High-Level-Language Compilation for Reconfigurable Computers

Andreas Koch
Tech. Univ. Darmstadt
Embedded Systems and Applications Group (ESA), FB20
Darmstadt, Germany
Email: koch@esa.informatik.tu-darmstadt.de

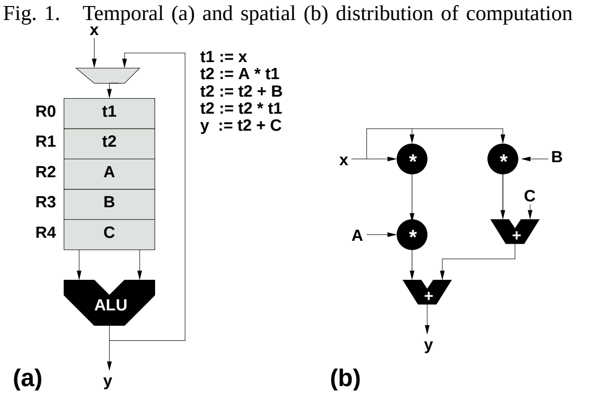
Nico Kasprzyk
Tech. Univ. Braunschweig
Dept. for Integrated Circuit Design (E.I.S.)
Braunschweig, Germany
Email: kasprzyk@eis.cs.tu-bs.de

Abstract— **The compiler COMRADE accepts full ANSI C and compiles it into hybrid hardware/software applications for execution on a reconfigurable adaptive computer system. After defining the model-of-computation underlying the flow, we describe the general compilation process as well as some specific techniques. These include path-based partitioning, module-generator based datapath and controller synthesis, as well as reconfiguration scheduling.**

I. Introduction

The tremendous progress in microelectronics no longer linearly translates into increased computer performance. While we can build chips with billions of transistors, most of the processor architectures implemented on them still follow patterns developed in the mid-1940s for vacuum-tube based systems.

Fig. 1. Temporal (a) and spatial (b) distribution of computation



As an alternative, reconfigurable/programmable logic devices (PLD) allow the implementation of compute units (CU) following a different paradigm: Instead of *temporally* distributing a computation across shared compute resources under control of a software program (Figure 1.a), it is distributed *spatially* across dedicated compute resources, increasing the parallelism of the computation (Figure 1.b).

Conventional CPUs and CUs complement each other well: The computation-intense kernels of an application can be spatially mapped to CUs, while the less critical or unsuitable parts remain temporally mapped on the CPU. Together, both kinds of processors form an *adaptive computing system* (ACS).

The CUs themselves are realized by mapping them onto a reconfigurable fabric (RF), which may be implemented, e.g., by an FPGA or a coarse-grained device, sometimes called a field-programmable node array (FPNA). Such ACSs can accelerate a broad variety of applications [3] [4] [5].

Lacking, however, are abstractions and software tools to make the potential of adaptive computers accessible to the majority of applications programmers, who are unfamiliar with the digital circuit design methods currently used to “program” an ACS.

The complexity involved in developing automatic compilers targeting adaptive computers is significant: In addition to conventional compiler technologies, issues of automatic hardware synthesis both at the architectural and logic level also have to be considered. Further intricacy is involved in partitioning the application between the conventional and the reconfigurable processor, and generating communications interfaces between them in hardware (HW) and software (SW). A completely novel topic, even in HW design, is the full exploitation of reconfigurability, which in itself can occur in various forms.

In this work, we present an overview over our research on COMRADE, a third-generation tool flow for automating the mapping from a conventional high-level language (HLL) to a program running on an adaptive computer. We describe specific areas of progress, ranging from abstract models of computation down to the efficient composition of datapaths from HW operators.

II. Sample Applications

In the following discussion, we will present the effect of various translation phases when processing a number of real applications. These are versatility a Wavelet-based image compression [6], the adpcm and g.721 audio compressors, the capacity Huffman-table generator [7], and the elliptic-curve cryptography tool pegwit [8].

III. Fundamentals

In contrast to a human designer, who can creatively choose the best abstraction from a wide spectrum for solving the current problem, an automatic tool flow is constrained by the paradigms and schemes statically encoded in its algorithms.

```
...
u = a + b;
v = c - d;
for (n=0; n<10; ++n) {
  v += u;
  if (v > 1000)
    printf("error v=%d", v);
}
w = 53 * v;
...
```

Fig. 2. Input program with CU-infeasible statement

Application	Variables transfered	
	statically	at run-time
versatility	0	0
adpcm	0	0
g.721	0	0
capacity	12	1600
pegwit	0	0

TABLE I
COMMUNICATIONS OVERHEAD FOR CU-CPU EXECUTION SWITCHES

Thus, they have to formulated carefully to yield the best quality of results across a broad range of applications.

