ColSpace: Towards Algorithm/Implementation Co-Optimization

Jiawei Huang, John Lach

Department of Electrical and Computer Engineering
University of Virginia
{jh3wn, jlach}@virginia.edu

Abstract—**Application-specific integrated circuits (ASICs) are physical implementations of algorithms, so implementation metrics are determined in large part by the algorithm specification. However, the system abstraction layers that have been developed to manage the ever-increasing complexity of digital systems separate algorithm designers from hardware designers, forcing the latter to work within the design space specified by the former, even for applications such as multimedia that do not have hard fidelity requirements. Designers typically employ informal iterative design to adjust fidelity, but a formal design methodology would increase designer efficiency and improve the quality of the solutions.**

This paper introduces such a methodology (and accompanying tool) that enables algorithm and implementation metrics to be co-optimized during early design exploration, opening the design space to include solutions that may provide, for example, significant performance improvements while only slightly compromising fidelity. Hierarchical dependency graphs (HDGs) are used to represent both the algorithm and the implementation architecture, providing a common interface through which algorithm designers and hardware designers can explore the collaborative space (ColSpace) together. Using the proposed technique, the ColSpace tool can trade off various metrics to find the best overall design while managing complexity with the HDG hierarchy. Two image processing case studies demonstrate that in ColSpace-optimized designs, latency savings can exceed fidelity losses, resulting in cost function reductions that would not have been possible without this co-optimization methodology.

I. Introduction

Dramatic increases in system complexity have led to the development of design methodologies that partition optimization efforts into separate, independent layers of abstraction. A great number of advances have led to strong optimization capabilities at each layer, but solutions that require cross-layer optimization are often neglected. This is problematic given that some of those solutions may yield better overall system metrics, including both application and implementation metrics.

Consider, for example, the design of an automatic target recognition system that has application metrics such as false recognition rate and implementation metrics such as latency and power. In traditional design methodologies, image processing engineers would design the algorithm with the goal of optimizing the application metrics, and the algorithm would then be delivered to hardware designers, who would in turn work to optimize the implementation metrics. Given the independence of abstraction layers, hardware designers are typically forced to work within the constraints of the specified algorithm, despite the fact that many applications (e.g. multimedia, probabilistic heuristics, etc.) could tolerate slight fidelity degradation if it enabled significant improvements in implementation metrics. Application examples include not only traditional multimedia but also probabilistic heuristics, such as classification, detection, and tracking applications.

Modern tools help hardware designers adjust implementation "knobs" (e.g. source voltage, component configurations, number of pipeline stages, etc.) to find and traverse the Pareto optimal curve, which is conceptually a hyper-surface with one dimension for every metric of interest. If fidelity is included as an additional dimension, the resulting N+1-dimensional hyper surface effectively creates an infinite number of N-dimensional cross-sections, some of which may include solutions more desirable than those on the original curve based on the initial algorithm specification.

What thwarts the exploration of this expanded design space is generally poor collaboration between algorithm designers and hardware designers. Due to the independent system abstraction layers, it is rare that a designer working in one area has enough knowledge of the other to reason about such tradeoffs. In addition, it is difficult for these groups to communicate efficiently with one another, as they use different languages and formalisms to represent their ideas. Therefore, the evaluation of how an implementation decision affects the algorithm and, by extension, the application fidelity is a challenge. (Typically only a uni-directional communication is performed, i.e. how algorithm changes affect implementation metrics). Manually tuning the algorithm by experienced designers to achieve overall optimization is increasingly difficult with the growing algorithmic complexity. A more formal methodology would improve both designer efficiency and solution quality.

It is especially important to explore the co-optimization space early in the design flow. While detailed low-level designs provide more accurate estimates of implementation metrics to evaluate each design option, they are too time consuming to create and alter. As a result, only a small portion of the design space can be practically explored.

This paper introduces ColSpace – a new design methodology (and accompanying tool) for early design space algorithm-implementation co-optimization with an emphasis on dataflow intensive applications (latency more sensitive to simplifying computation). Hierarchical dependency graphs (HDGs) are used to represent both the algorithm and its hardware implementation, providing a common data structure through which hardware designers and algorithm developers can communicate during design space exploration. An automated design exploration algorithm is designed to operate on these data structures and is able to identify and evaluate candidate fidelity-compromising transformations. This methodology is demonstrated with two image processing case studies, which produced solutions that provide significant reductions in latency with small impacts on fidelity.

The remainder of this paper is organized as follows. Section II is a review of related work. Section III introduces the HDG data structure for design space representation. Section IV presents the proposed algorithm for design space exploration and co-optimization. This algorithm is demonstrated with two case studies in Section V. We draw conclusions in Section VI.

