Algorithmic and memory optimizations on multiple application mapping onto FPGAs

Harry Sidiropoulos, Ioannis Koutras, Dimitrios Soudris
School of Electrical and
Computer Engineering
National Technical University of Athens, Greece
Email: [harry,joko,dsoudris]@microlab.ntua.gr

Kostas Siozios
Department of Physics
Aristotle University of Thessaloniki, Greece
Email: ksiop@physics.auth.gr

***Abstract*—Field Programmable Gate Arrays (FPGAs) offer a low power flexible accelerator alternative due to their inherent parallelism. Reprogrammability, although its their key feature, it is used almost exclusively on design time due to the constrains imposed by the modern CAD tools that require even days to run and tens of GB of RAM. In order to effectively utilize FPGAs on run time we propose a novel methodology and the supporting toolflow that enable efficient mapping of multiple applications onto heterogeneous FPGAs. With the use of a floorplanning step, memory optimizations and custom memory allocators, we alleviate the constrains imposed by CAD tools, and provide a proof of concept that application mapping onto FPGAs can be done on run time. Experimental results prove the efficiency of the introduced solution, as we achieve application's mapping 40× faster on average compared to a state-of-art approach, without performance degradation and with 12× on average reduced memory usage.**

I. Introduction

Existing applications impose a continuously increasing demand for processing power. This trend affects not only scientific and industrial applications, but also consumer and end user applications. As an outcome, a number of design strategies and methodologies have been proposed that take into advantage the additional flexibility offered by heterogeneous systems. These are usually consisting of general-purpose CPUs and hardware accelerators. They span from thousands-core data centres and cloud hardware infrastructures to small embedded systems like cellphones, smart TVs, watches and cameras.

In order to efficiently integrate on this new landscape, FPGAs need to support fast application development and implementation. Industry has taken steps towards faster application development, exploring diverse solutions. Examples of these solutions can be found in the EDA tools of leading commercial FPGA companies, like Xilinx that has integrated a High Level Synthesis environment in the toolflow Vivado [1], and Intel-Altera that supports OpenCL kernels [2]. For faster application implementation the main body of research focuses on faster mapping algorithms and tools.

This research work was partially funded by VINEYARD H2020 European project.

Another approach for supporting fast application implementation relies on reconfiguring parts of the FPGA on runtime in order to change an already implemented design or replace it with another. This technique has been also used in commercial FPGA platforms. Research on this topic aims usually to identify a proper region over the target architecture, with a sufficient amount of contiguous free hardware resources. Additional problems are resource fragmentation and the challenge of coherent and transparent pausing and continuing of the applications.

As mentioned before there is a need for a dynamic environment that supports and accelerates multiple applications. In current commercial ([1], [3]) and academic tool flows (VTR [4]), there is no direct support for mapping multiple independent design onto an FPGA. More specifically in the academic VTR project, which is a mature widely accepted FPGA toolflow, the only way to map multiple applications is to merge their HDL description and execute the toolflow from the beginning. This is inefficient, counter-intuitive and in a real-world scenario this could only have been done once, in the design time.

An important but often neglected part of the P&R steps is the memory footprint of the tools. Since modern designs and FPGAs are reaching the range of million logic cells the main bottleneck for the execution time of the tools is the memory usage and not the algorithms used in each step. As an example Table I shows the memory footprint of the VPR 7.0 tool (the P&R tool of the VTR project) when mapping designs onto a medium sized, midrange FPGA consisting of 30,000 Logic Blocks. Its clear that memory usage can easily become a bottleneck for execution easily overshadowing algorithmic improvements on the tools. This is specially true if we consider the scenario, when at runtime we need to map multiple applications in parallel onto a single FPGA.

II. Related Work

The most computational intensive task during application implementation onto an FPGA, is the placement and routing (P&R) step. In order to overcome this limitation researchers have already proposed a number of solutions [5], [6], [7]. Authors in [5] have developed a parallel placer based on a simulated annealing algorithm in order to decrease execution

TABLE I
MEMORY FOOTPRINT OF VPR 7.0 TOOL WHEN MAPPING DESIGNS ONTO A MEDIUM SIZED FPGA.

Benchmark	Memory	Benchmark	Memory
arm_core	5.611 GB	mkDelay-Worker32B	5.381 GB
bgm	6.019 GB	mkPktMerge	5.237 GB
blob_merge	5.368 GB	mkSMAdapter4B	5.251 GB
boundtop	5.281 GB	or1200	5.291 GB
ch_intrinsics	5.214 GB	raygentop	5.258 GB
diffeq1	5.225 GB	sha	5.270GB
diffeq2	5.214 GB	stereovision0	5.484 GB
LU8PEEng	5.907 GB	stereovision1	5.498 GB
stereovision2	6.006 GB		

time and incorporated this placer in Altera's FPGA toolflow. In [6] and [7], authors incorporate known techniques from the Application Specific Integrated System's (ASICs) domain in order to reduce the placer's execution time.

The main body of research concerning the execution time of the application mapping is focused on the algorithmic side, mainly of the placement step (since it is time consuming, and greatly affects the quality of mapping) [8], [9], [10] and on the algorithmic side of the routing step [11], [12]. Towards fast application implementation in [13], we introduced the idea of a VKernel, that acted as a wrapper in order to accelerate application mapping and accommodate dynamic mapping of multiple applications onto an FPGA.