A. Programming Abstraction

Since our primary aim is to make ACS technology accessible to a large user base, we have chosen C as the input language for the compiler. Specifically, we aim to support the full ANSI C language. This includes pointers, function calls, and control flow in loops. Furthermore, the user is not required to give the compiler translation hints using pragmas or similar annotations. We are aware that C is not the most suitable language for describing HW designs in. However, it still is the most widely used language in the embedded computing area. In our project, this dominance takes precedence over the possibly improved quality of results achievable using a specialized, but more exotic, programming language as input format.

During compilation, computation-intense *kernels* (generally loops) are extracted from the C source code and mapped to dedicated CUs implemented on the RF. Since ANSI C does not contain constructs for explicitly modeling parallelism (e.g., on the thread-level, TLP), we are limited to exploiting instruction-level parallelism (ILP). However, this is augmented with speculative execution of conditional branches and restructurings (profile-based inlining, loop transformations) that increase parallelism.

The parts of the input program that cannot be efficiently mapped to a CU, such as calls to the C standard library or floating point computation (due to excessive area requirements of current RFs), remain in SW.

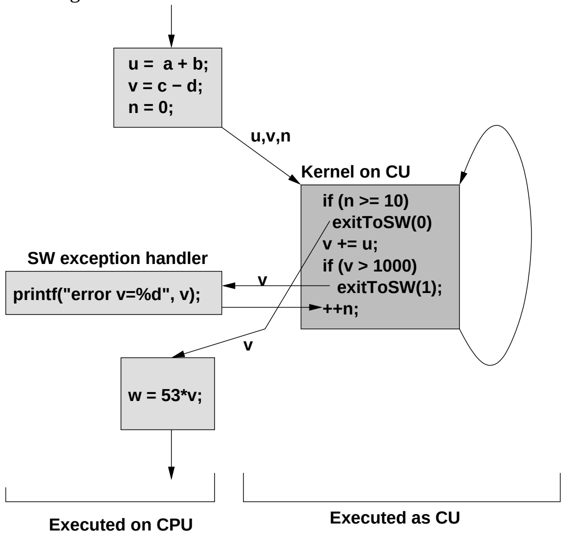
B. Model of Computation

Our underlying model of computation orchestrates this interaction between SW executing on the CPU and CUs executing on the RF. The fundamental idea is, that both kinds of processors assist each other in the fast and efficient (e.g., in terms of RF area and CPU power consumption) execution of the program:

- The RF assists the CPU, which is possibly of a slower, low-power variety, by efficiently accelerating the computation intense parts of the program.
- The CPU assists the RF by executing the rarely executed (e.g., error handling) or CU-unsuitable parts of the program, which would require massive area on the RF.

To demonstrate this approach, Figure 2 shows a fragment of C code that contains rarely executed parts (the assignments to u, v, n, w) and a kernel consisting of a for-loop. However, the kernel contains a printf() function call to the standard C library, which can neither be avoided by inlining nor efficiently realized as part of the CU. Thus, the relevant data (only v in the example) is transferred to a SW *handler* providing this functionality as a *service* to the CU (Figure 3).

Fig. 3. CPU-CU interaction for CU-infeasible code



After executing the SW handler, the execution of the CU is resumed. If the handler had changed the values of variables ‘live’ in the CU (not required in this example), they would be transferred back to CU analogously.

Obviously, the degree of interaction between the CPU and CU is limited by the communications overhead of this HW/SW execution mode switch. If it is too high, any performance gain by the CU is lost in data transfer time. To this end, communications are considered both in the architecture of the target system (Section III-C) as well as during the HW/SW partitioning step itself (Section V). In practice, the majority of kernels isolated by HW/SW partitioning of real applications have only negligible communications overhead (Table I).

Note that the CPU and the CU do *not* execute in parallel in this model. As described previously, C does not allow the modeling of thread-level parallelism. Thus, the assignment of operations to CPU and CUs is an exclusive one. Ideally, however, this will not be a disadvantage: If the HW/SW partitioning performs its task well, it will have moved all relevant kernels from the CPU onto the CU. From a practical perspective, this should allow a reduction in the peak-power consumption of the system, since the now-idle CPU can be set into a sleep-state. However, our current prototyping HW platform [1] does not allow measurements confirming this hypothesis.