II. Related Work

Much of the prior work on methodologies for implementing algorithms written in a high-level languages (e.g. Matlab, C, etc.) has been focused on compilation and code conversion, such as the AccelDSP tool from Xilinx. However, these methodologies are simply one-way translations. As a result, if a hardware designer wants to explore an implementation that potentially compromises fidelity, it can be extremely difficult to communicate the desired change to the algorithm designer who must evaluate the application impact. To address this lack of bi-directional communication in existing methodologies is a primary motivation of this work.

ColSpace is not the first tool to employ dependency graphs (also known as task graphs) for IC design. SPARK [1] is a high-level synthesis framework that converts an ANSI-C behavioral description into synthesizable RTL VHDL using a hierarchical task graph as an intermediate representation. Vallerio and Jha developed an automatic task graph extraction tool to facilitate the development of embedded systems [2].

Karankonstanis et al. [3] studied the tradeoffs between quality/power-dissipation/yield in a color interpolation algorithm. They suggested omitting non-critical components of the system in order to achieve better power efficiency. While this technique can be applied to other signal processing applications, our methodology is more formalized and extensible. The technique of omitting non-critical components can just be a fidelity-compromising transformation in our library.

ColSpace is also different from traditional HW/SW codesign, which focuses more on identifying the HW/SW partition than on making any fundamental changes to the specified algorithm. ColSpace is therefore orthogonal to, and can be applied in, the codesign paradigm.

III. Design Space Representation

The HDG is the data structure that provides a representation of both an algorithm and its implementation, enabling bi-directional communication between the algorithm and hardware designers for ColSpace exploration. It shares some similarities with traditional task graphs [2, 11, 13] but is structured hierarchically to manage design complexity. At the top level of the graph, each node represents an algorithmic task (e.g. edge detection, noise removal, etc.). Subsequent levels are similar, with nodes representing sub-tasks, all the way down to the most basic operations, such as additions and multiplications.

The hierarchical structure of the HDG is also suitable for describing the initial structure of the hardware architecture. Each node in the HDG can represent a hardware functional unit that performs the corresponding algorithmic operation. Each hardware unit has a set of metrics (latency, area and power) associated with it. Designers can experiment with various low-level components (e.g. adders, multipliers, and other fundamental components) by altering the metrics associated with the nodes, and the metrics are automatically traced up to the necessary levels at which implementation metrics can be evaluated. This allows rapid evaluation of the impacts of architectural changes at the system level.

HDGs offer bi-directional algorithm-implementation communication: the impact of algorithmic/implementation alterations on hardware/fidelity metrics can be evaluated. Traditional design methodologies only support communication in one direction (algorithm→ implementation), thus limiting the quality of systems they can produce. In addition, this bi-directionality also leaves open the possibility for automatic algorithm code updates based on HDG alterations, facilitating the re-simulation of the altered algorithm.

The ColSpace tool comes with some common library tasks/blocks pre-loaded to ease HDG development, and the library can be extended by the designer. The tool is available for download [16] - the current version supports only latency and throughput implementation metrics, with area and power to be added soon.

The rest of the paper will focus on latency-fidelity tradeoff exploration. Assuming infinite hardware resources, we will maximize parallelism by duplicating hardware units to achieve an initial lowest-latency implementation. The proposed algorithm seeks alternative implementations that can achieve even lower latency with an acceptable fidelity penalty. Note that an area-fidelity tradeoff requires additional information, such as the scheduling and mapping of operations to functional units, and the design space is much bigger than that of the latency-fidelity tradeoff.

While ColSpace is designed to handle dataflow intensive algorithms, it is also capable of representing simple control dependencies. Loops (e.g. *for* and *while* loops) are modeled as special nodes within the HDG. Descending a level into a loop reveals the subgraph of one loop iteration. The latency of a regular loop is the latency of one iteration times the number of iterations. But in order to identify and represent parallelism, the tool can unroll loops and/or denote them as *ordered/unordered* and *concurrent/nonconcurrent* [7]. "Unordered" means the iterations can be executed in any order; "concurrent" means several iterations can be executed simultaneously (i.e. no inter-iteration dependencies). The latencies for each loop type are calculated accordingly. Many loops in image and signal processing algorithms are both unordered and concurrent, an ideal property for extracting parallelism.

IV. Design Space Exploration with ColSpace

Now that design representation has been addressed, this section introduces the design exploration algorithm through HDG manipulations that include fidelity-compromising alterations. Design evaluation methods are also presented as they provide necessary information for successful exploration.

