

# A Coarse-Grain FPGA Overlay for Executing Data Flow Graphs

Davor Capalija and Tarek S. Abdelrahman

*The Edward S. Rogers Sr. Department of Electrical and Computer Engineering  
University of Toronto, Toronto, Canada  
{davor, tsa}@eecg.toronto.edu}*

**Abstract**—We explore the feasibility of using a coarse-grain overlay to transparently and dynamically accelerate the execution of hot segments of code that run on soft processors. The overlay, referred to as the Virtual Dynamically Reconfigurable (VDR), is tuned to realize data flow graphs in which nodes are machine instructions and the edges are inter-instruction dependences. A VDR consists of an array of functional units that are interconnected by a set of programmable switches. It can be rapidly configured by the soft processor at run-time to implement a given data flow graph. The use of a VDR overcomes two key challenges with run-time translation of code into circuits: the prohibitive compile time of standard synthesis tools and the limited run-time reconfigurability of commodity FPGAs. We conduct a preliminary evaluation that shows that the execution of a benchmark can be sped up by up to 9X over a Nios II processor using a benchmark-specific VDR overlay. The overlay incurs a 6.4X penalty in resources compared to Nios II. This work is a resubmission of earlier work that appeared in FCCM 2011 [1].

**Keywords**-Overlay architectures; dynamic acceleration of soft processors; just-in-time compilation

## I. INTRODUCTION

Recent years have seen an increasing interest in the use of Field Programmable Gate Array (FPGA) devices as accelerators in heterogeneous computing systems. FPGAs offer massively parallel resources that if exploited well can deliver raw computational power that surpasses traditional processors and even many-core architectures such as GPUs. It is no surprise that today several commercial systems offer heterogeneous high-performance computing (HPC) systems that integrate FPGAs, such as those produced by Convey Computers [2] and Faster Technology [3].

However, in order to exploit the massive parallel resources available on FPGAs, it is often necessary to use low-level hardware description languages (HDL) such as Verilog or VHDL. Regrettably, these languages result in lower productivity compared to high-level languages, such as C or Java. They also typically require extensive knowledge of the low-level FPGA fabric (e.g., LUTs and routing) and the corresponding tool chain (i.e., synthesis, place-and-route and timing analysis). The required hardware design expertise limits the potential of FPGAs to the narrow segment of hardware designers. The use of high-level programming languages would open up the FPGAs to a multitude of programmers, engineers and scientists. While hardware synthesis of high-level languages has been an active area of research for a couple of decades, this approach has had limited success [4].

In this paper, we explore raising the level of abstraction of programming FPGAs through the use of a coarse-grain overlay that supports the execution of data flow graphs. The overlay hides all of the intricacies of the FPGA fabric (such as LUTs and timing) yet it exposes a massively parallel compute fabric.

The overlay can be targeted by either a static compiler or a dynamic run-time system. The overlay, which we term the Virtual Dynamically Reconfigurable (VDR) overlay, can be implemented on any commodity FPGA. It consists of a mesh of 32-bit functional units (such as adders and multipliers) interconnected by programmable network of 32-bits wide switches that is configured at run-time. The overlay facilitates the implementation of data flow graphs: the nodes of such a graph (machine instructions) are mapped to VDR's functional units and its edges (inter-instruction data flow) are realized by configuring the VDR's programmable switches.

In this paper, we explore the feasibility of using the VDR overlay in the context of a Just-In-Time (JIT) compiler in order to dynamically translate machine instructions executing on a soft processor into FPGA circuits. The soft processor executes machine instructions and monitors the execution to detect frequently executed (i.e., *hot*) segments of code. A circuit that implements the data flow graph of the trace is realized on the overlay and the sequential code is re-written to include control transfer to the overlay. Subsequent executions of the hot code segment are performed on the overlay resulting in better performance.

The above use of the VDR overlay allows us to overcome two challenges often associated with run-time generation of FPGA circuits. The first is prohibitive compile time of synthesis tools to realize the necessary circuits (typically several minutes, if not hours). The second is that commodity FPGAs offer limited facilities for dynamic reconfiguration, although next generation FPGAs, such as Altera's Stratix V or Xilinx's Virtex 7, promise better measures in this respect. Through the use of an offline pre-synthesized VDR overlay with a radically faster run-time mapping of data flow graphs onto the overlay we are able to overcome these challenges.