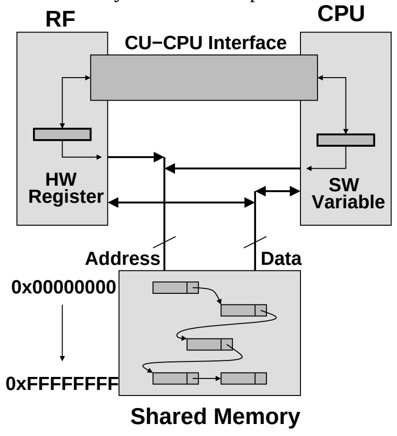
C. Target System Architecture

The choice of C as input language and the associated model of computation significantly influence the architecture of our target system.

1) SW-CU Communications: The need for quick SW-CU communication requires a *low-latency* communications channel. Since a good partitioning can often avoid excessive communications, our model relies on the quick interchange of relatively small quantities of data. In our experiments, the only 12 variables of 32b each need to be transferred between all CUs and CPU. Thus, the bandwidth requirements are rather low.

2) Memory System: Since we fully support pointer accesses as well as the interchange of pointer values (addresses) between CUs and SW, the RF must have independent (master-mode) access to a memory space shared with the CPU (Figure 4).

Fig. 4. Shared memory and address space between CU and CPU



While this requirement is not problematical from a bandwidth perspective (RF and CPU do not execute concurrently), caches have to be organized carefully. Ideally, the RF and CPU would share a common cache hierarchy to avoid inconsistencies. This can be realized in an SoC context by integrating the RF with a configurable processor core [9] [10], or using the tightly-coupled memory approach of some embedded processor core (as in certain ARM7 and ARM9 types) [11]. If the sharing of cache hierarchies is not possible, more heavyweight coherency solutions such as bus-snooping or mutual cache invalidation (used on our current platform) could be employed.

3) RF Processing Element Granularity: The native widths of data types in C are 8b, 16b, 32b, and 64b words. Of these, 32b operations are still the most common (default size of the standard integer type). Thus, the most efficient target RF architecture would have relatively coarse-grained processing elements (PE) and multi-bit bus interconnects. Examples would include composable 4b or 8b logic blocks or ALU slices. However, since such PLDs are still rather uncommon, we are currently mapping to FPGA-based RFs and compose the wider operations from the 4-LUT PEs (Section VI). The inefficiency of the too-narrow PEs can to some degree be ameliorated by specialized optimizations. For example, bit-wise constant propagation can recognize variable, unused, and constant values within a variable or expression at the granularity of single bits. This can then be used to specialize the HW operators at the bit-level, exploiting the fine granularity of FPGA PEs.

4) Reconfigurability: In our model of computation, each CU encompasses *all* of the parallelism extractable from a kernel sequentially described in C. With no TLP in ANSI C, only a single CU will thus execute at a time on the RF. Thus, for our purposes, a RF holding just a single CU, but capable of rapid reconfiguration (possibly assisted by a configuration cache) would be ideal. But again, most commercially available PLDs do not have this ability. It can be emulated, however, using mechanisms actually available in current chips:

- By using *partial* reconfigurability, we could combine background-loading techniques (such as double or triple-buffering) with a logical partitioning of the RF. This would allow to pre-configure the next CUs to be executed (possibly speculatively) into currently unused logical partitions, without affecting the currently executing RF. Such systems have already been demonstrated [12] [13].
- With a sufficiently large RF, we can use *configuration merging* to efficiently fuse multiple CUs into a smaller number of larger configurations. These can then be configured without requiring partial configuration capabilities from the RF. *Soft* reconfigurations can then occur just by switching multiplexers on the RF between different CUs, *hard* reconfigurations actually load a new bitstream. This approach does not require partial reconfigurability from the underlying PLD.