As discussed in Section I, *fidelity-compromising* methods can be extremely powerful and provide a primary motivation for ColSpace, but a methodology is necessary to use them to achieve design objectives, just like a synthesis algorithm for applying various non-functional optimizations. Fidelity-compromising transformations are designed to replace an implementation of an algorithm routine (i.e. a function) with other implementations with similar results but much lower latencies. For example, replacing $\frac{1}{1+x}$ with $1-x$ reduces computation time, but uses a value not completely accurate (x is constrained in the range of (-1,1)). These transformations are necessary vehicles for exploration along the fidelity axis in the design space. Algorithms in many application domains such as signal processing and numerical analysis typically involve such mathematical expressions and the final result is not required to exact. Fidelity-compromising transformations can be parameterized by a threshold (such as x in the previous example) to control the level of inaccuracy being introduced. Fidelity-compromising transformations are usually obtained in two ways. A general approach uses mathematical approximation theory, while an application-specific approach uses multiple algorithms that accomplish the same goal in a different way (such as Canny edge detector versus Sobel edge detector). Although HW designers often manually attempt such simplifications during the optimization stage, it is largely done ad hoc and its quality cannot be guaranteed. Furthermore, when the algorithm becomes more complex, HW designers often can no longer understand the algorithm in sufficient depth to identify such transformations. A formal method is necessary for joint collaboration between algorithm and implementation teams. In an effort to provide a list of available transformations for the design space exploration algorithm and to intelligently explore the design space, the tool can indentify latency bottlenecks within a design and direct the search algorithm to focus on developing transformations around those bottlenecks. Table 1 shows how to specify a fidelity- compromising transformation in the library.

Table 1. An example fidelity-compromising transformation

before	$\frac{1}{1+x}$			
after	1	$1-x$	$1-x+x^2$	$1-x+x^2-x^3$
p1 (order)	0	1	2	3
p2 (threshold)	A			
condition	$\|x\| < A, A \in (0,1)$			
|Δlat|	11	10	9	8
|Δfid|	A	A^2	A^3	A^4

Each transformation should at least contain: the expression before and after the transformation, a set of tunable parameters (two parameters in the above example: the order of approximation and the threshold of the condition), the condition to apply the transformation and the resultant latency and fidelity changes (Δlat, Δfid). The latency and fidelity changes can either be constants or functions of the parameters. Optional code realizations can also be included to allow automatic source code update (see subsection D). It is quite obvious that greater latency saving is accompanied by higher fidelity degradation.

Problem statement: given the following:

1. an HDG describing an algorithm and its lowest latency implementation and a runnable source code,
2. a library containing applicable fidelity-compromising transformations (in the same format as Table 1), $T_n : (\Delta FID_n, \Delta LAT_n, p_1, p_2, ..., p_{H_n})$ where H_n denotes the number of thresholds that can be tuned for transformation T_n
3. a latency evaluation routine that gives total latency LAT_{total} and a fidelity evaluation routine that gives global fidelity FID_{total}, and
4. a global cost function C to be minimized,

the goal of the design space exploration algorithm is to determine:

1. which transformations to apply,
2. the proper settings for each transformation, and
3. the order in which to apply those transformations

so as to minimize C.

To obtain the initial lowest latency implementation, an HDG is set up to represent the dependency information of the algorithm. Hardware parallelism will be fully utilized so that independent operations can always be executed concurrently by replicating hardware units. Hidden parallelism (identified by the "Exploit Parallelism" feature of the tool, discussed below) will also be taken into consideration. The global latency is determined by the critical sequential path present in the design. No fidelity-compromising transformation will be applied yet in this process.

A. Algorithm Overview

The design space exploration problem is formalized in a way that any optimization method can be utilized to solve it. Here a greedy algorithm is presented to showcase a typical search procedure and some common caveats. The algorithm has two variants: constraint-based and cost-function-based. To demonstrate the former, consider the following design question: what is the minimum latency $LAT_{\min}$ if we allow up to 10% fidelity degradation?

- *For every fidelity-compromising transformation T_n, tune its thresholds $P_n = (p_{(n,1)} ... p_{(n,H_n)})$ so that the global fidelity reduction $\Delta FID_{total} = 10\%$*
- *While P_n is set, calculate resultant global latency LAT_{total} using profiling information.*
- *Find the transformation T_k that gives minimum latency*

If multiple threshold settings P_n can result in

$FID_n = 10\%$, all of them will be tried for calculation and only the lowest LAT_{total} is recorded.

For cases where the objective is to minimize a latency-fidelity cost function, such as $\frac{LAT_{total}}{FID_{total}}$ (again, a greedy algorithm is used for demonstration):

```
1. For every fidelity-compromising transformation $T_n$,
   tune its thresholds $P_n = (p_{(n,1)} \cdots p_{(n,H_n)})$ so as to
   minimize $d(\frac{LAT_{total}}{FID_{total}})$. If calculating $d(\frac{LAT_{total}}{FID_{total}})$ is
   hard, we can apply l'Hôpital's rule using $d(LAT_{total})$
   and $d(FID_{total})$.
2. Find the transformation $T_k$ with minimum $d(\frac{LAT_{total}}{FID_{total}})$.
   If that value is still positive, end algorithm, return
   current $P$ settings.
3. Apply $T_k$ with corresponding $P_k$.
4. Re-evaluate $F_n$ and $L_n$ for every transformation.
```

The derivatives can be calculated against a discretized domain (to be demonstrated in Section V). As long as the step size is chosen small enough so as the cost function is monotonic in this range, global optimal point can still be properly located.

