Software-based Dynamic Overlays Require Fast, Fine-grained Partial Reconfiguration

Hossein Omidian
Xilinx Inc.
San Jose, CA, United States
hossein@xilinx.com

Guy G.F. Lemieux
University of British Columbia
Vancouver, BC, Canada
lemieux@ece.ubc.ca

ABSTRACT

In this paper, we consider *dynamic overlays* which use fine-grained partial reconfiguration (PR) to continuously adapt to their software-based workload. In particular, we show how to modify a traditional (static) overlay developed for OpenVX into a dynamic overlay. We use a Xilinx FPGA, and show that the dynamic overlay needs unsupported features including faster PR, relocatability, and fine-grained configuration is needed for performance. Since these features are not available in Xilinx FPGAs, we estimate the application-level speedup they would provide. We find that vector custom instruction (VCI) chaining, which allow a VCI to directly cascade its result into another VCI is also essential. Overall, we find the static overlay achieves a speedup of roughly 20× faster than a Cortex-A9 processor, but with improved PR and chaining a speedup of 106× is attainable. While there have been calls for fast, fine-grained PR devices for decades, we believe that dynamic overlays may be the first true "killer application" that will justify adding these features to all FPGA devices.

1 INTRODUCTION

Overlays add a new layer of configuration on top of the FPGA bitstream; customizing this new layer allows an overlay to solve a specific problem. Some overlays, such as [1, 4, 5, 7] also use partial reconfiguration (PR) as part of this new configuration layer, but they only perform PR once when the overlay is customized or the application is loaded, and do not do any further PR at run-time. For this reason, we call these *static overlays*.

In this work, we introduce the concept of *dynamic overlays*, where the overlay continuously changes a portion of the FPGA logic at run-time through PR. We are not presently aware of any dynamic overlays in the literature, although they may exist. In addition, we make a distinction between fine-grained and coarse-grained PR. This work assumes a fine-grained approach where a single large PR region can be filled with many smaller, relocatable PR modules; in this approach, configuration time is proportional to the size of the PR module's partial bitstream. In a coarse-grained approach, the entire PR region or subregion needs to be reconfigured — while it might be possible to do this without disturbing other PR module instances, the configuration time is proportional to the entire PR region or subregion because of the necessary bit scrubbing.

To demonstrate the value of dynamic overlays, we build one to accelerate OpenVX applications and show the potential speedup when support is provided for fine-grained, relocatable bitstreams with fast PR. OpenVX is a standard software API for computer vision defined by Khronos Group, who also defined OpenCL. We assume the application is written by a pure software programmer, and that it is fully portable to other vendors' OpenVX implementations.

Previous work uses HLS methods for synthesizing bitstreams from OpenCV [16] and OpenVX applications (e.g., OpenVINO [3], UC Irvine [11], and UBC [7] projects). However, all of these methods generate a completely new bitstream when the application changes by even a small amount. This makes it difficult to modify the application, since the programmer needs to use complex vendor place and route tools and a powerful PC. In contrast, the latest system from UBC [8] uses a static overlay, where partial configuration is used to partially customize the overlay at application load-time, but the bitstream remains static at run-time.

In this work, we extend our prior work [8] to investigate a new OpenVX system that uses a combination of VCI chaining and a dynamic overlay to achieve even greater acceleration. The original static overlay uses the VectorBlox MXP soft vector processor [10] (SVP) with static vector custom instructions (VCIs). The dynamic overlay allows VCIs to be reconfigured dynamically at run-time. In this work, a VCI is designed as a small, relocatable PR module within a large PR region (PRR) that can host multiple VCIs at once. Each VCI accelerates one type of compute kernel in an OpenVX application; as an application progresses from kernel to kernel, a different VCI needs to be used. Due to finite PRR capacity and internal fragmentation, some VCIs may need to be ejected to load a new VCI. The size of the PRR partial bitstream and the rate at which a VCI can be configured may affect application throughput. When a VCI is not available, the SVP can still be used to execute the kernel using regular vector instructions. As VCIs can vary considerably in complexity and size, the size of the bitstream for each PR module also varies. Hence, simpler VCIs can be configured faster, whereas more complex ones may take longer to configure but can also perform faster computation.