The execution time and computational resources when mapping an application onto an FPGA becomes a serious headache for designers as the designs continually increase in size. An important work that highlights this problems is [14], where the authors explore the gap between academic and commercial tools. More specifically in this work multiple large benchmark were used to test the execution time, memory consumption and quality of solution between VTR [4] and Altera's Quartus[3]. The results show that VTR on average needed almost 4 hours, in some cases exceeding the 48 hours limit, and 22GB on average memory. Altera's Quartus[3] was significantly better, but still needed 2 hours and 4GB of RAM on average.

Managing the memory for such intensive tasks has been studied for a long time. Most present operating systems also take some of the work from the dynamic memory allocators. Sophisticated virtual memory designs, implemented in the kernel space, make many memory decisions on the user space easier and faster. In these environments contemporary, third-party memory allocators cannot consistently beat the system allocators [15]. Requiring such large amounts of memory from modern computer systems is trivial, but the systems are not optimized for this specific type of workload by default.

Low-level dynamic memory allocators adhere typically to a standard group of functions defined in the C standard library, namely malloc, realloc, calloc and free, but their implementations vary a lot. Dlmalloc [16] has been the reference point for most of them, organizing memory in bins of arbitrary sizes and putting data accordingly. This idea has been the base for many different allocators including ptmalloc, Hoard [17] and jemalloc [18]. Ptmalloc continues the concept of bins for different sizes, but extends it in the field of multi-threading applications. Hoard introduces the concept of memory blow-up, where an application cannot properly return memory to the system although it is not actively used by it. Finally, jemalloc shares a similar strategy in handling multiple heaps, called arenas, but also maintains several thread caches in order to reduce the volume of synchronization events in multi-threaded applications.

All these memory allocators are general purpose ones and as such, they misuse the memory space when an application is employed that has heavy memory requirements, but with a specific memory pattern.

III. CONTRIBUTION

Throughout this research work we introduce a novel methodology and the supporting tool-flow for performing dynamic mapping of multiple applications onto an FPGA. We consider a heterogeneous FPGA platform as a pool of hardware resources including logic blocks, memory and DSP blocks, where applications can be mapped as dynamic kernels onto these resources. In order to make it feasible to map these kernels at runtime we have significantly reduce both the execution time and the memory footprint, by combining aggressive memory optimizations with custom dynamic memory allocators.

The key features of the proposed methodology, named Het-JITPR, and the supporting toolflow are summarized as follows:

- Mapping of multiple application kernels onto a single modern FPGA architecture consisting of adaptive logic blocks, non uniform routing, DSP and RAM blocks.
- The processing requirements have been greatly reduced through both algorithmic and memory optimizations.
- Mapping of those dynamic kernels is performed onto the FPGA, through aggressive P&R, even under runtime constrains.

We extended the idea presented in [13] by further reducing computational resources and execution time. We investigated thoroughly the memory consumption, memory usage and execution time bottlenecks and developed optimizations dealing with those bottlenecks. More specifically:

- We investigated the memory usage throughout the entire tool flow and identified the most memory hungry operations.
- In the floorplanning step of our tool-flow, we have developed through experimentation a complex heuristic that takes into account multiple mapping variables
- We introduced optimizations in order to reduce the memory bottlenecks and experimented with custom memory allocators to further optimize memory usage.

The remainder of this article is organized as follows. Section 2 presents the related work. Section 3 gives an overview of our methodology for efficient kernel mapping on FPGAs and the details of the optimizations in the introduced supporting framework. Experimental results are discussed in Section 4. Finally, conclusions are summarized in Section 5.

IV. Proposed Solution

A. Overview of the proposed methodology

The introduced methodology is software supported by a tool-flow, named Het-JITPR. Whenever a new application has to be mapped onto the target architecture, a generic version of its netlist is fed as input. The first step is performing a floorplanning step targeting to determine the most suitable region over the FPGA, where the application will be implemented, with the introduced VKernel-Planner tool.

The application's netlist is then placed and routed (P&R) strictly onto the resources allocated for it. In both these steps the tools are unaware of the rest FPGA fabric, outside the selected region.

The proposed placer is based on a fast simulated annealing algorithm, based on the VTR toolflow's placer algorithm [4], that supports also hard blocks. The placer's execution time has been significantly improved by reducing the number of moves compared to the original VPR algorithm as previously researched in [19]. Furthermore since we place the application onto a region that is tailored specifically for this application's requirements, we significantly decrease the solution space of the placement, thus achieving even faster execution times.

Apart from the placer, our framework incorporates a router that identifies the routing segments and switches that should be used in order to create connected paths from net sources to net destinations for all the networks in a circuit. The routing algorithm is based on the PathFinder negotiated congestion algorithm [4], but it has appropriately tuned for faster execution times. This speedup of the execution time is achieved by setting a very high cost value for initial channel overuse.

After successful P&R the VKernel's occupied resources are directly linked to corresponding FPGA resources. At this step, all the necessary information for computing the partial bitstream file for the new application is available. Whenever an application has to be deallocated from the reconfigurable device, we appropriately update the information about its resources (we mark them as non-utilized).

