. After an iteration with no repairable violations, unrepairable violations are penalized a small cost of 1. This is the same as a wire exceeding the maximum length by 1, and the violating DPU is only 1 position away from a polarity compliant position. We use a small penalty because we are not relying on GAs stochastic global search to resolve polarity violations. Instead, we rely on the ubiquity of finely interleaved regions of violation and compliance to potentially fix all polarity violations through adjacent moves. The token penalty provides some differentiation with a solution that is equally good except for the polarity violation.

V. Scheduling of GA Steps

The overall framework for the GA is as follows. 1) After generating an initially random population of genomes and evaluating them for cost, a subset of genomes is selected as parents according to fitness; 2) offspring and mutants are generated, evaluated for cost, and merged into the population; 3) the cycle is then repeated, starting with selection of parents according to fitness; and 4) the search stops when a solution is found that does not contain violations of tile bindings or polarity bindings, or outstanding violations of local wirelength limits after global wires have been used to eliminate the worst offenders. Section VI describes an additional empirical stopping condition.

The evaluation of a genome consists of the following steps, in order: 1) the random keys are sorted to obtain the DPU arrangement; 2) tile misalignments are penalized; 3) repairs are repeatedly attempted on tile binding violations until no further fixes are possible; 4) repairs are repeatedly attempted on polarity violations until no further fixes are possible; 5) remaining polarity violations are penalized; 6) violations of local wire length are found and penalized; 7) global wires are used to pardon the worse violators; and 8) outstanding costs and violations are tallied.

TABLE I
Kernels Tested. *MULs*: Multipliers. *DF-I*: Direct Form I. *DF-II*: Direct Form II. *Xpose*: Transposed DF-II. *x4*: 4 Interleaved Streams. Brackets Show Memory Read/Write DPUs. Easy[Hard] = No[Extra] Constraints

Kernel	DPUs	MULs	DPUs+ MULs	Nets	Driven Ports	Tiles
IIR Xpose	7(2)	5	12	11	15	1
IIR DF-II	8(2)	5	13	12	15	1
IIR DF-II x4	14(8)	5	19	15	21	4
FIR Xpose	10(2)	5	15	14	22	1
IIR Xpose x4	13(8)	5	18	14	27	4
FIR DF-I	19(2)	5	24	23	31	1
FFT Easy	20(2)	6	26	24	50	3
FFT	20(2)	6	26	24	50	3
FFT Hard	20(2)	6	26	24	50	3

VI. Results

The proposed GA was implemented in C++/STL. In [24], we initially applied the GA to the netlist of [17], which implements a 1024-point FFT in one slice. Of the kernels tested, it was the most complex, consisting of a central butterfly engine, address generation for twiddle factors and data undergoing transformation, and selective interleaving to maintain 1 butterfly/cycle. The kernel's use of array resources is summarized in Table I, second last row. We comprehensively characterized its convergence behavior across several parameters, including the ***f***raction of the ***p***opulation to replace with ***k***ids (PFK) in each generation, as well as the ***f***raction of each generation's ***k***ids to be made up of ***o***ffspring (KFO) rather than mutants. A population size of $N_{\mathrm{Pop}} = 200$ was used, based on preliminary trials. Aside from the extremities, the algorithm was robust across both parameters' value ranges [Fig. 8(a)–(c)]; 50% PFK was recommended, with 30%–50% KFO. Here, we briefly describe some probable reasons for this "sweet spot." Lower PFKs require more generations, due to the lesser exploration per generation [Fig. 8(a)], but this is offset somewhat by the lesser work per generation. Thus, there is no overriding trend in run time across the PFK range, though 50% PFK seems the most robust with respect to different KFO [Fig. 8(b)].

For the FFT, the combination of high KFO and low PFK suffers more failed searches [Fig. 8(c)]. This combination yields a greedy search. High KFO means more exploitation by generating offspring rather than diversification by generating mutants. Low PFK leans toward a steady-state replacement strategy, which again is greedy because most of the population remains undisplaced between generations. For tough problems such as the FFT, an overly greedy search can impede escape from local optima.

Fig. 8(d) shows that aside from KFO, diversification can be obtained by limiting P_{b2w} (***p***robability ***b***est to ***w***orst genome, Section III-C). This was done under steady-state replacement, thus the dramatic drop in success for low mutation rates. This result is given to show how easily selective pressure can be controlled by ranked fitness.