2 SYSTEM OVERVIEW

This study uses a ZedBoard. The overlay consists of an ARM Cortex-A9 host processor (667MHz), the VectorBlox MXP soft vector processor (SVP) [10] at 100MHz, and an empty PRR reserved for vector custom instructions (VCIs) [9]. We explore a range of PRRs up to 14,000 LUTs. To use a VCI, the associated partial bitstream must be relocated to an available location and then configured into the PRR.

HEART 2019, June 6–7, 2019, Nagasaki, Japan

ACM ISBN 978-1-4503-7255-8/19/06…$15.00
https://doi.org/10.1145/3337801.3337816

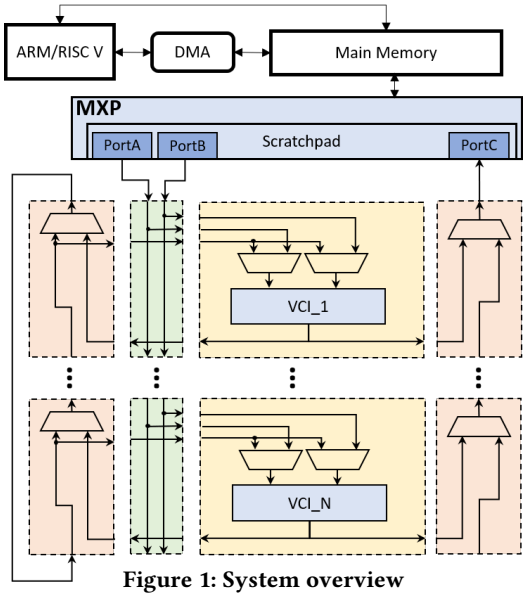

Figure 1: System overview

When used as a static overlay, the PRR can be filled with partial bitstreams for multiple VCIs until full. However, when used as a dynamic overlay, the contents of the PRR are modified on the fly at run-time, enabling the use of more VCIs than the PRR can hold at once.

Figure 1 provides a view of the overall system. The MXP has its own scratchpad and supports up to 16 different VCI opcodes. The scratchpad provides two streaming source operands on PortA and PortB, and accepts results on PortC. These ports are configurable in width for each VCI, and can provide or consume multiple words of data every cycle. Each VCI can be implemented within the PRR and connect to these ports. A mux network is implemented as part of each VCI and connects it with the MXP. This mux network can directly forward results between VCIs in a way that bypasses writing the intermediate result to the scratchpad. In this way, multiple VCIs can be chained together for higher performance, assuming they still conform to the 2-input, 1-output format.

In this system, there are three types of users: the Overlay Architect, the FPGA Architect, and the Application Programmer. The Overlay Architect designs the entire OpenVX run-time system and creates VCIs for the OpenVX kernels. Several instances of each kernel are created, each conforming to a different data width for the ports; wider implementations will achieve higher performance, but will also use more FPGA logic and require a longer time to reconfigure. When multiple VCIs are chained together, the data widths of their ports must match since there is no internal buffering. Any of these users may write OpenVX kernel code in C/C++ for the MXP, but it is most likely the Overlay Architect.

The FPGA Architect creates an instance of the overlay in an FPGA device, and allocates a certain amount of area to the PRR. The FPGA Architect also pre-synthesizes a library of VCI implementations into PR modules that will fit the unique column footprint and width of the PRR.[1] The FPGA Architect does not need to know about the application, and can do this work concurrently with application development.

The Overlay Architect implements OpenVX kernels in C++ using Vivado HLS to produce VCI implementations. The FPGA Architect uses these to produce multiple VCI implementations, one for each width from 1 to N lanes, where N is a power of 2 that matches the largest SVP to be used. This produces a minimal-area VCI for each width. Each VCI implementation is noted with its throughput, area requirement, and bitstream size. Each OpenVX kernel also has a pure software SVP implementation, where the throughput, in terms of CPU clock cycles per pixel, is recorded at each image tile size.