We describe data flow graphs (DFGs) and traces and their advantages as units for realizing circuits. We also present our VDR overlay and describe how trace data flow graphs can be efficiently mapped onto it to exploit parallelism. We use an EEMBC benchmark to evaluate and demonstrate the feasibility of our approach: compared to soft processor execution, up to 9X performance improvement is achievable using an application-tailored overlay, while incurring only 6.4X resource overhead.

The remainder of the paper is organized as follows. Section II describes data flow graphs and traces. Section III describes the VDR overlay. Section IV describes our model of application acceleration via DFG-to-VDR mapping. Section V presents our experimental evaluation. Related work is described in Section VI. Finally, Section VII gives concluding remarks.

The work presented in this paper re-presents our earlier work that appeared in FCCM 2011 [1].

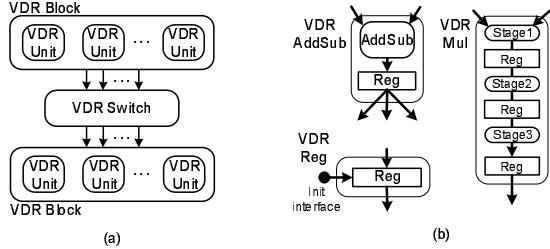


Figure 1. (a) Building blocks of the VDR Overlay architecture, (b) The architecture of several VDR units

## II. DATA FLOW GRAPHS AND TRACES

A data flow graph (DFG) is a graph that expresses data dependencies between a set of operations or instructions. The data dependencies are explicit and represent only true dependences, there are no false dependences. The nodes in the DFG represent instructions (such as arithmetic or logic operations) and edges represent the flow and data dependences between instructions. The data flow graph execution model is data-driven, a node can execute as soon as its input nodes are ready. We believe that such abstraction is a natural choice for exploiting massive FPGA parallelism. The data-driven execution can be implemented in a distributed and localized fashion, both are key design techniques for achieving high-performance circuits in FPGA fabric.

Traces are straight-line segments of code and can be modeled as directed acyclic data flow graphs. Traces have appealing properties [5] that motivate us to use them as units for the run-time mapping to the overlay. There is a large number of traces in a program, which allows high utilization of large commodity FPGAs. Traces are short, a few dozens instructions on average, which makes them a good starting point for overlay mapping. Furthermore, since traces represent only hot paths of execution they omit infrequently executed code, which results in more resource-efficient circuits. Finally, traces execute frequently, which indicates that there will be repeated execution of traces as circuits and therefore high potential for performance improvement. Nonetheless, traces do have drawbacks. In particular, it is possible for execution to leave a trace before the end of the trace (*early exits*). This necessitates a mechanism that falls back to soft processor execution.

## III. THE VDR OVERLAY

The VDR overlay architecture lends itself to easy mapping of data flow graphs of various topologies. Its building blocks, shown in Figure 1(a), are VDR blocks and unidirectional VDR switches. A VDR block consists of multiple VDR units, each a 32-bit functional unit (FU) that performs the operation of one (or several) instruction(s) of the soft processor's instruction set. The overlay is synchronous and all VDR units use the same clock. The units within the VDR block are not connected and can operate in parallel. Different blocks may contain different numbers of units, and the units within a block may or may not be the same. A VDR switch connects two VDR blocks, allowing any unit in one block to be connected to any unit in the other block. Each VDR block can be connected

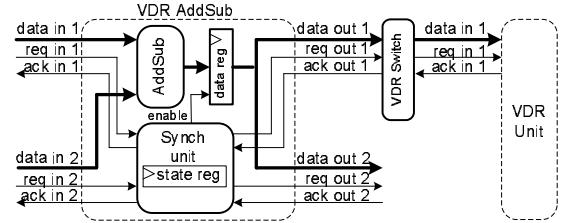


Figure 2. The AddSub unit with 2 in and 2 out edges

to a single block or multiple blocks, which makes it possible to arrange VDR blocks in various topologies, both regular and irregular.

### A. VDR Units

The architecture of several VDR units is shown in Figure 1(b). Each VDR unit has a reconfiguration interface to configure it to a desired functionality at run-time. An example unit is a VDR AddSub unit, which can implement all addition and subtraction instructions (*add*, *addi*, *sub*, *subi*, etc. in Nios II). VDR Load and Store units have a special *memory interface* which is connected to a *memory scheduler*, which in turn accesses the memory subsystem.

Each VDR unit that produces a result has a 32-bit register to store the result. The VDR Reg unit contains only a 32-bit register and an *initialization interface* which serves to communicate register values between the processor and VDR overlay. The maximum number of input/output edges of a unit is determined during offline synthesis. The latency of VDR units is matched to the latency of the processor's instructions they realize. This allows VDR units to have the same maximum operating frequency ( $F_{max}$ ) as the functional units in the processor.