B. Floorplanning optimizations - VKernel-Planner

The VKernel-Planner tool initially, assigns a number of uniformly distributed seeds across the FPGA. Each seed represents a potential area for the new application kernel. The number of seeds, the seeds' size, as well as the distance between two consecutive seeds, are tuned at runtime, since their selection is affected by the availability of hardware resources and the performance requirements.

Each of the seeds is expanded towards x and y directions repeatedly, with a fast variation of the flood-fill algorithm until the seeds include sufficient amount of hardware resources for the application's requirements. The areas represented by each seed, can be overlapping with each other, but they cannot overlap with the area assigned to another previous application.

The cost of each seed's solution evaluated by the VKernel-Planner has three components, C_α, C_β and C_γ, associated with the efficiency of resource allocation.

- The overuse of resources is represented in component C_α and it affects the total number of applications that can be instantiated onto the target FPGA.
- C_β component represents how close the applications are mapped onto the FPGA, and affects the fragmentation of reconfigurable resources.
- The regularity and the squareness of the candidate region are represented in component C_γ and they affect the quality of the application's mapping.

Its important to state that the term $Bounding\ Box$, included in most of the following equations is the minimum rectangle containing a given set of blocks. It is defined as $[x_{min}, y_{min}] - [x_{max}, y_{max}]$ of the coordinates of each block in the given set.

Each cost component is calculated by the following equations:

1) Component α (Eq.1):

$$C_\alpha = \sum_{i=1}^{n} k_i * \frac{R_{req\ i}}{R_{free\ i}} \qquad with \sum_{i=1}^{n} k_i = 1 \tag{1}$$

$R_{req\ i}$: *the number of type i blocks required by the kernel*
$R_{free\ i}$: *the number of free type i blocks inside the seed's area*
k_i : *a weight factor representing how critical is resource type i*

Each candidate region might include more resources than those actually required by the VKernel-Planner. In an FPGA there are several resource types i.e. Logic blocks, I/O pads, DSP blocks, etc., each one with different number and physical position inside the reconfigurable fabric. With the C_α we penalize the amount of extra resources, through the $R_{req\ i}/R_{free\ i}$ term and k_i is a normalized weight depending on the type of resources i. Extra DSP and memory blocks are penalized with the highest k factor, lower penalty factor have the unutilized I/O blocks and the lowest the logic blocks. In order to better illustrate this cost, Figure 1(a), shows an example of two candidate regions for the application kernel where $C_{\alpha\ I} > C_{\alpha\ II}$.

2) Component β (Eq.2):

$$C_\beta = 1 - \frac{Free\ blocks\ at\ Utilized\ Bounding\ Box}{Total\ blocks\ at\ Utilized\ Bounding\ Box} \tag{2}$$

By Utilized Bounding Box we denote the Bounding Box that contains both the candidate region for the new application kernel and every already deployed kernels. C_β represents how close the new application kernel will be deployed to the previous kernels. This affects the quality of mapping for future designs since it introduces high irregularity on future kernel's shape. In order to better illustrate this cost, Figure 1(b), shows an example of two candidate regions with their respective Utilized Bounding Boxes where $C_{\beta\ I} > C_{\beta\ II}$ and the grey area is reserved from already deployed kernels.

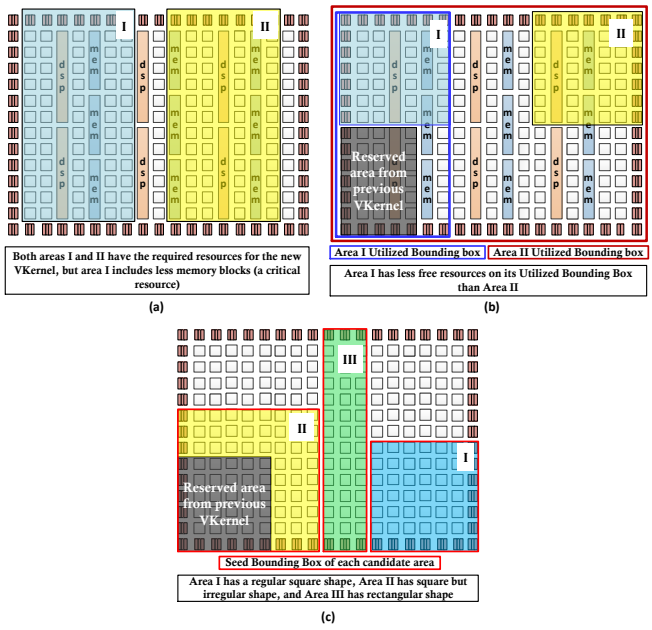

Fig. 1. Example of the three cost components used to evaluate solutions on VKernel-Planner. (a)C_α Eq 1 where $C_{\alpha\ I} > C_{\alpha\ II}$, (b) C_β Eq 2 where $C_{\beta\ I} > C_{\beta\ II}$, (c) C_γ Eq 3 where $C_{\gamma\ I} > C_{\gamma\ II}$ and $C_{\gamma\ I} > C_{\gamma\ III}$.