The Application Programmer simply follows the OpenVX API as defined by Khronos Group. The application is written in C, and can be debugged using any vendors' OpenVX implementation. Once it is debugged, it can be migrated to the FPGA system described here; the only portability concerns are minor details like memory capacity. Note the Application Programmer knows nothing about FPGAs, and does not use any FPGA vendor tools, only the OpenVX run-time system created by the Overlay Architect and the OpenVX library and FPGA device instance created by the FPGA Architect.

3 MAPPING OPENVX APPLICATIONS

An OpenVX application is defined as an OpenVX compute graph of kernel nodes that is built up dynamically by the C program. The OpenVX run-time system consists of two phases: an analysis phase that is allowed to optimize the graph, and an execution phase. During these phases, the run-time decides whether to run each kernel node either on the static overlay or a VCI, and whether VCIs will be loaded statically or dynamically.

Given the compute graph, and a target image size, the run-time system uses an execution time model to determine the best tile size, which kernel nodes should be implemented as a VCI, and which VCI implementation to use. It also determines whether to use bypassing/chaining, node fusion and dynamic PR. All of these steps are described below.

3.1 Finding Different Implementations

Consider an application described as a graph G with N nodes.

$$G = (V, E)\ ,\ V = \{f_1, f_2, ..., f_N\} \tag{1}$$

For each node f_m we can find N_{SVP} different software-based vector processor (SVP) implementations $S_m^1, S_m^2, ..., S_m^{N_{SVP}}$ as well as N_{VCI} different VCI hardware implementations $P_m^1, P_m^2, ..., P_m^{N_{VCI}}$. Each SVP implementation S_m^s can perform functionality of f_m on an image tile, in $t(S_m^s)$ time. Each VCI implementation P_m^s can perform functionality of f_m with area cost $A(P_m^s)$, and the number of pixels it can consume/produce $NP(P_m^s)$ each firing (i.e., tile width).

Using Execution Time Analysis (described below) and considering available resources, such as the size of the PRR, scratchpad capacity, and PR reconfiguration speed, the OpenVX run-time system decides which node should be implemented as SVP software and which one should be implemented as VCI hardware.

3.2 Execution Time Analysis

To execute a general OpenVX compute graph, the run-time system needs to fetch an image, one tile at a time, from main memory

[1]The footprint is defined by the order of the different types of columns, e.g. LUTs, DSPs, and BRAMs.

into the MXP scratchpad. For each tile, it must then execute the whole compute graph, one kernel node at a time (either as SVP or VCI implementations). In every stage of traversing the graph, the intermediate data is saved in the scratchpad. This means the tile size must be calculated based on the available scratchpad size, which will be influenced by the amount of intermediate data needed within each compute kernel, as well as the amount of state needed to hold intermediate results while traversing the entire graph. After all of the kernel nodes in the graph are executed on a tile, the results are written back to main memory before fetching the next tile. The execution time for these components is discussed below.

SVP Software Implementation

Assuming it takes $t_{DMA_{M2S}}$ to read from the memory to the scratchpad and $t_{DMA_{S2M}}$ to write from the scratchpad to the memory, the execution time of executing a compute graph G with N nodes $f_1, f_2, ..., f_N$ and subset of selected SVP implementations $S_1, S_2, ..., S_N$ on image tile T_j is t_{T_j}. This is:

$$t_{T_j} = t_{DMA_{M2S}} + [\sum_{i=1}^{N} t(Si)] + t_{DMA_{S2M}} \quad (2)$$

The overall execution time for N_T tiles in the image is:

$$t_A = \sum_{j=1}^{N_T} t_{T_j} \quad (3)$$

Accelerated VCI Implementation

Consider node f_m in the compute graph G. Instead of using SVP implementation S_m with execution time $t(S_m)$, it is possible to use a VCI hardware implementation P_m with execution time $t(P_m)$ and PR reconfiguration time t_{PR} to improve the execution time. Assuming we need to reconfigure this node N_{PR} times during the processing of the entire image, the execution time can be improved if:

$$N_T t(S_m) > N_{PR} \cdot t_{PR} + N_T \cdot t(P_m) \quad (4)$$

For OpenVX kernels implemented as VCIs, we can define kernel throughput $\Theta(P_m)$ as the number of pixels consumed/produced in each clock cycle. The same formulation can be used here to calculate VCI execution time $t(P_m)$:

$$t(P_m) = SetupTime(P_m) + \frac{TileSize}{\Theta(P_m) \cdot F_{max}} \quad (5)$$

where the F_{max} is the speed of the SVP (here, 100MHz). In addition:

$$t_{PR} = \frac{PR_{size}}{PR_{rate}} \quad (6)$$

Node Chaining

Each VCI normally implements one OpenVX kernel node, and only one VCI is executing at a time. However, if the graph topology allows, it's possible to find a cluster of nodes where a series of VCIs can chain together, sending the output of one directly to the input of the next, without writing intermediate results to the scratchpad. This scratchpad bypassing allows us to take advantage of pipeline parallelism by overlapping VCI execution to achieve higher performance.

Although chaining improves performance, it requires all VCI implementations in the chain to be active at the same time. That is, there must be sufficient area in the PRR to hold the entire VCI chain. In addition, the overall VCI chain still needs to follow VCI topology restrictions: overall, there can be a maximum 2 input operands (PortA and PortB) and one destination operand (PortC). Finally, every node in the chain must use the same data width so they are rate-matched.

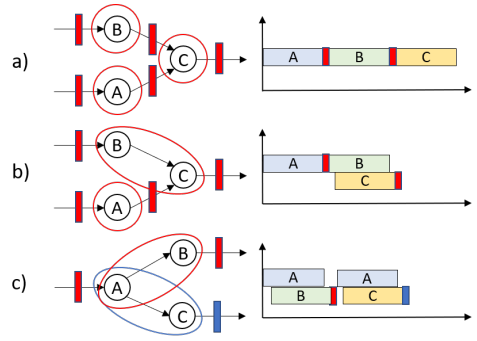

Figure 2: Node Clustering and Bypassing the Scratchpad

For example, Figure 2(a) shows a graph with three nodes and its execution timeline. Standalone VCI implementations are used for each node in the graph. This means each VCI implementation needs to write its results to the scratchpad before executing its successors. It also means each node must wait for its predecessors to finish their jobs before it can begin. Since the execution of nodes A, B and C do not overlap, the system only needs to keep one VCI configured at a time within the PRR.

In contrast, nodes B and C can be chained as shown in Figure 2(b). The chain bypasses the scratchpad for writing, so the result of node B is sent directly by the mux network to the VCI implementing node C. For this to work, the VCI for node A must be executed first, and the mux network must be configured to stream data through the VCIs of both B and C, which must be active simultaneously.

In a different example, shown in Figure 2(c), nodes B and C are both using the result of node A. This concept of fan-out was not present in the previous two examples. To avoid writing the result of A to the scratchpad, two VCI chains must be formed: A and B, as well as A and C. This example shows that clustering needs to consider all uses of the intermediate data between nodes.

Now that we have explained VCI chaining, we will describe the execution time analysis for standalone VCIs as well as VCI chains.

Pruning Slow Standalone VCIs

To enhance performance and reduce the search space, standalone VCI implementations that are slower than SVP implementations are pruned. Hence, we will only keep VCIs that satisfy the following equation:

$$t(S_m) > \frac{PR_{size} \cdot N_{PR}}{PR_{rate} \cdot N_T} + \frac{TileSize}{\Theta(P_m) \cdot F_{max}} + SetupTime(P_m) \quad (7)$$

Bypassing the Scratchpad

Similarly, we will prune VCI chains that are slower than the SVP implementations. Assuming we can implement a sequence of N_C nodes as a VCI chain, we will only keep VCI chains that satisfy the