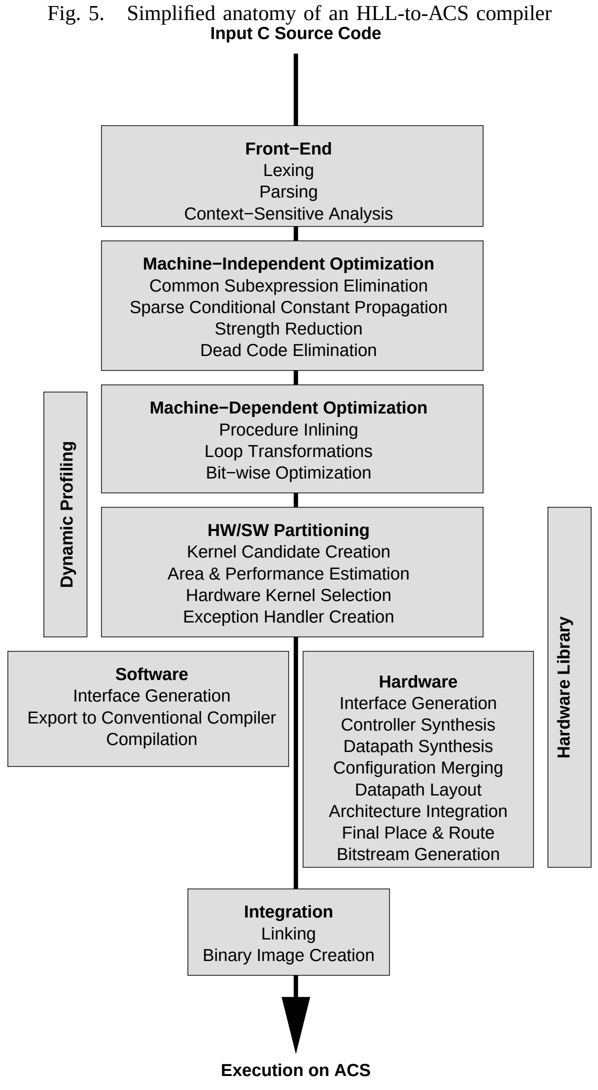
Due to the immaturity of software tools dealing with partial reconfiguration of commercial PLDs, our current system supports the second approach (see Section VII).

IV. Anatomy of an ACS Compiler

Figure 5 gives an overview over the general anatomy of an ACS compiler. Up to the HW-SW partitioning step, the phases are not ACS-specific. Two general support facilities aid many of the later translation phases:

First, a number of *dynamic profiling* analyses observes the behavior of the input program during run-time when processing a given data set. Many optimization decisions, e.g., dealing with the fraction of execution time spent in a certain function or the communication patterns between blocks, are based on this data. Dynamic profiling is generally more accurate than static estimation methods, but requires a representative

Fig. 5. Simplified anatomy of an HLL-to-ACS compiler



input data set when the program is being monitored during execution.

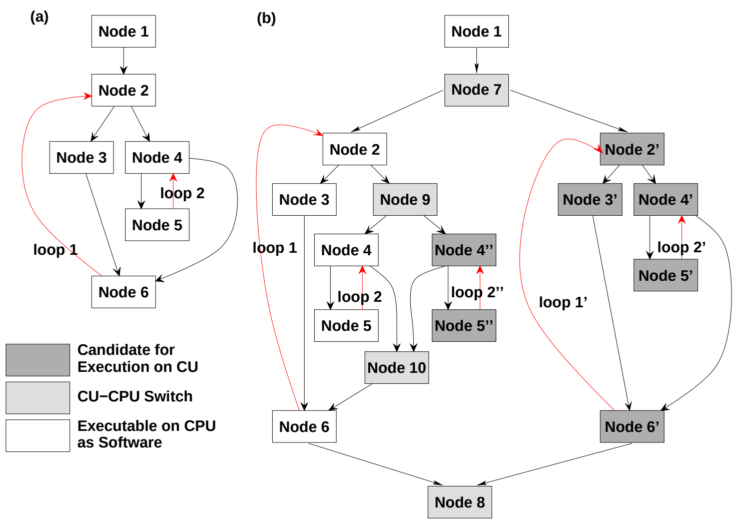
Second, when estimating the effect of moving operations to the CU, the same module library used for the actual HW generation can be queried by the main flow for HW operator characteristics. This occurs via a target technology-independent interface. It gives each step in the flow a dedicated view into the data. Examples for different views include operator behavior and interface, timing, area requirements, circuit topology etc., down to actual placed netlists of the operator hardware.

The next Sections will examine some of the phases in greater detail.