3) Component γ (Eq.3):

$$C_\gamma = \left(1 - \frac{|nx - ny|}{nx + ny}\right) + \left(\frac{Free\ blocks\ at\ SBB}{Total\ blocks\ at\ SBB}\right) \quad (3)$$

nx : *the width of the seed's area*

ny : *the height of the seed's area*

SBB : *Seed Bounding Box*

By Seed Bounding Box we denote the Bounding Box that contains the blocks of the seed's area. C_γ represents the squareness $(1 - |nx - ny|/(nx + ny))$ and the regularity $(\frac{Free\ blocks\ at\ Seed\ Bounding\ Box}{Total\ blocks\ at\ Seed\ Bounding\ Box})$ of the seed's area. High irregular shapes from our experience affect in a negative way the quality of P&R of an application, thus we prefer the region reserved for a kernel to be regular. This concept is illustrated in figure 1(c) where $C_{\gamma\ I} > C_{\gamma\ II}$, $C_{\gamma\ I} > C_{\gamma\ III}$ and the grey area is reserved from already deployed kernels.

After calculating C_α, C_β and C_γ for each seed, VKernel-Planner keeps the solutions (seeds) that form the pareto front in the three dimensional space created by these cost components. From the pool of these seeds VKernel-Planner selects one depending which aspect of cost we want to optimize, by user defined factors.

The runtime overhead of the VKernel-Planner is negligible since its complexity is $O(n \times \sqrt{\frac{n}{m}})$, where n denotes the number of slices found to the target architecture and m is the number of slices required for application implementation.

C. Memory Optimizations

1) Application-level optimizations : The user input to our toolflow of the application that we map onto the FPGA is a .net, netlist file. This file represents hierarchically the input design expressed in the complex blocks that our FPGA architecture consists of, Logic Blocks, I/O pads, Memory and DSP units. A Logic Block in the .net file will contain the name of the block, all the input and output connections, and the internal architecture concerning how the LUTs, the latches, the flip-flops, etc. are interconnected in the block. Naturally this corresponds to the architecture of the target FPGA.

In profiling of VPR 7.0 (the P&R tool of the VTR framework [4]) we found out that a significant amount of memory is spent on the internal representation of these complex blocks. An independent routing graph describes this interconnection in each block. The memory usage of each benchmark is shown in Table II. The variation in memory are analogous to the benchmark size.

TABLE II
MEMORY USED FOR THE REPRESENTATION OF EACH BENCHMARK'S NETLIST INSIDE VPR 7.0.

Benchmark	Memory	Benchmark	Memory
arm_core	228.67 MB	mkDelay-Worker32B	114.49 MB
bgm	491.86 MB	mkPktMerge	5.237 GB
blob_merge	94.59 MB	mkSMAdapter4B	19.42 MB
boundtop	41.82 MB	or1200	47.61 MB
ch_intrinsics	9.23 MB	raygentop	32.68 MB
diffeq1	10.61 MB	sha	36.68 MB
diffeq2	8.75 MB	stereovision0	158.44 MB
LU8PEEng	393.10 MB	stereovision1	166.87 MB
stereovision2	464.18 MB		

When a complex block is read from the netlist its internal interconnection is processed and the corresponding routing graph is created. This graph is persistent throughout the execution of the P&R. In order to reduce the graphs footprint we save in a global vector only the information about the pins of the block, their connections, if they are unconnected, or their equivalence, but not the internal connections. Then we free the graph before reading the next complex block. With this modification we achieve two things:

1) The memory used by the vector is much less than this used by the graph since we don't include the internal connections
2) With the use of a vector we achieve better memory locality than with the graph.

Although there is an effective reduction in the memory footprint this solution has the drawback that we can't change the internal mapping of the blocks. This option is used sometimes by slow placement algorithms that use multiple optimization phases in an effort to further optimize the quality of mapping. Since the default VPR placer and our placer are not changing the internal mapping of the blocks, disabling this option has no effect in the quality of the results.

2) Architecture-level optimizations : During the routing phase the blocks of the design are interconnected through the resources of the FPGA. For this reason independent from the routing algorithm that will be used, the EDA tools

need the representation of all possible interconnection paths. In VPR 7.0 a large percentage of execution time is spent building the architecture's rr graph, a routing resource graph (rr graph) that has as nodes every pin and every track of the FPGA routing infrastructure. This percentage has been reduced significantly by using the proposed VKernel-Planner tool, due to appropriate selection of a subarea for the application kernel.

Each switchbox in an reconfigurable architecture connects the horizontal and vertical channels at their junction. Depending on the switchbox architecture each wire from each side (top, bottom , left, right of the switchbox) can be connected up to $3 \times N$ wires where N is the channel width. This becomes more complicated when we have non-uniform routing architecture, where tracks can have variable lengths. For example in an FPGA with array size 200×200 (modern medium-size FPGA) and channel width of 200 the memory size of this array is 1,775.4 MB or 1.73 GB.

In order to reduce the memory footprint we changed the allocation of this array from static to dynamic. The information is needed every time a horizontal or vertical channel is build (in the rr graph) at the position x,y. In order to minimize the memory allocation we allocate and calculate the array at each x column and then deallocated it. This gives us the best trade-off between execution time and memory footprint. If the allocation/deallocation would be done in finer grain (in x,y point and not column) we found out that there was a significant increase in execution time.