There are two ways to reduce the simulation time, but both come with some impact on accuracy. One is to apply only uncorrelated transformations. Transformations that are uncorrelated can be applied separately and their combined effect on LAT and FID will be the same as applying them simultaneously. Another way is to calculate a local disturbance error of the fidelity from the transformation and propagate that error across the entire design to obtain a global error estimate. Symbolic Noise Analysis (SNA) [12] can accomplish this task without simulation.

B. Latency Evaluation Routine

There are two types of latencies used in the latency evaluation routine. First, every node in the HDG is assigned a *delay (LAT)* value. This value is defined in the library for basic nodes such as adders and multipliers; it reflects the relative speed of execution of every functional unit at the logic level. The unit is control step, or c-step, which is a multiple of the clock period. A *path delay (PLAT)* is the accumulative delay along a data path formed by nodes and edges. The path is specified by a pair of nodes (N_{start}, N_{finish}), where N_{start} and N_{finish} are the beginning and end of the path, respectively. The latency of a composite node N (a node with sub-structures) is computed as:

$$LAT_N = \max(PLAT(N_{input}, N_{output})) \tag{1}$$

which is the length of the critical path from N's inputs to its outputs.

Fidelity-compromising transformations allow a node to have multiple LAT values corresponding to different alternative implementations. If the pertinent transformation is controlled by a threshold, then each alternative implementation receives a percentage of execution. The aggregate latency for that node is given by

$$LAT_{total} = \sum_n LAT_n W_n \tag{2}$$

where W_n is the percentage of execution of transformation n. For correlated transformations, a node with multiple alternative designs cannot be substituted by an aggregate node. Each combination of alternative designs for all correlated transformations must be considered. For instance, if transformation T_k has A_k alternative designs, there are a total of $M = \prod_k A_k$ execution paths, each with latency LAT_m and corresponding execution percentage W_m. The aggregate latency for these nodes is given by

$$LAT_{total} = \sum_{m=1}^{M} LAT_m W_m \tag{3}$$

It can be easily shown that this formula reduces to simple aggregate node formula (2) when all transformations are uncorrelated:

$$LAT_{T_1T_2} = LAT_{T_1} + LAT_{T_2} \tag{4}$$

$$W_{T_1T_2} = W_{T_1} W_{T_2} \tag{5}$$

For nodes involving control flow, since the number of iterations of a loop may not be known at design-time, the application can be profiled to determine the mean or worst-case number of iterations, or the number of iterations can be modeled as a variable. In the latter, a latency analysis will prompt the user to enter a value for each variable in the HDG (which can again be estimated via profiling), enabling a quick analysis for a number of datasets. In the case of a branch, both paths are represented, with the longer path defining the latency.

C. Exploit Parallelism

Although hierarchical design allows manageable control of complex systems, it hides parallelism across hierarchical boundaries. Consider the case where a composite node *N* is composed of another two composite nodes N_1 and N_2 with the following relation:

$$LAT_N = LAT_{N_1} + LAT_{N_2} \tag{6}$$

However, if the critical path of N_1 and N_2 are not both on *N*'s critical path at the same time, there exists potential opportunity for structural alteration of N_1 and N_2 so that *N*'s delay can be lowered. This opportunity is detected by comparing the following two values:

$$D_i := PLAT(A, N_i) - RPLAT(A, N_i) \tag{7}$$

and

$$D_{i+1} := PLAT(A, N_{i+1}) - RPLAT(A, N_{i+1}) \tag{8}$$

N_i and N_{i+1} are two immediately neighboring nodes connected by some edges. *A* is predecessor node of N_i. Here, *RPLAT* refers to *recursive path delay*. It differs from PLAT in that it flattens the path between starting node and finishing node to the bottom hierarchical level while evaluating the path delay.

There is a hidden parallelism if and only if the following holds:

$$D_i \neq D_{i+1} \quad (9)$$

Our initial HDG will be preprocessed by "exploit parallelism" (automated in the tool) to take into account all hidden parallelisms while maintaining hierarchy, and latency will be reduced to theoretical minimum. In the resultant HDG, PLAT and RPLAT will return equal values.

D. Fidelity Evaluation Routine

In most applications, fidelity metrics can only be obtained by running an actual implementation of the algorithm and evaluating its output. A pure software approach is usually preferred due to its ease of development. Fidelity-compromising transformations will be implemented in the target programming language, and there is a switch statement wherever one or more fidelity-compromising transformations are applicable so that a simulator can easily run the simulation with different transformations turned on and off. For transformations with thresholds, a profiling counter should be kept to record the percentage of executions going down each path.

The fidelity evaluation routine can be automated by including replacement code segments as part of the transformation library. Whenever a transformation is applied in the HDG, the corresponding code segment will be looked up in the library and enabled in the source code.