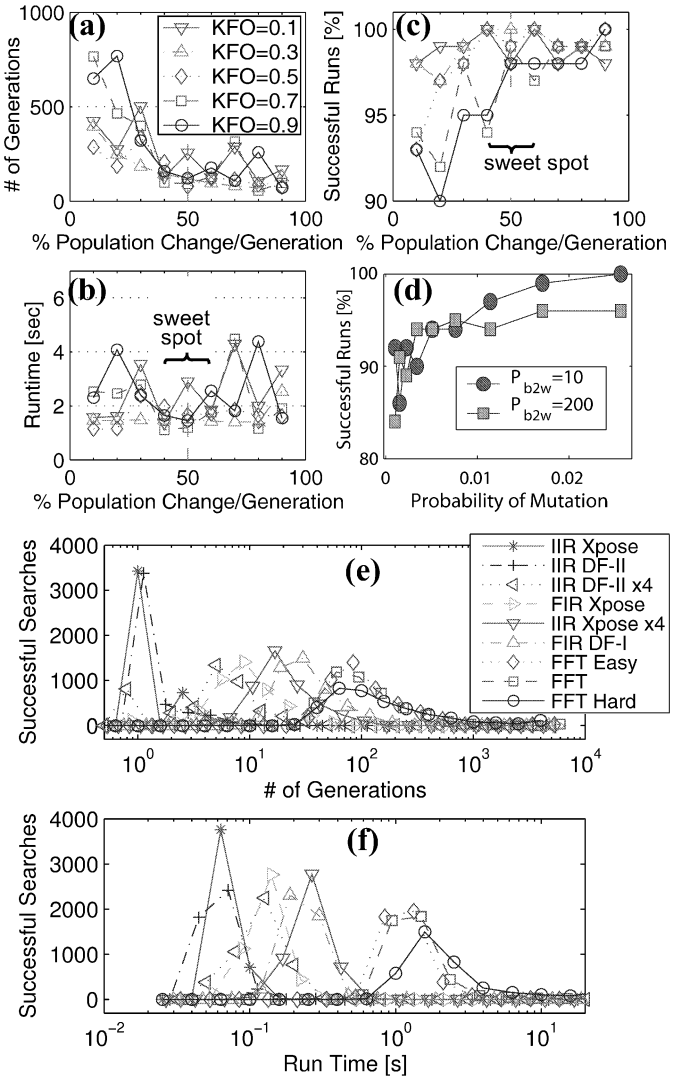


Fig. 8. (a)–(d) Convergence of successful searches for the FFT kernel. (e)–(f) Distribution of search times for placement for different kernels. Histogram data was generated in the $\log x$ domain.

A. Other Kernels

We applied our GA to a suite of kernels designed at a high level, i.e., datapath design [Table I, Fig. 8(e)]. This includes modifications of the FFT, a 9-tap symmetric finite impulse response (FIR) filter, and an infinite impulse response (IIR) biquad (second order) [25]. Due to latency in the feedback loop, all the IIRs require 4-slow timing [26]; the "×4" filters use this to interleave four data sets on the same filter [27]. This imposes extra placement constraints on memory read/write DPUs, since the memories (Fig. 1) must be chained into 4 data sets within the slice.

As in the FFT, the performance of the search was found to be fairly robust across most of the KFO and PFK range. Success rates for the FIR/IRR kernels were all 100%. This means that a solution was found to meet all constraints before the termination condition (no improvement in the minimum number of violations for 35 000 generations). This termination condition was empirically determined to allow the capture of most of the data points in the search time distributions. Its adequacy is evident in the distributions of Fig. 8(e)–(f), obtained by aggregating results from KFO $\in \{0.1, 0.3, \ldots, 0.9\}$ and PFK $\in \{0.1, 0.2, \ldots, 0.9\}$. One hundred searches were performed for each KFO/PFK combination, thus yielding 4500 searches per curve. Because of the ubiquitous 100% success for the nonFFT kernels, the caveat against high-KFO/low-PFK does not apply. The searches even tended to be slightly faster in this regime. It seems that for less difficult problems, the search space is more densely populated with feasible solutions, and a greedy search is beneficial.

Kernels are listed in Fig. 8(e)–(f) and Table I in order of difficulty of placement, as determined by their search time distributions. There is a strong correlation between placement difficulty and the intuitive measures of circuit complexity in Table I. The FFT kernels were changed by adding/removing constraints. "FFT Easy" has no tile/polarity constraints, while "FFT Hard" has more than necessary. The FFT distributions shrink with greater difficulty due to failed searches.