Table 1: Some common patterns used for node fusion

Pattern
ConvertColor, Gaussian
Gaussian, Sobel_X
Gaussian, Sobel_Y
ConvertColor, Gaussian, Sobel_X
ConvertColor, Gaussian, Sobel_Y
Sobel_X, Sobel_Y, Magnitude
Sobel_X, Sobel_Y, Phase
Gaussian, Sobel_X, Sobel_Y, Magnitude
Gaussian, Sobel_X, Sobel_Y, Phase
ConvertColor, Gaussian, Sobel_X, Sobel_Y, Magnitude
ConvertColor, Gaussian, Sobel_X, Sobel_Y, Phase
Magnitude, Phase, Non-Maxima
Sobel_X, Sobel_Y, Magnitude, Phase, Non-Maxima

following equation:

$$\sum_{j=1}^{N_C} t(S_M^j) > \sum_{j=1}^{N_C} \frac{PR_{size}^j \cdot N_{PR}^j}{PR_{rate}^j \cdot N_T} + \max[\frac{1}{\Theta(P_m^j)}] \cdot \frac{TileSize}{F_{max}} + \sum_{j=1}^{N_C} SetupTime(P_m^j) \quad (8)$$

We prune the problem space by eliminating all implementations that cannot satisfy equations 7 and 8.

Pre-synthesized Node Fusion

Instead of VCI chaining, it is possible to fuse nodes together. This accomplishes a similar result, but the VCI must be pre-synthesized, so the pair of nodes to be fused must be known in advance by the Overlay Architect. This can also be automated by precomputing a library containing the most frequently-used pairs of OpenVX nodes.

The difference between VCI chaining and node fusion is shown in Figure 3. With chaining, the mux network is used to steer the output of *A* to the input of *B*. With node fusion, the connection is made internally within a single PR module, and the entire fused function is synthesized into a single VCI. This can yield higher performance within an area budget; for example, with node chaining, each VCI might be limited to 2 pixels per firing, whereas node fusion might be able to support 3 or 4 pixels per firing within a similar budget due to lower fragmentation.

It may be possible to fuse more than two nodes together. While producing all combinations of nodes would be infeasible, it is possible to consider only common patterns. For example, Table 1 lists a few common patterns with up to 5 nodes. These patterns must still conform to the overall two-input, one-output operand structure, but with node fusion it is possible encapsulate more complex internal structures, e.g. there may be more than two internal branches.

3.3 Solving the Space/Time Tradeoff

After pruning the problem space, the next step is exploring space/-time tradeoffs to find suitable implementations for each OpenVX node and solving the scheduling problem by finding which ones need to be implemented as SVP and which ones need to be implemented as a VCI or VCI chain.

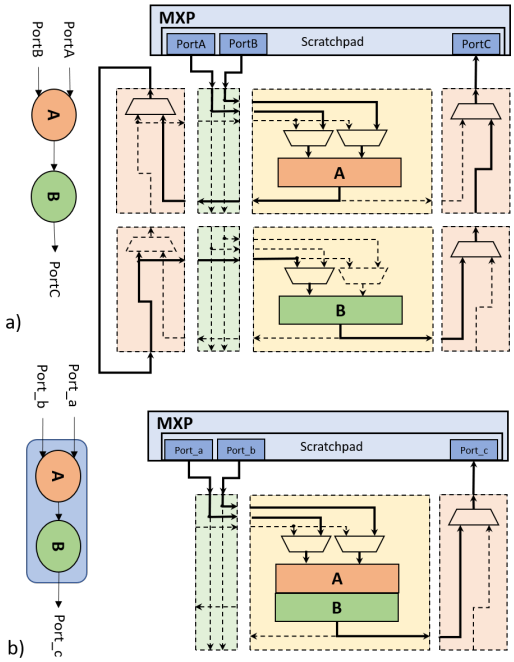

Figure 3: VCI chaining versus node fusion

Previous studies have shown the scheduling problem can be defined as an ILP problem and be solved by ILP solvers [6, 12]. Although ILP solvers can solve these problems, they lack flexibility such that it can be difficult to model certain types of constraints or opportunities such as node fusion.

Rather than address the complexities of ILP modelling, in this work, we developed a heuristic approach that allows us to consider combining or fusing multiple nodes into a single node. Going through different possible nodes to fuse, the heuristic uses an exhaustive search to find which nodes need to be implemented as SVP and which ones need to be implemented as VCI. The heuristic is simple and limited, but shows reasonable results. Due to space limitations, we are unable to describe it fully herein.