E. Scalability

The state-space explosion problem is connected with any state exploration algorithm. This paper does NOT propose a new algorithm that increases the exploration efficiency. Instead, the proposed algorithm is intended for exploring an expanded design space in a new direction. It faces the same scalability issue but it has the potential to yield better designs than the most efficient exploration algorithms in the restricted design space. In addition, the ColSpace tool can help focus designer and exploration algorithm effort by identifying implementation metric problem areas.

V. Case Studies with Imaging Algorithms

To demonstrate the utility and power of the ColSpace methodology, it was applied to two image processing case studies – a speckle reducing algorithm and a tracking algorithm.

A. SRAD

SRAD (Speckle Reducing Anisotropic Diffusion) [8] is a moderately complicated image processing algorithm tailored to ultrasonic and radar imaging applications. It aims to remove multiplicative noise in imagery. The images in Figure 1 illustrate the SRAD process. The fidelity metric compares the reconstructed image (d) to the original (a). Real-time processing requires a minimum frame-per-second throughput [9], thus defining a frame latency constraint. The Matlab implementation of SRAD is significantly slower than real-time [9], so latency reduction is the focus of this algorithm- implementation co-optimization.

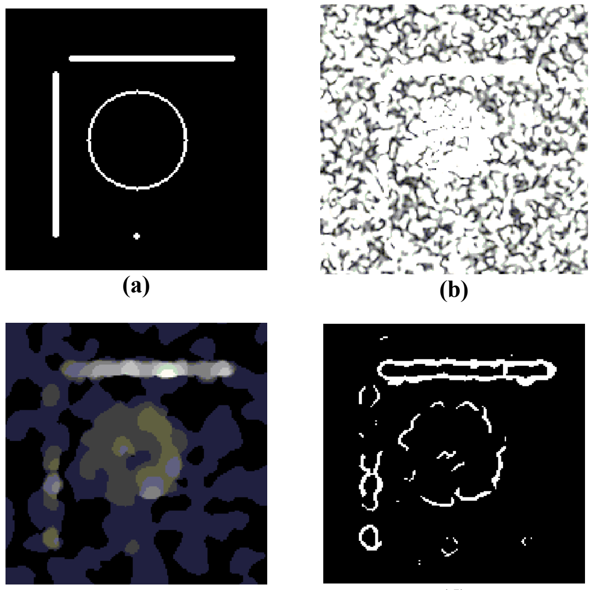


Figure 1: Ultrasound images: (a) Perfect edges (b) Noisy image (c) SRAD output (d) Detected edges from SRAD output

The initial HDG based on the Matlab code was supplemented with the component-level latency details listed in Table 2. The latency of the HDG is then lowered by duplicating hardware units to create parallelism and applying fidelity-preserving transformations. These transformations are standard logic synthesis optimizations, including dead code elimination, loop unrolling and term rewriting. The resulting HDG has 9 levels in the hierarchy and consists of a total of 950 nodes. Given this initial architecture, each pixel requires 48 clock cycles to finish one iteration of its update workload, and each pixel in the frame is processed serially. An improved Jacobian update removes inter-pixel dependency so that all pixels can be processed in parallel. The latency of processing one frame is 100 iterations/frame $\times$ 48 cycles/iteration = 4800 cycles/frame, or a throughput of 208K frames/second (fps) with a circuit running at 1GHz. Such a high throughput can rarely be achieved in reality due to the tremendous cost of building a circuit that supports such parallelism. While massive parallelism is expensive, we can still meet the real-time constraint by reducing the latency of processing each pixel. ColSpace can do that by applying fidelity-compromising methods.

Table 2: Principle component latencies [10]

Principle Component	Latency (unit = 2 clk cycles)
Addition/Subtraction	1
Multiplication/Squaring(^2)	1
Division/Inversion	10
Square Root	10
Shift	1
Memory Access	1

Table 3 contains relevant entries from the fidelity-compromising transformation library. Both entries are mathematical approximations with thresholds.

Table 3. Transformation library for SRAD case study

Transformation	(a)	(b)
Before	$\frac{1}{(1+\frac{1}{4}x)^2}$	$\frac{1}{1+y}$
After	$1-\frac{x}{2}$	$1-y$
p1 (threshold)	TH(x)	TH(y)
Condition	$\lvert x\rvert < TH(x), TH(x) \in (0,2)$	$\lvert y\rvert < TH(y), TH(y) \in (0,1)$
|Δlat|	11	10
|Δfid|	$\frac{3TH(x)^2}{16}+\frac{TH(x)^3}{32}$	$TH(y)^2$

Now consider how these two transformations are obtained. As discussed in Section IV, the ColSpace tool automatically finds the critical path and bottlenecks, which then become targets for additional fidelity-compromising transformations. For example, the bottleneck of the SRAD algorithm comes down to two DIVs and one SQRT. Then appropriate transformations can be easily developed to specifically attack the bottleneck operations. Transformations (a) and (b) have been developed to replace the two expensive DIVs. Both transformations are applied when x or y is smaller than their respective threshold. Since the values of x and y are not known until runtime, the transformations can be made conditional based on a pre-determined thresholds (TH(x) and TH(y)).