V. HW/SW-Partitioning

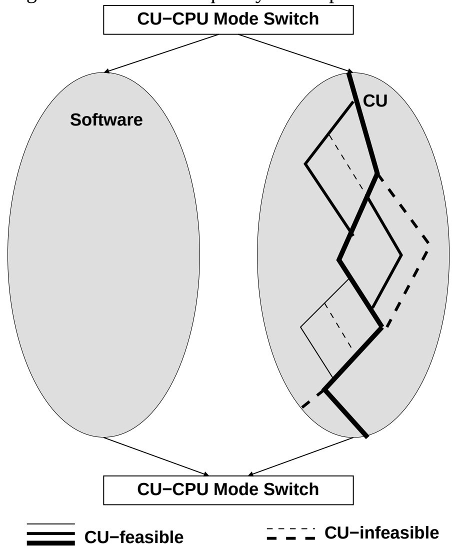
Currently, the HW/SW partitioning step attempts to move as much of the kernels in the input program to CUs as possible. For each kernel, this is not an all-or-nothing operation. Instead, an incremental method is used [16]:

Fig. 6. Generating possible CUs from all-SW version



- For each loop nest forming the kernel currently under consideration, all *combinations* of assigning all loop levels to SW to moving all loop levels into the CU are generated as *candidates*. For example, in Figure 6.a, the variations of executing the nested loop1 and loop2 all on the CPU, executing the outer loop1 on the CPU and the inner loop2 as a CU, to executing both loops as CU, are created (shown in Figure 6.b).
- Inside each of these candidates, a graph of all execution *paths* through all of the C basic blocks is built. This graph also marks the nodes that are infeasible in a CU (non-inlineable function call, floating-point arithmetic). In Figure 7, such paths are shown in dashed lines.

Fig. 7. Execution frequency-based path selection



From these two data structures, the CU for the kernel is then assembled at the granularity of C basic blocks. Initially,

Application	# Kernels	# Candidates	# CUs	% Instructions executed on CUs
versatility	20	11	9	80
adpcm	1	1	1	80
g.721	10	7	2	33
capacity	22	20	5	40
pegwit	83	47	11	50

TABLE II
EFFECT OF PATH-BASED PARTITIONING

a path is selected leading from the entry into the kernel to the exit. This path will encompass the most frequently executed basic blocks (thicker lines in Figure 7). As long as there is still area available on the PLD, secondary paths are grown along this primary path, also favoring the more frequently executed blocks.

In this fashion, the partitioning algorithm tries to greedily move the largest possible part of the kernel onto the CU. Basic blocks that have been marked infeasible beforehand or that do not fit on the RF remain in SW.

At this stage, the final decision whether to actually implement the candidate on the RF has not been made. This occurs only after the communication costs have been considered (frequency of HW/SW switches and amount of data transferred), and the CU speedup over the CPU has been estimated. For simplicity, the latter is currently calculated by just summing the flat execution times of SW instructions and HW operators in cycles, taking into account the CPU and RF clock speeds.

Table II shows the effect of first picking candidates from all the kernels in the input program and then only realizing the most promising ones as actual CUs. In this fashion, between 33. . . 80% of the instructions executed by the programs can be accelerated by implementation in a CU.

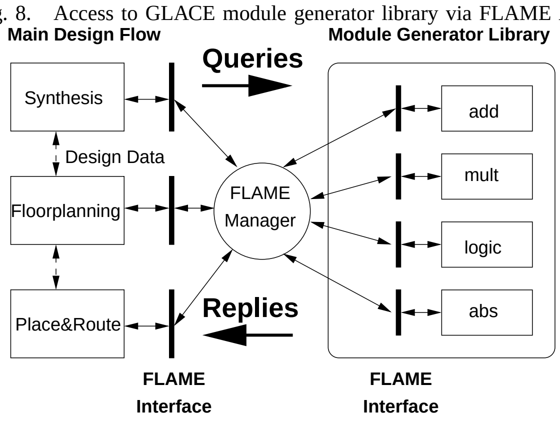
VI. HARDWARE SYNTHESIS

After the CUs destined for actual HW implementation have been selected from the candidates, the underlying logic is synthesized.

First, this step creates separate control and data flow graphs (CFG and DFG) in the static single-assignment form (SSA) [18] [17] from the intermediate representation of the C program. In this form, each variable is assigned to exactly once. When the C source program would require multiple assignments, these all occur to separate pseudo-variables that are then resolved into a single assignment by a so-called Φ function. This SSA form is not only useful for HW synthesis, but also allows the efficient realization of other optimization passes (e.g., constant propagation, strength reduction, etc. [18])

The operator nodes in the DFG are mapped to actual HW operators, composed from RF PEs, by accessing the GLACE [14] module library using the FLAME [15] interface. GLACE contains a wide variety of parametrizable module generators covering all C operators (arithmetic, logic, memory accesses) as well as system interfaces (dedicated I/O registers

Fig. 8. Access to GLACE module generator library via FLAME API



for communication with the CPU). The Φ nodes are mapped to multiplexers, also supplied by GLACE. In this fashion, the datapath portion of the CU is assembled. Figure 9 shows an example of such a datapath.