The difficulty rankings in Table I make intuitive sense. In rows 1–2, the easiest kernels are the simple IIR kernels, which have no high fan-in or fan-out nodes, and use very few resources. FIR Xpose is harder to place than the simple IIRs because it represents a folded 9-tap filter rather than a 5-tap biquad. More algorithmic delays and adders yield a more complex circuit. The "×4" IIRs in row 4 and 6 are more complicated than their noninterleaved counterparts because there are 4 times as many memory read/write DPUs. Furthermore, these DPUs are constrained to the tiles for their associated data sets. FIR DF-I is the most complicated of the non-FFT kernels to place (and in particular, more complicated than FIR Xpose) because this structure uses an adder tree, which is known to be hard to organize physically. The adder tree also requires its own DPUs; in contrast, the adders in a DF-II structure are in-line with the delay line, and can share DPUs with algorithmic delays. The FFT and its variants are the most complex.

B. Varying Population Size, Array Size, and Iteration Scheme

The problem and the algorithm were varied to gauge the the performance and the problem difficulty. The most complex kernel was used (FFT).

Population sizes can be used to control diversity. $N_{\mathrm{Pop}} \in \{50, 100, 200, 400, 800\}$ were tested, yielding search times from tenths of seconds to several seconds. Larger N_{Pop} needed fewer generations to find a solution, but took more time overall. This suggests the use of small N_{Pop}. A caveat is that the successful searches dropped from nearly 100% at $N_{\mathrm{Pop}} = 800$ to 87% for $N_{\mathrm{Pop}} = 50$. Adjacent population sizes had run time distributions that overlapped considerably; hence, search times are sensitive to N_{Pop}, but not extremely so.

For the same set of N_{Pop}, the effect of problem size was explored by nearly doubling the tile size, to 16 DPUs. A preliminary design of a back-to-back FFT/FFT was used for the test, nearly doubling the size of the FFT kernel. (This emulates the complexity of a more plausible back-to-back FFT/IFFT). Most of the run times ranged from just under 1 s to 40 s, about an order of magnitude more than the baseline FFT. This is still within the realm for an interactive CAD environment. As above, small

N_{Pop} finds solutions faster, but fails more often. Overall success rates are degraded by doubling the size, ranging from 45% to 82%. This makes it important to adopt a strategy of early termination and restart of searches. For successful searches, search times are not extremely sensitive to N_{Pop}.

The GA is just one iteration framework. Our initial studies of SA shows it to be a promising iterator for further research. Many of the discussed concepts of problem encoding and hybridization with local search/repair also apply. After empirically tuning the shape of the cooling curve and the variation of search neighborhood with temperature, SA was found to perform almost identically to GA in terms of probability of successful search versus run time.

As an indication of problem difficulty, the GA and SA were compared to a random search. The greater the ratio $V2I$ of ***v***alid solutions to ***i***nfeasible points in the search space, the faster a solution is found. Chameleon's slice has approximately 27 DPUs, yielding 10^{28} arrangements of DPUs for kernels that take up the entire slice. A random search of any practical duration has negligible chance of revisiting a solution point, and is the same as the early stages of a randomly ordered exhaustive search. Three kernels were tested: 1) FFT; (2) FFT/FFT in the double size slice; and 3) FFT without repair. In no case was a solution found by the random search after running for days and evaluating billions of solution points. Not even repair of violations increases $V2I$ enough for a solution to be found by the lengthy beginnings of an exhaustive search. Therefore, an intelligent iteration scheme such as GA or SA is needed.

VII. Conclusion

We identify the generalizable features of a coarse-grain reconfigurable array, such as location-specific resources at fine and gross levels. A GA approach was investigated for placement because of its flexibility in handling architectural constraints, particularly as platforms evolve. The chromosome was encoded using random keys to avoid the need for special measures for ordering problems. Mating was performed with uniform crossover because of the distributed building blocks.

A baseline GA was set up based on a netlist's realizability using local wires. The fitness nonlinearity of constraint driven problems was circumvented via ranked fitness, which also allowed for controlled and predictable selective pressure. The heterogeneous distribution of array resources was handled by binding constraints. Penalization was used to minimize gross position violations, while non-Lamarckian local search was used to fix all other violations. Tile constraints were handled by remapping the position-specifying genes to subranges for the home tile. Tile sizes were regulated by penalization. Local search heuristics are presented to repair remaining violations of fine-and gross position constraints.