4 EXPERIMENTAL RESULTS

In this section, we investigate the estimated speedup provided by the SVP and various VCI configurations. The performance of three different hardware configurations are measured: the baseline ARM Cortex-A9 running at 667MHz; the SVP running at 100MHz together with the A9; and the SVP and A9 with different VCI options also at 100MHz. More than 25 OpenVX kernels are implemented as ARM, SVP and HLS versions. For the HLS versions, a library of different PR bitstream implementations for each VCI instance were generated to facilitate the trade-off finding process; this included determining the PR bitstream size and ensuring they could run at the target 100MHz. Note that we have fully implemented and verified all OpenVX kernel functions for the SVP and VCIs. However, due to time limits, we have not implemented the ability to load VCIs using PR. Also, Xilinx PR limitations do not allow us to vary PR rates, perform fine-grained relocation of PR modules, or load a

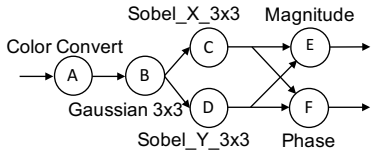

Figure 4: Graph representation of *Sobel* application

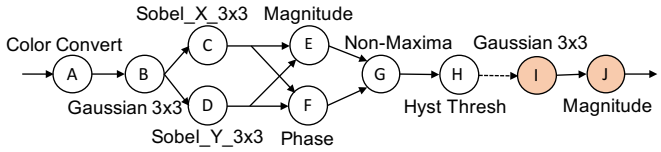

Figure 5: Graph representation of *Canny-blur* application

PR module based only upon its minimal bitstream size. Instead, we estimate speedup using VCI execution time profiles and estimated configuration times.

We generated two versions of the MXP configured with 4 lanes (SVP-V4) and 8 lanes (SVP-V8), where each lane is 32 bits wide. Using a ZedBoard, we measured actual SVP runtimes for the OpenVX kernels on just the ARM processor and on the ARM with MXP enabled. One experiment showed, on average across four different kernels (ColorConvert, Gaussian, Sobel and Magnitude), the SVP achieves a 4.6× speedup on SVP-V4 over the Cortex-A9, and an 8× speedup using SVP-V8. Neither of these results used VCIs, and they were all measured on real hardware.

To show the capabilities of our approach, we implemented two benchmarks as OpenVX compute graphs: 1) *Sobel* application with 6 nodes (Figure 4) and 2) *Canny-blur* application with 10 nodes (Figure 5).

All the kernels in both applications can be run using image tiles except the hysteresis thresholding kernel (node H) in *Canny-blur*. The hysteresis thresholding kernel needs the global image perspective, so it must DMA all tile results prior to that node before running the subsequent nodes. Thus, to run *Canny-blur*, we need to run the first part on whole image, save the results for the whole image to memory, and then read the results back for the second part.

4.1 Impact of Bypassing and Fusion

To show which kernels are actually selected as VCIs, and which are chained together, we refer to the speedups shown in Figure 6 on an SVP-V4 with various PRR area budgets. A breakdown of the VCIs selected for each area budget (numbered 1 through 5) are shown in Table 2. The SA rows of the table contain a list of which nodes in Figure 5 are implemented as *standalone* VCIs (i.e., without any bypassing); these results are similar to the static overlay without bypassing in prior work [8].

The first case of bypassing, BP1, chains together A and B, but is only slightly faster than SA1 which implements separate VCIs for A and B within the same area budget. As the area budget for the PRR increases, bypassing begins to show more improvement. Once the area budget is at 11,500 LUTs bypassing is over twice as fast as the standalone result (note the log scale on the Y-axis).

Table 2: List of kernels implemented as VCIs in Figure 6

Implementation	Kernels implemented as VCIs
SA1	A, B
BP1	AB
SA2	A, B, C
BP2	(AB), C
SA3	A, B, D
BP3	(AB), D
SA4	A, B, C, D
BP4	(ABC), (ABD)
SA5	A, B, C, D, E, F
BP5	(ABCDE) , (ABCDF)