3) Universal Memory Optimizations : In order to accurately evaluate placement solution, VPR before placement constructs an array that correlates routing delays with blocks' physical distance. For this reason the router builds the rr graph of the FPGA and routes blocks that have different distances between them. We have replaced this step with an encoded bit file that contains this information. The size of this file is a few KBytes even for large FPGAs. Our proposed placer reads that file and creates the array without the need for rr graph and routing. Lastly minor memory footprint reduction has been achieved by restructuring and changing the internal types in various graph structures.

4) Proposed Dynamic Memory Allocators: Even if it is a CPU and memory intensive task, we have found out that dynamic application mapping to an FPGA has static requirements in terms of memory usage and a solid memory usage pattern. As such, we propose to deploy a simpler dynamic memory allocator to reduce the execution path of memory allocation operations, improving the execution times and/ or the memory footprint.

While profiling the VPR application, we have noticed that the memory request sizes were very specific and it is safe to assume that the application data structures use those specific, fixed sizes. Those sizes can be used to tweak the allocator's internal data structure organization in order to host memory requests of them in a faster and more efficient way.

We have compiled our solution against two modern dynamic memory allocators, jemalloc [18] and Lockless [20]. Jemalloc is the default allocator in FreeBSD and Firefox, while Lockless is an allocator using many optimization tricks that modern systems support. When the application frees memory portions, the chosen memory allocators are more reluctant to return it to the system than the default allocator in most Linux systems, glibc's allocator. If the system runs out of memory, the virtual memory of the Linux kernel is smart enough to put to the swap area the memory addresses of processes which are currently not running.

In jemalloc each thread maintains a cache of objects and during an allocation request and before the thread accesses the memory regions, it checks first for a cached available object. Allocation via a thread cache requires no locking whatsoever in contrast with the possibility of using multiple locks while searching over different memory regions. As we have noticed, some small sizes are dominantly popular and thus we may tweak this configuration to maximize the allocation speed. At any case, thread caches should offer speed without effecting the memory fragmentation.

Lockless takes a different approach to the management of small sizes by using a slab allocator. Slabs are memory regions of 64 KiB plus 128 bytes for keeping the necessary metadata. There are many slabs, one for each size up to 512 bytes increasing in step of 16 bytes. One possible drawback compared to jemalloc's implementation is that slabs are traded between parallel threads and might be subject to synchronization overhead. However, both Het-JITPR and original VPR are using a single thread, so no particular slowdown is expected by using the slab allocator.

V. Experimental results

This section provides a number of experimental results and comparisons that highlight the efficiency of the proposed solution. For this purpose, we employ 17 common hardware designs that some make use of multipliers and memories. Table III summarizes the characteristics of these benchmarks. The implementation medium for our experimentation is an FPGA platform consisted of an array of 200×200 logic blocks, as well as a number of embedded multipliers and memories, similar the Altera Stratix IV architecture.

A. Quality of mapping a single design

Next, we quantify the performance of the derived solution in terms of maximum operation frequency. The results of this analysis for the two alternative flows are summarized in Table IV. Based on this table, our framework has a gain in terms of maximum operation frequency compared to the VPR, $1.26\times$ on average, because we significantly reduced the solution space of placement. The placer is using a simulated annealing algorithm where we significantly decreased the number of random moves compared to VPR [4] tool. The FPGA architecture used consists of roughly 40,000 blocks of every type. With the proposed toolflow that number is tailored specifically to each application. So lets take three examples from Table IV:

TABLE III
CHARACTERISTICS OF THE EMPLOYED BENCHMARKS.

Benchmark	LUTs	Inputs	Outputs	Mem	Mult
arm_core	13697	133	179	40	0
bgm	30782	257	32	0	11
blob_merge	6018	36	100	0	0
boundtop	3037	275	192	1	0
ch_intrinsics	425	99	130	1	0
diffeq1	485	162	96	0	5
diffeq2	322	66	96	0	5
LU8PEEng	21739	114	102	45	8
mkDelayWorker32B	5631	511	553	43	0
mkPktMerge	228	311	156	15	0
mkSMAdapter4B	1977	195	205	5	0
or1200	3053	385	394	2	1
raygentop	2147	239	305	1	7
sha	2277	38	36	0	0
stereovision0	11472	157	197	0	0
stereovision1	10287	133	145	0	38
stereovision2	29768	149	182	0	213

– **ch_intrinsics**: In VPR [4], placement evaluates in total 225,986 solutions while each block has 40,000 possible positions. Our placer evaluates 33,782 solutions but each block has 465 possible positions.
– **mkPktMerge**: In VPR [4], placement evaluates in total 523,852 solutions while each block has 40,000 possible positions. Our placer evaluates 64,610 solutions but each block has 1,180 possible positions.
– **LU8PEEng**: In VPR [4], placement evaluates in total 3,908,545 solutions while each block has 40,000 possible positions. Our placer evaluates 298,480 solutions but each block has 3,080 possible positions.

This also explains why in the largest benchmarks, we see a slight decrease in the Max operating frequency. As the design reaches in size the size of the FPGA the gains offered by the VKernel-Planner step are diminished.

TABLE IV
COMPARISON BETWEEN THE HET-JITPR AND THE VPR TOOLFLOW, IN TERMS OF MAXIMUM OPERATION FREQUENCY.