The GA performed robustly on several kernels over a wide variety of search conditions. Search times for a doubled array size and kernel size was still within the realm of interactive CAD, though early restarts may be needed due to more failed searches. Simulated annealing was found to perform similarly. Prolonged random searches were unable to find a solution. This indicates that intelligent iterators like GA/SA are required, due to the sparsity of feasible solutions.

Future work includes extending the GA to more completely capture the constraints and functionality that can be expected in reconfigurable platforms. For example, multipliers are expensive and available in only a few locations in Chameleon. They also have limited functional overlap with regular DPUs. Therefore, they should be handled differently from regular DPUs. Placement of the chainable memories should also be handled by the algorithm. GA parameters KFO and PFK can also be allowed to vary according to search progress, as can be the degree of mutation, e.g., by more than one gene.

Acknowledgment

The authors would like to thank E. Hum and D. Starks at Nortel Networks for technical support. They would also like to thank M. Rollins, R. Karamchedu, C. Cump, and B. Van Dyje, formerly of Chameleon Systems, for technical support. Finally, the anonymous reviewers deserve thanks for suggestions that greatly improved this paper.

References

[1] R. Hartenstein and R. Kress, "A datapath synthesis system for the reconfigurable datapath architecture," in *Asia and South Pacific Design Automation Conf. (ASP-DAC) 95*, 1995, pp. 479–484.

[2] H. Singh, M.-H. Lee, G. Lu, F. Kurdahi, N. Bagherzadeh, and E. C. Filho, "Morphosys: An integrated reconfigurable system for data-parallel and computation-intensive applications," *IEEE Trans. Comput.*, vol. 49, no. 5, pp. 465–481, May 2000.

[3] B. Salefski and L. Caglar, "Re-configurable computing in wireless," *Design Automation Conf.*, pp. 178–183, 2001.

[4] L. Miyamori and K. Olukotun, "REMARC: Reconfigurable multimedia array coprocessor," *IEICE Trans. Inform. Syst.*, vol. E82-D, pp. 389–397, 1999.

[5] C. Ebeling, D. C. Cronquist, P. Fraklin, and C. Fisher, "RaPiD—Reconfigurable pipelined datapath," in *Proc. 6th Int. Workshop on Field-Programmable Logic and Applications*, 1996, pp. 126–135.

[6] E. Mirsky and A. DeHon, "MATRIX: A reconfigurable computing architecture with configurable instruction distribution and deployable resources," in *IEEE Symp. FPGAs for Custom Computing Machines*, K. L. Pocek and J. Arnold, Eds. Los Alamitos, CA: IEEE Comput. Soc. Press, Apr. 1996, pp. 157–166.

[7] M. B. Taylor, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, J. Kim, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal, "Evaluation of the raw microprocessor: An exposed-wiredelay architecture for ILP and streams," in *Proc. 31st Annu. Int. Symp. Computer Architecture*, Munchen, Germany, Jun. 2004, pp. 2–13.

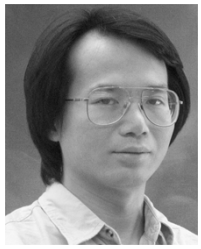
[8] R. Hartenstein, M. Herz, T. Hoffmann, and U. Nageldinger, "Mapping applications onto reconfigurable KressArrays," presented at the *9th Int. Workshop on Field Programmable Logic and Applications (FPL'99)*, Glasgow, U.K., Aug.–Sep. 1999, pp. 385–390.

[9] T. Callahan, P. Chong, A. DeHon, and J. Wawrzynek, "Fast module mapping and placement for datapaths in FPGAs," in *Proc. ACM/SIGDA Int. Symp. Field Programmable Gate Arrays*, Monterey, CA, 1998, pp. 123–132.

[10] A. Sharma, "Development of a place and route tool for the RaPiD architecture," M.S. thesis, Univ. Washington, Seattle, WA, 2001.

[11] L. McMurchie and C. Ebeling, "Pathfinder: a negotiation-based performance-driven router for FPGAs," in *Proc. Association for Computing Machinery Int. Symp. Field Programmable Gate Arrays (FPGA'95)*, 1995, pp. 111–117.