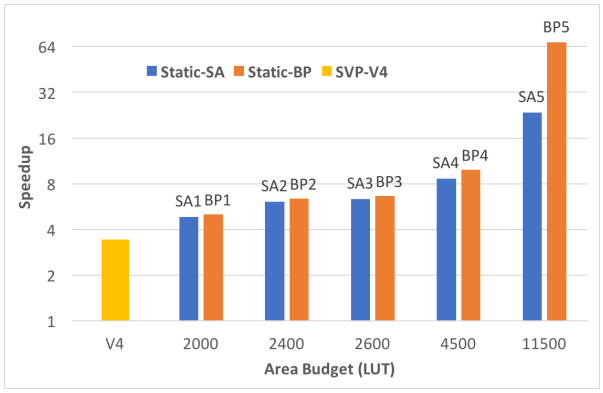

Figure 6: *Sobel* speedup by due to bypassing

4.2 Impact of Dynamic Overlay

So far, all of the VCIs are fully static, conforming to a static overlay. In this section, we will estimate the performance improvement with a dynamic overlay.

Now, instead of static VCIs within the PRR, we will dynamically reconfigure each VCI while evaluating a graph. To simplify discussion, suppose $N_{PR} = N_T$ in Equation 4. This might be the case when the first use of a VCI incurs latency, but future uses within a graph can hide the latency (e.g., configuration prefetching). In this case, the time to reconfigure and run on the VCI must also be faster than the software-only SVP implementation. That is,

$$t(S_m) > t_{PR} + t(P_m). \tag{9}$$

When we allocate a new VCI to the PRR, we need to select a precise location. First, we do a simple first-fit search strategy in the free space. If that fails, we eject the configured VCI that has been idle for the longest time. We find that most VCIs with the same bandwidth require a similar amount of space within the PRR.

This allocation heuristic is far from perfect, and it may lead to internal fragmentation within the PRR. Since the entire graph is known, a more precise scheduler can look at whether kernels are used multiple times in the graph. Also, since the multiple image tiles will be passed through the entire graph, the reconfiguration schedule is cyclic. Using these properties, a better heuristic can optimize for the fewest reconfigurations while also avoiding fragmentation.

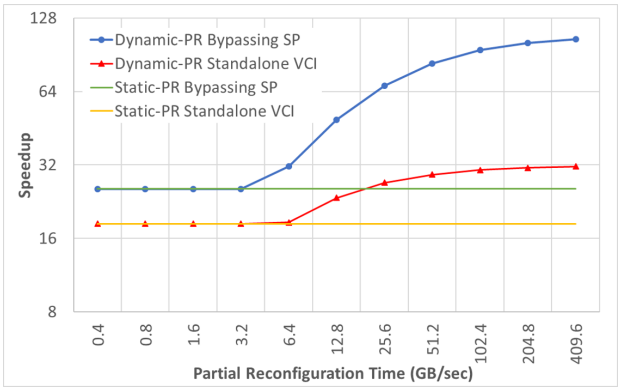

Figure 7: *Canny-blur* speedup due to dynamic overlay (14,000 LUT budget)

Rather than attempting to create such an optimal heuristic, we went with a simple one (which uses more reconfigurations than necessary, thereby underestimating the speedup) and ignored the fragmentation problem (since it is likely solvable). These should be addressed in future work.

Changing the PR Rate

When using a dynamic overlay, the PR rate determines how long it takes to configure a new VCI. Considering Equation 6 and 9, this rate can significantly influence run-time.

Figure 7 shows the impact that PR has on speedup. In this figure, the *Canny-blur* application is run on an SVP-V8 with both dynamic overlay and static overlay on an image of size 1920×1080. As the PR rate increases on the X-axis, the overall speedup (relative to ARM) improves. Although bypassing is initially a small advantage, its advantage grows as PR rate increases.

To put these results into context, the maximum PR rate for a Xilinx 7-series device is 400MByte/s using their on-chip ICAP interface.[2] This is the lowest PR rate shown in the figure, and it is so slow that the overlay cannot dynamically switch any VCIs. Hansen et al. demonstrated the ICAP can be overclocked over 5× to achieve 2.2GByte/s on real devices [2]. Unfortunately, the dynamic overlay provides no benefit until 6.4GB/s or higher. Xilinx's new Versal architecture [15] supports 16GB/s, which is a point where significant gains start to be seen. Assuming PR configuration data can be cached on-chip, even higher PR rates are possible. For example, Trimberger et al. proposed a time-multiplexed FPGA architecture in 1997 with a 33GByte/s reconfiguration rate [13]. We hope this work provides incentive for exploring much faster PR rates and fine-grained, relocatable PR in future FPGA devices.