During actual runtime, a transformation can be either applied (Yes) or not applied (No) based on whether or not the threshold condition is met. Therefore there are a total of $2^2 = 4$ execution paths for these two candidate transformations. The latency evaluation routine considered each path that could be taken through the conditional transformations, as shown in Table 4. The percentage of time each of these eight paths was taken is threshold-depended and was determined by profiled executions.

Table 4. Critical path latencies for different transformation combinations

Case	A	B	C	D
Apply(a)	No	Yes	No	Yes
Apply(b)	No	No	Yes	Yes
Latency	4.8K	3.7K	3.8K	2.7K

Fidelity is evaluated in terms of *Pratt's figure of merit* [14], which is a value between 0 (worst) and 1 (best), generated by comparing the SRAD output with the output of a perfect edge detector. The original SRAD algorithm, free of any alterations, produces an average fidelity of 0.3678. Parameterizing the transformations using thresholds enables quantitative analysis. Figure 2 shows threshold exploration results for transformations (a) and (b) using three different cost functions: $\frac{LAT}{FID}$, $\frac{LAT}{\sqrt{FID}}$, and $\frac{LAT}{FID^2}$. The two axes denote discretized threshold settings, and each square denotes a certain combination of TH(x) and TH(y), with the two extremes being never apply transformation (modeled by a very small threshold of x and y) and always apply transformation (modeled by a larger thresholds). So the upper right corner corresponds to most frequently applying transformations while the lower left corner the least frequent. A lighter color signifies a higher cost function value, so darker colors are better design points. The derivative is calculated by taking the difference between adjacent squares. The plots shown are the average results from running ten different ultrasound images.

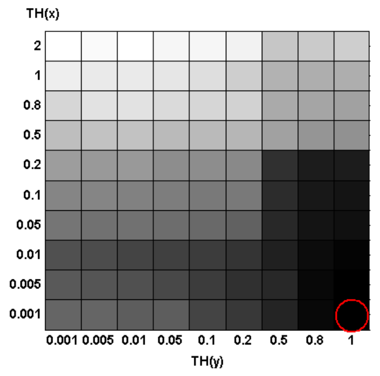
Since there is possible for a greedy algorithms exploring this space to converge to a local minimum on a 2D design curve, Table 5 compares results using different neighborhood sizes (i.e., number of squares away from the initial point). The neighborhood size determines the region within which $d(\frac{LAT}{FID})$ will be evaluated. The one with the lowest cost function in the neighborhood will become the design point in the next iteration. Larger neighborhoods are more likely to escapce local minima, but they require more processing time.

Table 5. Tradeoff between quality of result and exploration effort for different neighborhood sizes

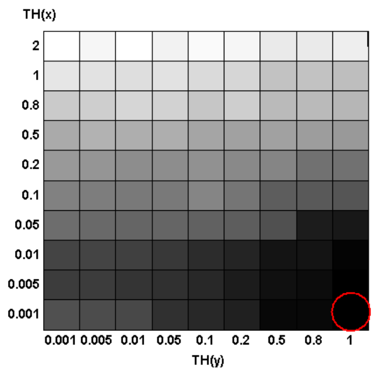
Neighborhood Size	Average Space Explored	Average Result Quality (compared to optimal point)
1	20%	98.15%
2	36.11%	100%
3	42.00%	100%
4	49.78%	100%

The tradeoff between the two transformations is more subtle. Observation shows that the optimal point can change based on the defined cost function and that the optimal points never apply transformation (a). This suggests that the system is more sensitive to transformation (a) than transformation (b). Thus, a good strategy is to apply transformation (a) conservatively (with a small threshold) and transformation (b) aggressively (with a larger threshold).

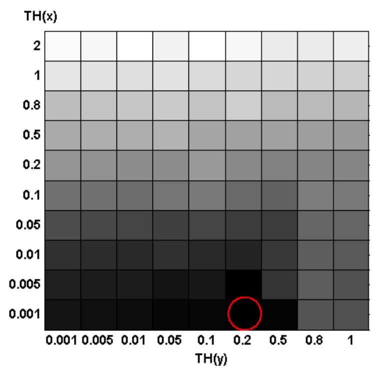
This cost-function-based approach can also be used to solve a constraint-based problem, where one metric is optimized while another is subject to a constraint (such as the real-time requirements of SRAD). The latency vs. fidelity and fidelity vs. latency Pareto curves in Figures 3 and 4 enable a designer to identify the design points where one metric is constrained and the other is minimized. These curves are derived from the mesh grid plot by searching the entire design space using the proposed algorithm and finding all points with equal fidelity (to construct Figure 3) or equal latency (to construct Figure 4). Then from this group of points, the one with lowest latency or highest fidelity is chosen.