### B. Inter-unit Communication and Synchronization

Our VDR overlay uses data-driven execution of VDR units through the use of a *synchronous handshake protocol* among the units. This eliminates the need for a centralized finite-state machine, global communication or broadcasting. Each VDR unit has a small control circuit for synchronization with its producer and consumer units using (1) a data bus, (2) a data ready request signal (*req*) and (3) data received acknowledge signal (*ack*). Figure 2 shows the control circuit for the VDR AddSub unit with 2 input edges and 2 output edges. The protocol allows pipelined execution with the throughput of 1 data element per clock cycle, and stalling for a variable number of cycles by keeping *ack* low. The progress of VDR units in a pipeline is synchronized by back-propagating *ack* in the same clock cycle.

An issue with the back-propagation of *ack* arises in deep pipelines of units and a circular pipeline of units. The first leads to a long combinational path of the *ack* signal, and the second leads to a combinational loop. Depending on the topology of the VDR overlay, a long combinational path of synchronization signals can negatively impact the overlay's  $F_{max}$ . In order to address this issue, we introduce a VDR *SyncBreak* which can be inserted into the data flow between units. The *SyncBreak* unit breaks the combinational path of

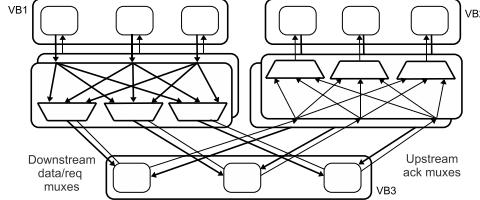


Figure 3. VDR switches among three VDR blocks

*ack* signals, into two combinational paths by registering the *ack* signal. Since this delays the *ack* signal by one cycle, the SyncBreak does not know whether the data is received by the consumer and hence buffers the current data in a 32-bit register. In the event that the consumer has stalled (*ack* is low) the data will not be lost. The VDR SyncBreak unit presents a trade-off: additional resources are invested in order to localize synchronization, achieve higher overlay  $F_{max}$  and allow throughput of 1 data/clock cycle.

### C. VDR Switches

A VDR switch provides many-to-many connectivity among VDR units in two connected VDR blocks. Switches employ only combinational circuits they do not add latency in terms of clock cycles. Figure 3 shows an example of a switch design for the  $VB_3$  block that is connected to two blocks,  $VB_1$  and  $VB_2$  with two switches. Both switches consist of six 3-to-1 multiplexers. We assign a mux to the first (second) incoming edge of a node in  $VB_3$ , and connect the mux inputs to all outgoing edges of  $VB_1$  ( $VB_2$ ).

Each mux requires a reconfiguration register with  $\lceil \lg(\text{mux\_size}) \rceil$  bits. The VDR has a *reconfiguration interface* that facilitates the configuration of the muxes in the overlay. The interface is a chain of reconfiguration registers ( $N$  is the number of registers, and  $M$  is the size of the largest register).

## IV. APPLICATION ACCELERATION THROUGH DATA FLOW GRAPH EXECUTION

We exploit parallelism at three levels: instruction-level, pipelined and DFG-level parallelism, as illustrated by the example shown in Figure 4.

A soft-processor can execute up to one instruction per cycle (IPC) because it has one FU. In contrast, the VDR overlay contains many FUs that execute in data-driven fashion. Within a single DFG, we exploit instruction level parallelism (Figure 4(c)). A DFG (generated from a trace in this case) often represents a loop body and it may be possible to pipeline the consecutive execution of multiple trace instances on the same overlay-mapped DFG (Figure 4(d)). At steady state, most units of the overlay DFG are utilized, exploiting pipelined parallelism. Finally, there may exist parallelism among traces [6] and it is possible to exploit this trace-level (or DFG-level) parallelism by replicating the DFGs on the overlay (Figure 4(e)).