Benchmark	Max Op. Freq (MHz)		Gain
	VPR [4]	Het-JITPR	
arm_core	52.716	53.215	1.009×
bgm	39.938	36.514	0.914×
blob_merge	90.913	89.820	0.988×
boundtop	124.247	146.075	1.176×
ch_intrinsics	118.986	256.546	2.156×
diffeq1	40.905	45.932	1.123×
diffeq2	50.841	57.959	1.140×
LU8PEEng	8.5423	8.4703	0.992×
mkDelayWorker32B	95.567	128.85	1.348×
mkPktMerge	141.55	220.384	1.557×
mkSMAdapter4B	113.544	162.929	1.435×
or1200	59.423	66.332	1.116×
raygentop	107.758	192.735	1.789×
sha	70.7102	72.062	1.019×
stereovision0	172.764	218.32	1.264×
stereovision1	99.393	154.015	1.550×
stereovision2	57.356	53.450	0.932×
Average	-	-	**1.265**×

B. Analysis of execution time and Memory footprint

Execution time is a critical metric since we target application mapping at runtime, on the end user platform. For every optimization we propose we use the VPR flow for comparison as it is highly regarded in academia and it is flexible enough to support hard blocks like memories and multipliers. Since the VPR does not support multiple benchmarks, the comparison in terms of execution time is performed for mapping a single benchmark.

Memory usage and footprint is quickly becoming the computational bottleneck for FPGA CAD tools. In order to map applications onto FPGAs on runtime in the end user platform, it must be significantly reduced.

Table V shows the effect of the memory optimizations in terms of execution time and memory footprint on the original VPR and the proposed Het-JITPR. The "gains" and the "% of the VPR memory footprint" columns are in comparison with the original VPR's execution time and memory footprint.

The speed-up achieved in the original VPR with the memory optimizations is because: i) the universal optimizations cut down the computation time and ii) the memory footprint reduction results in faster computation and memory locality. If we add the optimizations with the use of the floorplanning step and our custom placer and router, we achieve a speed-up of around 40× on average compared to the stock VPR.

The memory footprint gains achieved with the proposed memory optimizations totalling in 40% reduction on average. It's evident that despite the large variation in the benchmark sizes we don't observe such variations in the memory reductions. That confirms the profiling results that suggested that the FPGA architecture is the main culprit for memory consumption in the process of application mapping. All memory optimizations are carefully developed in order to avoid affecting the quality of the final solution, meaning the application is placed and routed onto the FPGA in the exact same way with and without these optimizations. When we add also the algorithmic optimizations of the Het-JITPR we reach 90% reduction.

Table VI shows the effect of the custom dynamic memory allocators when all the othe memory optimizations are turned on. We see that both jemalloc and Lockless allocators result in an increase of 12% and 28% respectively on average in terms of memory footprint compared with the glibc's stock memory allocator. When comparing this with the execution time columns in table VI the trade-off between execution time and memory footprint is apparent. Jemalloc and Lockless achieve a 5% and 10% decrease on average execution time.

The reason behind this is two-fold: (a) their allocation strategy matches better the memory usage patterns of the Het-JITPR toolflow and (b) the memory allocators provide simpler and therefore faster locking mechanisms than glibc's malloc. When the requested sizes for memory allocation are limited and without large variance, these allocators perform much better than the default one. Both allocators try to improve the execution speed of their allocation function by moving some of the work to be done during de-allocation; sometimes

TABLE V
COMPARISON BETWEEN THE ORIGINAL VPR AND HET-JITPR TOOLFLOW WITH MEMORY OPTIMIZATIONS, IN TERMS OF EXECUTION TIME AND MEMORY FOOTPRINT.

Benchmark	Execution Time				Memory Footprint			
	VPR	VPR with Memopts	Het-JITPR with Memopts	Gain	VPR	VPR with Memopts	Het-JITPR with Memopts	% of the VPR memory footprint
arm_core	176.03	111.00	21.54	8.17x	5746	3417	655	11.39%
bgm	216.18	163.51	38.6	5.60x	6163	3509	1399	22.69%
blob_merge	146.81	83.28	5.02	29.25x	5497	3361	281	5.11%
boundtop	143.41	79.57	3.05	47.02x	5408	3337	191	3.54%
ch_intrinsics	140.02	76.10	1.2	116.68x	5339	3325	132	2.46%
diffeq1	141.62	77.44	1.45	97.67x	5350	3327	132	2.47%
diffeq2	140.79	76.75	1.11	126.84x	5339	3327	129	2.41%
LU8PEEng	196.81	140.54	33.91	5.80x	6049	3468	1121	18.54%
mkDelayWorker32B	159.81	92.66	14.59	10.95x	5510	3372	571	10.37%
mkPktMerge	142.28	80.73	3.1	45.90x	5363	3330	144	2.69%
mkSMAdapter4B	143.37	79.02	2.44	58.76x	5377	3338	176	3.26%
or1200	144.98	81.51	5.83	24.87x	5418	3343	297	5.47%
raygentop	142.94	79.29	3.44	41.55x	5384	3340	183	3.40%
sha	141.78	78.00	2.05	69.16x	5396	3338	181	3.34%
stereovision0	150.59	87.29	10.86	13.87x	5616	3402	408	7.27%
stereovision1	158.96	95.24	12.69	12.53x	5630	3410	502	8.92%
stereovision2	229.27	175.18	53.18	4.31x	6150	3534	1713	27.86%
Average	**159.74**	**97.48**	**12.59**	**42.29x**	**5572.67**	**3381.13**	**483.18**	**8.31%**

TABLE VI
COMPARISON BETWEEN THE CUSTOM ALLOCATORS IN HET-JITPR WITH ALL OTHER MEMORY OPTIMIZATIONS TURNED ON, IN TERMS OF EXECUTION TIME AND MEMORY FOOTPRINT.