(a)



(b)



(c)

Figure 2: 2D plots of various cost functions with two conditional transformation thresholds as variables. Representing cost function: (a) latency/fidelity$^{0.5}$; (b) latency/fidelity; (c) latency/fidelity^2. The red circle denotes the optimal design point that minimizes the cost function.

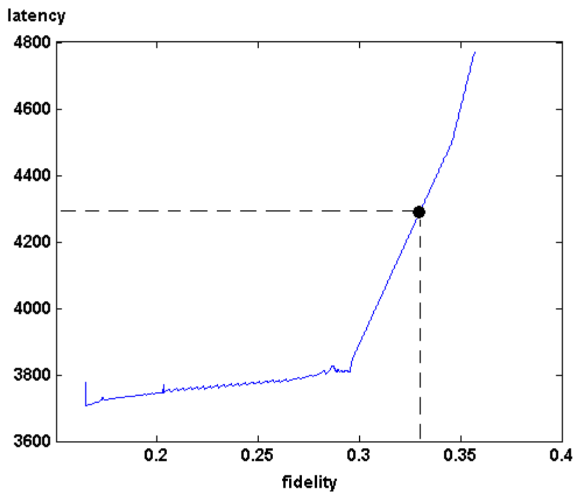


Figure 3: Latency vs. fidelity (with 90% fidelity point located)

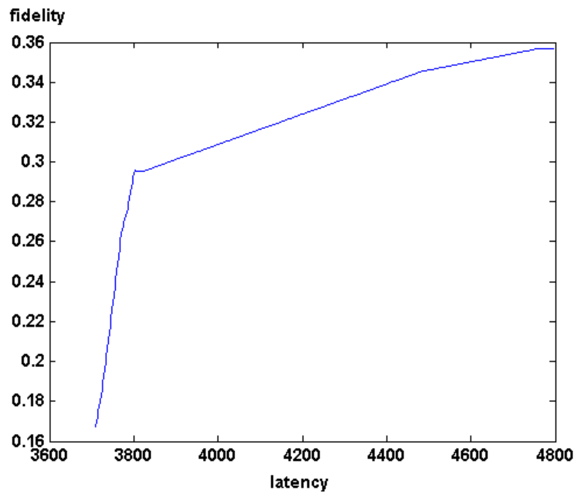


Figure 4: Fidelity vs. latency

Assume that our constraint is to allow no more than 10% degradation of fidelity, we will first calculate 90% best fidelity = 0.3310. We then look up the latency vs. fidelity plot to find the corresponding lowest latency, which is 4305, a 10.31% latency reduction. In addition to meeting metric constraints, these curves also allow designers to identify breaking points at which big savings in system metrics can be achieved. For example, fidelity=0.3 and latency=3800 are turning points where the slopes change abruptly.

Table 6 is a summary of improvements resulting from fidelity-compromising transformations that were applied (based on the optimal points selected in Figure 2). It is clear that the benefit of applying fidelity-compromising transformations is highly dependent on the type of cost functions. If the cost function puts greater weight on latency, our methodology can lower the cost function by a maximum of 13%. As the emphasis shifts to fidelity, the reduction on cost function that can be achieved is significantly less.

Table 6: Impact of transformations on system metrics (fidelity, latency, and cost function)

	Δfid	Δlat	Δcost
$\frac{latency}{\sqrt{fidelity}}$	-16.73%	-20.74%	-12.99%
$\frac{latency}{fidelity}$	-16.73%	-20.74%	-4.36%
$\frac{latency}{fidelity^2}$	+0.05%	-0.72%	-0.82%

B. *Snake Tracker*

A second case study was conducted on an object tracking algorithm called *snake tracker* [15]. It processes a sequence of 91 consecutive video frames and tracks the location of objects – in this case leukocytes captured *in vivo* with video microscopy. Since the detected location from the previous frame is used in determining the current location, all 91 frames need to be processed sequentially. The procedure of processing each frame is composed of two loops with some initialization and updating computation between and around the loops. The first loop prepares a gradient vector flow graph on which the second loop evolves a snake contour to determine the cell location. We adopted two different fidelity metrics proposed in [15] to evaluate the tracking quality.

Program profiling indicates that the first loop has an average latency 8 times that of the second loop, a clear bottleneck limiting the overall latency.

A closer inspection of the first loop reveals that it basically repeatedly updates a fixed-size matrix. Assuming the update amount in consecutive iterations is close, we can double the update amount to approximate the effect of iterating twice.

Replacing the loop body with doubled update amount and simultaneously cutting the number of iterations by half is a fidelity-compromising transformation that can reduce latency. Such a transformation can also be stored in a library and recognized for potential use by the automated design exploration algorithm.

As noted before, latency calculation involving an undefined number of iterations is simple with ColSpace. If the number of iterations is entered by the user, no change to the HDG is necessary. Prior to profiling, a counter is inserted in the loop body to gather the average runtime iteration count.