The trace-to-VDR mapping algorithm first creates the DFG of the trace and breaks false dependences among instructions. Next, the DFG is augmented with VDR Reg nodes that correspond live-upon-entry registers and in-code constants. Edges

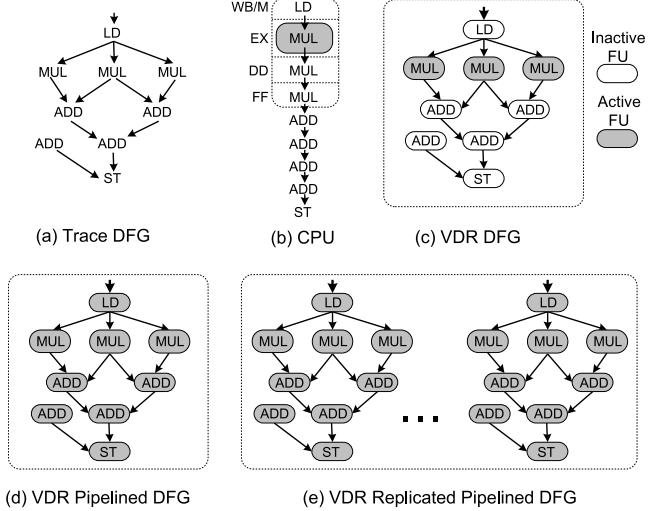


Figure 4. Exploiting different levels of DFG parallelism on the VDR overlay

that represent dependences among instructions in consecutive traces are added to enable the pipelined execution of traces on the same overlay-mapped DFG. The DFG is then partitioned to match the blocks that exist in the overlay. The DFG nodes are then mapped to the overlay. If a node cannot be mapped, the algorithm backtracks. Finally, the VDR reconfiguration bitstream is then loaded into the overlay.

## V. EXPERIMENTAL EVALUATION

We explore the feasibility of our approach by building an experimental system on an Altera Stratix III FPGA platform. It consists of a Nios II/f soft processor, two VDR overlay replicas, and a VDR *control interface* that allows Nios II to reconfigure the overlay. Each overlay instance is connected to a 2-port on-chip memory through a *memory scheduler*. In order to assess the  $F_{max}$  and resource impact of the reconfigurable switches of the overlay, we also design a hardwired version of the overlay in which the switches are replaced by direct wires.

We use a benchmark from the consumer category of the EEMBC suite and extract the sole trace that exists in it. We then design a tailored VDR overlay that is able to realize (i.e. fit) the DFG of that trace. This avoids the challenge of designing a generic overlay in this early stage of our evaluation.

We characterize the impact of the switches and of the memory subsystem utilization. We present the speedup of the execution of the benchmark with the overlay over its execution on Nios II. We also measure the overhead of reconfiguration and show that it is minimal.

### A. The RGB2YIQ Benchmark

The benchmark consists of a loop that iterates over all the pixels in the image; each iteration converts a 3-byte RGB-encoded pixel into a 3-byte YIQ-encoded pixel. The assembly code of the loop is obtained by compiling the source code with `gcc -O2`. The loop contains one trace that corresponds to the

VDR unit	# of units	Latency (cc)	Avg. ALUTs	Avg. FFs	Avg. DSPs
AddSub	16	1	70.7	33.3	0
Mul	9	3	8	3	9
Load	3	2+	74.7	35	0
Store	3	1+	65	0	0
Shift	3	3	131	105	0
Compare	1	1	32	2	0
Reg	15	1	34	33	0
SyncBreak	6	1	110	68	0

Table I  
PROPERTIES OF VDR UNITS

System	Nios II	VDR overlay	Hardwired data flow
$F_{max}$	290	172	286
ALUTs	1483	4753	2978
Logic FFs	1340	2065	1824
DSP Blocks	4	36	36

Table II  
 $F_{max}$  AND RESOURCE USAGE COMPARISON

entire loop body, has 35 instructions and no internal control flow. We use a  $640 \times 480$  image as input.

### B. VDR Overlay

The designed overlay contains 56 units, 31 blocks and 41 switches. The blocks range in size from 1 unit to 3 units per block. The overlay switches are built of 148 muxes. There is a total of 200 edges, which is more than twice the 74 edges in the trace data flow graph. This is because, although the VDR overlay is application-tailored in terms of its units, the switches allow creation of a number of data flow graph topologies to provide a measure of flexibility. Table I lists the VDR units comprising the VDR overlay. Load and store units can execute for variable number of cycles depending on the contention and throughput of the memory subsystem.

There are 148 2-bit reconfiguration registers in the VDR reconfiguration bitstream. The Nios II reconfigures the overlay by sending this bitstream to the overlay. Also, it sends seventeen 32-bit initial register values and constants.