Benchmark	Het-JITPR Execution Time			Het-JITPR Memory Footprint		
	Linux malloc (in seconds)	Jemalloc	Lockless	Linux malloc (in Mb)	Jemalloc	Lockless
arm_core	21,54	0,91x	0,88x	278	1,15x	1,22x
bgm	38,6	0,89x	0,85x	569	1,11x	1,02x
blob_merge	5,02	0,91x	0,85x	151	0,98x	1,44x
boundtop	3,05	0,97x	0,90x	136	0,85x	1,47x
ch_intrinsics	1,2	0,98x	0,93x	122	1,09x	1,23x
diffeq1	1,45	0,97x	0,93x	122	1,07x	1,26x
diffeq2	1,11	0,98x	0,96x	121	1,04x	1,06x
LU8PEEng	33,91	0,92x	0,90x	459	1,07x	1,01x
mkDelayWorker32B	14,59	0,95x	0,90x	301	1,33x	1,35x
mkPktMerge	3,1	0,99x	0,93x	125	1,06x	1,42x
mkSMAdapter4B	2,44	0,96x	0,89x	132	1,14x	1,27x
or1200	5,83	0,97x	0,90x	168	1,40x	1,49x
raygentop	3,44	0,97x	0,90x	133	1,16x	1,56x
sha	2,05	0,96x	0,90x	132	1,14x	1,23x
stereovision0	10,86	0,91x	0,87x	198	1,32x	1,43x
stereovision1	12,69	0,91x	0,88x	225	1,11x	1,32x
stereovision2	53,18	0,92x	0,88x	824	1,00x	0,94x
Average	-	**0,94x**	**0,90x**	-	**1,12x**	**1,28x**

they do not even decommit or purge the memory which the application no longer needs. These strategies, as well as the simple memory pattern of our application lead to a faster execution when these custom memory allocators are used.

Both allocators perform more aggressively than the original one and do not always return the de-allocated memory to the system; instead they prefer to keep managing it for future allocations that Het-JITPR might request. This eliminates the possible system call overhead for extra memory on a future request, but at the cost of additional memory fragmentation during application execution.It should noted that the memory footprint is always much smaller compared to the stock VPR's footprint.

C. Multiple designs onto the FPGA Fabric

The experimental results provided in the previous subsections affect benchmarks that are mapped onto the target reconfigurable architecture as stand-alone applications (assuming that our FPGA platform is empty). However, our proposed toolflow can implement multiple applications onto a single FPGA, even if other applications are already mapped to the device. In order to quantify this feature, we evaluate the efficiency of the proposed solution to handle multiple application implementations (i.e. through dynamic insertion and removal of applications).

Note that for this experimentation, the target architecture is similar to the previous cases. Regarding the sequence of

applications, we created a queue consisting of 110 circuits from a subset the previous benchmarks which are mapped dynamically onto the target architecture. More efficient scheduling algorithms, (like priority based scheduling) can be easily employed since our toolflow is developed as modular, but such algorithms are out of the scope of this research work.

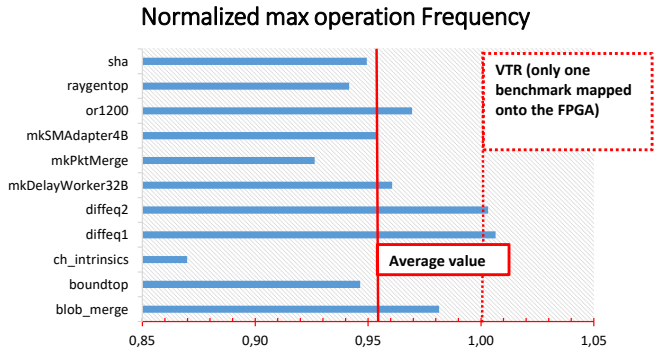

Fig. 2. Evaluation of the proposed framework when multiple applications are mapped onto the FPGA, in terms of maximum operation frequency

The flexibility of using even irregular contiguous areas for implementing the application, together with the proposed VKernel-Planner advanced heuristics and the finite resources of an FPGA platform affect the maximum operation frequency of each circuit. In our scenario of dynamic application mapping, the toolflow does not know a priori the characteristics of future applications.

Through our experimentation, each benchmark was mapped several times and the average maximum operation frequency is plotted at Figure 2. The reference solution corresponds to the performance of each benchmark when it is mapped onto the FPGA with the usage of VPR tool as a standalone application. On average the proposed solution has a penalty of only 4.7% with a small variation compared to our reference. This proves the efficiency of our solution even under runtime constrains.