5 CONCLUSION

This paper demonstrates the performance advantage of using a dynamic overlay over a static overlay. A dynamic overlay uses partial reconfiguration (PR) to dynamically change part of the overlay based upon the continuously changing needs exhibited by an application at run-time.

The dynamic overlay presented in this paper is used to accelerate OpenVX applications. It is based upon a soft vector processor (SVP) along with vector custom instructions (VCI) implemented as PR modules. When the set of VCIs is statically selected for an application, this creates a static overlay.

The OpenVX SVP-V8 static overlay with a 14,000 LUT area budget achieves about 17× speedup over the Cortex-A9. With node chaining and node fusion, this technique reaches about 28× speedup. A larger area budget allows encapsulation of even larger portions of the OpenVX graph, producing better results.

The OpenVX dynamic overlay allows VCIs to be reconfigured dynamically at run-time to maximize performance. With sufficiently fast PR rates, the overlay reaches 106× faster with bypassing, and 32× faster without bypassing. Hence, a combination of both fast PR and bypassing are needed to unlock maximum performance. This work requires a PR rate in excess of 50GB/s to achieve top performance, but the precise rate required is very system- and application-specific. Factors influencing this include DRAM speed, tile size, SVP scratchpad size, SVP width, and compute intensity of the OpenVX graph.

These speedup results are only enabled by a well-designed PR subsystem. The results in this paper depend upon very fast reconfiguration, relocating modules quickly with fine-grained positioning, and partial bitstreams that scale in size with the amount of area used (without requiring readback/scrubbing of larger regions or padding). This PR wishlist is not new, but OpenVX is an important application for FPGAs, and we hope that our dynamic overlay provides a compelling reason for vendors to reconsider implementing this wishlist – not only due to its high performance, but because it enables pure software programmers to use FPGAs.

[2]The PR rate for Intel FPGAs is similar [14].

REFERENCES

[1] Aklah, Z. T. *A Hybrid Partially Reconfigurable Overlay Supporting Just-In-Time Assembly of Custom Accelerators on FPGAs.* PhD thesis, 2017.
[2] Hansen, S. G., et al. High speed partial run-time reconfiguration using enhanced ICAP hard macro. In *IEEE IPDPSW* (2011).
[3] Intel. OpenVINO toolkit.
[4] Janssen, B., et al. A dynamic partial reconfigurable overlay framework for python. In *ARC* (2018), Springer.
[5] Koch, D., et al. An efficient FPGA overlay for portable custom instruction set extensions. In *FPL* (2013).
[6] Kooti, H., and Bozorgzadeh, E. Reconfiguration-aware task graph scheduling. In *IEEE EUC* (2015).
[7] Omidian, H., et al. Exploring automated space/time tradeoffs for OpenVX compute graphs. In *IEEE ICFPT* (2017).
[8] Omidian, H., et al. An accelerated OpenVX overlay for pure software programmers. In *IEEE ICFPT* (2018).
[9] Severance, A., et al. Soft vector processors with streaming pipelines. In *FPGA 2014.*
[10] Severance, A., et al. Embedded supercomputing in FPGAs with the VectorBlox MXP matrix processor. In *CODES+ISSS 2013* (2013).
[11] Taheri, S., et al. Acceleration framework for FPGA implementation of OpenVX graph pipelines. Tech. rep., Center for Embedded and Cyber-Physical Systems, UC Irvine, 2018.
[12] Tompkins, M. F. *Optimization techniques for task allocation and scheduling in distributed multi-agent operations.* PhD thesis, MIT, 2003.
[13] Trimberger, S., et al. A time-multiplexed fpga. In *FCCM 1997*, IEEE.
[14] Xiao, Z., et al. A partial reconfiguration controller for Altera Stratix V FPGAs. In *FPL* (2016).
[15] Xilinx. Versal: The first adaptive compute acceleration platform (ACAP).
[16] Xilinx. *Xilinx OpenCV User Guide UG1233*, January 2019.