### C. Overlay $F_{max}$ and Resource Usage

Table II compares the  $F_{max}$  and FPGA resource usage of Nios II, the VDR overlay and the hardwired data flow. The overlay usage reflects all the resources needed to implement the overlay including the VDR units, the switches and the reconfiguration registers. It shows that the hardwired data flow effectively matches the operating frequency of the processor. Once the switches are added, the maximum frequency drops to about 60% of the hardwired version.

One overlay instance consumes 3.2X the ALUT resources of Nios II. Further, the VDR overlay incurs overhead in terms of additional ALUTs and flip-flops (FFs) compared to the hardwired data flow, which accounts for the switches and their reconfiguration registers. We believe that this is an acceptable resource overhead to gain flexibility in the overlay, particularly given that FPGAs will soon have in excess of a million LUTs.

Version name	data flow replicas	Ports/Mem (total Ports)	Time (ms)	Speedup
Nios II	—	—	38.31	1
VDR-1	1	1 (1)	11.57	3.31
VDR-2	1	2 (2)	7.91	4.85
VDR-2x2	2	2 (4)	4.34	8.82
HDW-1	1	1 (1)	6.53	5.87
HDW-2	1	2 (2)	4.32	8.87

Table III  
BENCHMARK SPEEDUP FOR DIFFERENT SYSTEM VERSIONS

### D. Performance

We measure *speedup*, defined as the ratio of the execution time of the benchmark running only on Nios II to its execution time running on both Nios II and the overlay. In the latter case, the execution time includes: (1) the initial execution of 40 iterations on Nios II to detect the trace, (2) trace-to-VDR mapping, (3) overlay reconfiguration, (4) initialization of live-upon-entry registers, and (5) overlay execution. In this initial evaluation, the time to form the trace is not included since the trace is known a priori. However, this time is typically small [5], [6]. Therefore, our speedups are a tight upper bound on what is achievable.

We use three overlay versions and two hardwired versions. VDR-1 and VDR-2 use one and two ports to access memory, respectively. VDR-2x2 uses two replicas of VDR-2, each processing half of the image. HDW-1 and HDW-2 are hardwired versions of VDR-1 and VDR-2, respectively. Table III shows the speedups of these versions. The speedup of VDR-1 compared to HDW-1 shows the impact of the switches. Although both VDR-1 and HDW-1 execute the same number of clock cycles, VDR-1 is clocked at a lower frequency, and includes the overheads of trace-to-VDR mapping and VDR reconfiguration, resulting in lower speedup. Thus, it is critical that the number and layout of VDR switches on an FPGA be selected to minimize their impact.

Memory scheduler analysis for VDR-1 reveals the single memory port is close to full utilization and presents a bottleneck. Two ports in VDR-2 improve speedup by over 45%, and the utilization of each memory port is 74%. Finally, overlay replication in VDR-2x2 doubles the performance indicating the potential for DFG-level parallelism.

The execution time breakdown reflects that the overheads of using the overlay are negligible. In the case of VDR-2, the reconfiguration time accounts for 0.01%, while register initialization represents another 0.01%. The initial execution of the 40 loop iterations represents 0.06%. In contrast, the trace-to-VDR mapping represents 9.19%. This indicates the importance of making the mapping step more efficient.

## VI. RELATED WORK

Improving the performance of soft processors has been attempted through: application-specific pipeline customization [7], VLIW [8], vector extensions [9] multi-threading or multiple processors [10], [11], [12]. These approaches require either user effort, a specialized compiler flow or generate non-standard binaries. Our approach works with standard soft-processor binaries and transparently improves performance.

Lysecky et al. [13] propose the warp processor to dynamically accelerate program binaries on a special-purpose LUT-based FPGA using leaner CAD tools. Bergeron et al. [14] propose JIT-compilation of Scheme programs to commodity FPGAs using user annotations and standard CAD tools. In contrast, we circumvent the run-time use of CAD tools via a pre-synthesized overlay.

Shukla et al. [15] propose QUKU, which uses a systolic array of general-purpose ALUs targeted by a static compiler. In contrast, our overlay uses heterogeneous FUs with a flexible interconnect that is suitable for run-time mapping of data flow graphs. Kapre et al. [16] and Lucas et al. [17] explore overlay networks of larger processing elements (CPUs), whereas we focus on smaller functional units. Coole et al. [18] propose a coarse-grain overlay which features 16-bit functional units, an FPGA-like interconnect (routing channels and switch boxes) and a global control unit. In contrast, our approach uses a flexible nearest-neighbor interconnect and a data-driven execution with all control logic embedded in the VDR units. Grant et al. [19] propose an FPGA device which contains both coarse- and fine-grained elements. Although their approach reduces synthesis time it requires Verilog and low-level hardware knowledge. In contrast, our approach uses an overlay for commodity FPGAs which hides all low-level FPGA details.