VI. Conclusion

A novel methodology for supporting dynamic mapping of multiple applications onto an heterogeneous FPGA through VKernels was presented in this work. This methodology, along with memory optimizations and custom dynamic memory allocators, led to fast and efficient application implementation. Based on our experimentation, we achieve to perform application mapping $40\times$, faster on average,compared to a state-of-art approach, without any degradation in the maximum operation frequency, using only a small fraction, $1/12$, of memory used in a state-of-art tool for executing the proposed toolflow. This serves as a proof of concept that application mapping onto FPGA platforms can be performed dynamically even on end user embedded systems.

References

[1] T. Feist, "White paper: Vivado design suite," Xilinx Inc., San Jose, CA, USA,, Tech. Rep., 2012.

[2] "White paper:implementing fpga design with the opencl," Altera, Corp., San Jose, CA, USA, Tech. Rep., 2012.

[3] Altera, "Quartus ii development software," Altera Corporation, San Jose, CA, USA,, Tech. Rep., 2011.

[4] J. Rose, J. Luu, C. W. Yu, O. Densmore, J. Goeders, A. Somerville, K. B. Kent, P. Jamieson, and J. Anderson, "The vtr project: Architecture and cad for fpgas from verilog to routing," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '12. New York, NY, USA: ACM, 2012, pp. 77–86. [Online]. Available: http://doi.acm.org/10.1145/2145694.2145708

[5] A. Ludwin and V. Betz, "Efficient and deterministic parallel placement for fpgas," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 16, no. 3, pp. 22:1–22:23, Jun. 2011.

[6] M. Gort and J. Anderson, "Analytical placement for heterogeneous fpgas," in *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, 2012, pp. 143–150.

[7] H. Bian, A. C. Ling, A. Choong, and J. Zhu, "Towards scalable placement for fpgas," in *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '10. New York, NY, USA: ACM, 2010, pp. 147–156. [Online]. Available: http://doi.acm.org/10.1145/1723112.1723140

[8] C. Fobel, G. Grewal, and D. Stacey, "Forward-scaling, serially equivalent parallelism for fpga placement," in *Proceedings of the 24th Edition of the Great Lakes Symposium on VLSI*, ser. GLSVLSI '14. New York, NY, USA: ACM, 2014, pp. 293–298. [Online]. Available: http://doi.acm.org/10.1145/2591513.2591543

[9] B. Raza, H. Parvez, and M. Mohiuddin, "Exploring alternate trade-offs of placement quality versus runtime in simulated annealing algorithm," in *Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), 2014 9th International Symposium on*, May 2014, pp. 1–5.

[10] M. An, J. Steffan, and V. Betz, "Speeding up fpga placement: Parallel algorithms and methods," in *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*, May 2014, pp. 178–185.

[11] M. Gort and J. Anderson, "Accelerating fpga routing through parallelization and engineering enhancements special section on par-cad 2010," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 31, no. 1, pp. 61–74, Jan 2012.

[12] Y. O. M. Moctar and P. Brisk, "Parallel fpga routing based on the operator formulation," in *Proceedings of the 51st Annual Design Automation Conference*, ser. DAC '14. New York, NY, USA: ACM, 2014, pp. 193:1–193:6. [Online]. Available: http://doi.acm.org/10.1145/2593069.2593177

[13] H. Sidiropoulos, K. Siozios, and D. Soudris, "A framework for mapping dynamic virtual kernels onto heterogeneous reconfigurable platforms," in *2014 IEEE International Parallel & Distributed Processing Symposium Workshops, Phoenix, AZ, USA, May 19-23, 2014*, 2014, pp. 170–175. [Online]. Available: http://dx.doi.org/10.1109/IPDPSW.2014.23

[14] K. E. Murray, S. Whitty, S. Liu, J. Luu, and V. Betz, "Timing-driven titan: Enabling large benchmarks and exploring the gap between academic and commercial cad," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 8, no. 2, pp. 10:1–10:18, Mar. 2015. [Online]. Available: http://doi.acm.org/10.1145/2629579

[15] N. Douglas. (2013) ned productions - nedmalloc. [Online]. Available: http://www.nedprod.com/programs/portable/nedmalloc/

[16] L. Doug, "A memory allocator," Apr. 2000. [Online]. Available: http://gee.cs.oswego.edu/dl/html/malloc.html

[17] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson, "Hoard: a scalable memory allocator for multithreaded applications," *SIGPLAN Not.*, vol. 35, no. 11, pp. 117–128, Nov. 2000. [Online]. Available: http://doi.acm.org/10.1145/356989.357000

[18] J. Evans, "A scalable concurrent malloc (3) implementation for freebsd," in *Proc. of the BSDCan Conference, Ottawa, Canada*, 2006.

[19] H. Sidiropoulos, K. Siozios, P. Figuli, D. Soudris, M. Hübner, and J. Becker, "Jitpr: A framework for supporting fast application's implementation onto fpgas," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 6, no. 2, pp. 7:1–7:12, Aug. 2013. [Online]. Available: http://doi.acm.org/10.1145/2492185

[20] LockLess Inc. (2012) LockLess website. http://locklessinc.com/. Accessed: 2015-09-29.