[12] A. Sharma, "Piperoute: A pipelining-aware router for FPGAs," in *Proc. Int. Symp. FPGAs (FPGA'03)*, Monterey, CA, 2003, pp. 68–77.
[13] J. Lee and N. Dutt, "Compilation approach for coarse-grained reconfigurable architectures," *IEEE Des. Test. Comp.*, vol. 20, no. 1, pp. 26–33, 2003.
[14] G. Venkataramani, W. Najjar, F. Kurdahi, N. Bagherzadeh, W. Bohm, and J. Hammes, "Automatic compilation to a coarse-grained reconfigurable system-on-chip," *ACM Trans. Embedded Computing Systems*, pp. 560–589, Nov. 2003.
[15] W. Bohm, J. Hammes, B. Draper, M. Chawathe, C. Ross, R. Rinker, and W. Najjar, "Mapping a single assignment programming language to reconfigurable systems," *J. Supercomput.*, vol. 21, pp. 117–130, 2002.
[16] M. Gordon, W. Thies, M. Karczmarek, J. Lin, A. Meli, A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe, "A stream compiler for communication-exposed architectures," in *Proc. 10th Int. Conf. Architectural Support for Programming Languages and Operating Systems*, vol. 37, 2002, pp. 291–303.
[17] F. Ma, J. Knight, and C. Plett, "Reconfigurable logic design case," in *Proc. 4th SPIE Conf. Reconfigurable Technology: FPGAs and Reconfigurable Processors for Computing and Communications (Part of ITCom 2002)*, vol. 4867, Jul. 2002, pp. 113–126.
[18] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading, MA: Addison-Wesley, 1989.
[19] L. R. Foulds, *Combinatorial Optimization for Undergraduates*. New York: Springer-Verlag, 1984.
[20] D. Whitley, "A genetic algorithm tutorial," *Stat. Comput.*, vol. 4, pp. 65–85, 1994.
[21] M. Gen and R. Cheng, *Genetic Algorithms and Engineering Design*. New York: Wiley, 1997.
[22] Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs*, 3rd ed. Berlin, Germany: Springer-Verlag, 1996.
[23] H.-P. Schwefel and G. Rudolph, "Contemporary evolution strategies," in *Proc. Eur. Conf. Artificial Life*, 1995, pp. 893–907.
[24] F. Ma, J. P. Knight, and C. Plett, "Physical resource binding for a coarse grain reconfigurable array," in *Proc. Engineering and Reconfigurable Systems and Algorithms*, Jun. 2004, pp. 109–115.
[25] D. G. M. John and G. Proakis, *Digital Signal Processing*, 3rd ed. Upper Saddle River, NJ: Prentice-Hall, 1996.
[26] C. E. Leiserson and J. B. Saxe, "Retiming synchronous circuitry," *Algorithmica*, vol. 6, no. 1, pp. 5–35, 1991.
[27] K. K. Parhi, *VLSI Digital Signal Processing Systems*. New York: Wiley, 1999.



Fred Ma received the M.A.Sc. degree from the University of Waterloo, Waterloo, ON, Canada, in 1996 and the Ph.D. degree on EDA algorithms for coarse-grain reconfigurable logic at Carleton University, Ottawa, ON, Canada.

He has worked academically and with industry on CCDs and simulation methods for guided wave electromagnetics.

John P. Knight received the M.A.Sc. degree from Queen's University, Kingston, ON, Canada, and the Ph.D. degree from the University of Toronto, Toronto, ON, Canada.

He is a Distinguished Research Professor (retired) in the Electronics Department at Carleton University, Ottawa, ON, Canada. His research has been mainly in digital circuits, especially in the area of high-level synthesis, initially using heuristic and later evolutionary algorithms. He has authored or coauthored about 36 technical papers, mainly in this area and in low-power circuits.

Calvin Plett received the B.A.Sc. degree in electrical engineering from the University of Waterloo, Waterloo, ON, Canada, in 1982, and the M.Eng. and Ph.D. degrees from Carleton University, Ottawa, ON, Canada, in 1986 and 1991, respectively.

In 1989, he joined the Department of Electronics, Carleton University, where he is now an Associate Professor. He has been involved with various companies including Nortel, Sige Semiconductor, Philsar, Conexant, Skyworks, and IBM, both as a consultant and for student placement for cooperative research. He has authored or coauthored about 60 technical papers and is a coauthor of the book *Radio Frequency Integrated Circuit Design* (Norwood, MA: Artech House, 2003). His research interests include the design of analog and radio-frequency integrated circuits, including filter design, and communications applications.