It is important to compare this fidelity-compromising transformation with simply reducing the number of iterations without altering the loop body. ColSpace allows easy evaluation of the two options by implementing both as candidate designs in the HDG and providing a uniform latency evaluation interface. Results are shown in Table 7.

Table 7: Comparison of fidelity-compromising transformation and simply reducing the number of iterations

	Simply Reduce # of Iterations			Fidelity-compromising Transformation		
Fidelity Metric	Δfid	Δlat	Δcost	Δfid	Δlat	Δcost
Percentage of Frames Tracked	-1.4%	-50%	-5.4 %	+4.2%	-50%	-17.6 %
RMSE (root mean square error)	-5.1%		-2.2 %	-0.6%		-13.2 %

Although both techniques reduce the latency by equal amount (50%), the fidelity penalty is considerably less when applying the fidelity-compromising transformation, thus providing a greater cost function reduction. This further reinforces our belief that fidelity-compromising transformations are a more effective class of techniques for latency reduction beyond the capability of traditional parameter tuning, such as cutting the number of iterations, limiting the number of objects processed or altering the numerical precision. And the proposed exploration algorithm together with the HDG formalism provides an efficient methodology to identify and evaluate such transformations.

VI. Conclusions and Future Work

ColSpace is a new design methodology (and accompanying tool) for algorithm-implementation co-optimization during the early design exploration stage. To achieve the co-optimization, it utilizes HDG as a representation of both the algorithm and the implementation, as opposed to existing methodologies with separate representations of each. The automated exploration algorithm identifies and evaluates fidelity-compromising transformations and can solve both constraint-based and cost-function-based design optimization problems. Two image processing case studies show significant cost function reductions are possible from the base design with the help of ColSpace. These case studies can be applied to most dataflow intensive algorithms. While an informal, manual approach to algorithm/implementation co-optimization could possibly achieve the same results, ColSpace provides a formal, automated methodology that could dramatically improve designer efficiency.

Currently, the tool can represent an HDG and evaluate latencies in the frontend GUI. The number of available transformations in the library (~10 in current version) limits the quality of exploration result. Future work on this methodology includes expanding the library and refining the search algorithm in order to handle bigger case studies, automating the alteration of the algorithm specification (e.g. the Matlab code) and incorporating additional implementation metrics (i.e. area and power) into the tool.

Acknowledgments

This work was supported in part by the National Science Foundation under grant No. IIS-0612049.

References

[1] S. Gupta, et al., "SPARK: A High-Level Synthesis Framework for Applying Parallelizing Compiler Transformations," *International Conference on VLSI Design*, 461-466, 2003

[2] K.S. Vallerio, N.K. Jha, "Task Graph Extraction for Embedded System Synthesis," *International Conference on VLSI Design*, 480-486, 2003

[3] G. Karakonstantis, et al., "Design Methodology to Trade Off Power, Output Quality and Error Resiliency: Application to Color Interpolation Filtering," *IEEE International Conference on Computer-Aided Design*, 199-204, 2007

[4] dot, www.graphviz.org

[5] JGraph v5.9.2.1, www.jgraph.com

[6] J. Skansholm, extra.jar, www.cs.chalmers.se/~skanshol/Java_eng/

[7] J. Schaeffer, "The Enterprise Model for Developing Distributed Applications," *IEEE Parallel & Distributed Technology: Systems & Technology*, 1(3):85-96, Aug. 1993

[8] G. Dong, N. Ray, S.T. Acton, "Intravital Leukocyte Detection Using the Gradient Inverse Coefficient of Variation," *IEEE Transactions on Medical Imaging*, 24:910-924, 2005

[9] W. Wu, S. Acton, J. Lach, "Real-time Processing of Ultrasound Images with Speckle Reducing Anisotropic Diffusion," *Asilomar Conference on Signals, Systems, and Computers*, 1485-1464, 2006

[10] P. Soderquist, M. Leeser, "Division and Square Root Choosing the Right Implementation," *IEEE Micro*, 17(4):56-66, Jul/Aug 1997

[11] T. Stefanov, et al., "System Design using Kahn Process Networks: The Compaan/Laura Approach," *Design, Automation and Test in Europe,* 340–345, 2004

[12] A. Ahmadi, M. Zwolinski, "Symbolic Noise Analysis Approach to Computational Hardware Optimization," *Design Automation Conference,* 391–396, 2008

[13] F. Balmas, "Displaying Dependence Graphs: a Hierarchical Approach," *Eighth Working Conference on Reverse Engineering*, 261–270, Oct. 2001

[14] A.J. Pinho, L.B. Almeida, "Figures of Merit for Quality Assessment of Binary Edge Maps," *International Conference on Image Processing,* 3:591–594, Sept. 1996

[15] J. Cui, S.T. Acton, Z. Lin, "A Monte Carlo Approach to Rolling Leukocyte Tracking in vivo," *Medical Image Analysis*, 10(4):598-610, Aug. 2006

[16] ColSpace, http://people.virginia.edu/~jh3wn/getit.html