## VII. CONCLUSION AND FUTURE WORK

We explored the feasibility of user-transparent soft processor acceleration through the use of traces and a pre-synthesized overlay. Our approach eliminates the run-time use of computationally expensive CAD tools by reducing code-to-circuit synthesis to a much simpler trace-to-overlay mapping. Our preliminary evaluation using a benchmark, an overlay tailored to this benchmark and a Nios II processor shows that up to 9X improvement in performance over the soft processor can be achieved. This leads us to conclude that our approach is feasible.

Our long term goal is to design and build a complete system (called Kitsune) for user-transparent acceleration of soft processors, shown in Figure 5. It consists of a soft processor, a generic application-independent VDR overlay, a trace collection system (TCS), a trace cache, a trace-to-VDR mapping module, a VDR bitstream cache and a trace dispatcher. The TCS collects traces and deposits them in the trace cache. These traces are mapped onto the VDR using the trace mapper and the bitstreams are stored in the VDR bitstream cache. The trace dispatcher loads multiple bitstreams to the VDR configuring it for the appropriate traces.

## REFERENCES

- [1] D. Capalija and T. S. Abdelrahman, "Towards Synthesis-Free JIT Compilation to Commodity FPGAs," in *Proc. of FCCM 2011*.
- [2] Convey-Computers, <http://www.conveycomputer.com/>.
- [3] Faster-Technology, <http://www.fastertechnology.com/>.
- [4] G. Martin and G. Smith, "High-level synthesis: Past, present, and future," *Design Test of Computers, IEEE*, vol. 26, no. 4, pp. 18 –25, july-aug. 2009.
- [5] B. J. Bradel and T. S. Abdelrahman, "A characterization of traces in java programs," in *Proc. of PLC'05*.
- [6] ———, "Automatic trace-based parallelization of java programs," in *Proc. of ICPP'07*.
- [7] P. Yiannacouras, et al., "Application-specific customization of soft processor microarchitecture," in *Proc. of FPGA'06*.
- [8] A. K. Jones et al., "An FPGA-based VLIW processor with custom hardware execution," in *Proc. of FPGA'05*.
- [9] P. Yiannacouras et al., "Fine-grain performance scaling of soft vector processors," in *Proc. of CASES '09*.
- [10] D. Unnikrishnan et al., "Application-specific customization and scalability of soft multiprocessors," in *Proc. of FCCM'09*.
- [11] M. Labrecque and G. Steffan, "Improving pipelined soft processors with multithreading," in *FPL'07*.
- [12] F. Plavec, Z. Vranesic, and S. Brown, "Enhancements to FPGA design methodology using streaming," in *Proc. of FPL'09*.
- [13] R. Lysecky and F. Vahid, "Design and implementation of a MicroBlaze-based warp processor," *ACM TECS'09*, vol. 8(3), 1–22.
- [14] E. Bergeron et al., "Hardware JIT compilation for off-the-shelf dynamically reconfigurable FPGAs," in *Proc. of CC'08/ETAPS'08*.
- [15] S. Shukla, N. W. Bergmann, and J. Becker, "QUKU: A FPGA based flexible coarse grain architecture design paradigm using process networks," in *Proc. of IPDPS'07*.
- [16] N. Kapre et al., "Packet switched vs. time multiplexed FPGA overlay networks," in *Proc. of FCCM '06*.
- [17] A. D. C. Lucas et al., "Application development with the FlexWAFE real-time stream processing architecture for FPGAs," *ACM TECS'09*, vol. 9(1), 1–23.
- [18] J. Coole and G. Stitt, "Intermediate fabrics: Virtual architectures for circuit portability and fast placement and routing," in *Proc. of CODES+ISSS 2010*, pp. 13 –22.
- [19] D. Grant, C. Wang, and G. G. Lemieux, "A CAD framework for Malibu: an FPGA with time-multiplexed coarse-grained elements," in *Proc. of FPGA 2011*, pp. 123–132.

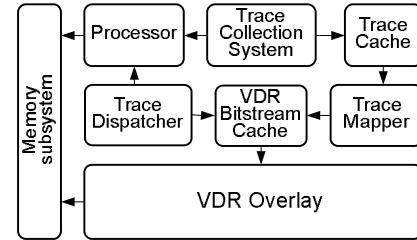


Figure 5. The Kitsune System