Fig. 9. Sample datapath with CPU and memory interfaces

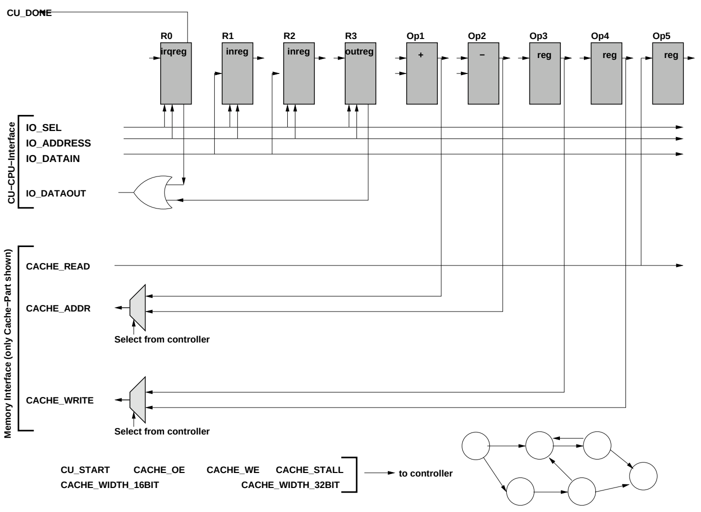
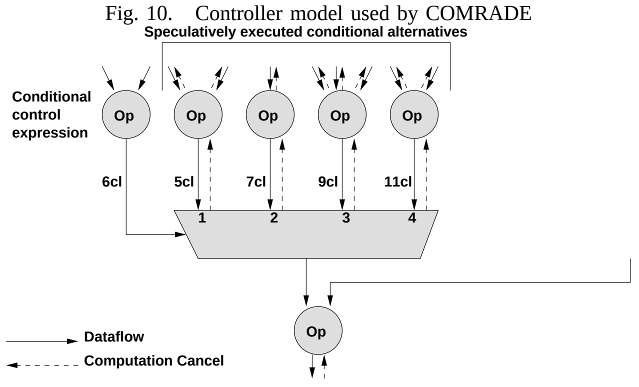


Fig. 10. Controller model used by COMRADE



Analogously, the CFG is converted into the controller part of the CU. Since it has a more irregular structure, the compiler creates it by using the encapsulated UCB SIS logic synthesis package [19].

Compared to many other efforts, the Petri net-based controller architecture used in COMRADE is significantly more powerful. It supports dynamic scheduling for variable-latency operators (e.g., cached memory accesses) as well as speculative execution. The latter means that all branches of a conditional are executed simultaneously. The final result is committed as soon as the control expression and the value of the corresponding branch is stable. Additionally, still executing computations on untaken branches can be canceled, discarding their data and immediately freeing their pipelines for a new set of input values.

In the example of Figure 10, the value of the conditional control expression is known after 6 clock cycles. Assuming the value of 2 for the control expression, the output of the entire conditional block is known after 7 clock cycles, the slower operators on inputs 3 and 4 no longer delay the calculation. These shortcut evaluations are fully exploited by the dynamic scheduling in the controller.

A potential disadvantage of this controller realization is that the number of HW registers (flip-flops) grows in the square of number of dependencies (both data and control) between HW operators. However, since the maximum number of operands of a C operator is three (for the ternary '?:'), the degree of the operator nodes is bounded.

Table III presents some characteristics of individual kernels in our benchmark suite. Specifically, it shows that both the size of the datapaths as well as that of the controller (despite the theoretical quadratic growth) are well manageable on the capacities of current PLDs.

Application / CU	#Datapath Ops.	# Controller FFs
versatility		
read_image.11	12	29
block_quantize.32	26	59
block_RLE_encode.46	31	79
block_RLE_encode.34	78	184
entropy_encode.14	7	23
hufenc.12	24	54
fcdf22.28	27	57
fcdf22.21	40	89
fcdf22.14	28	60
adpcm		
main.70	109	278
G.721		
update.126	12	39
update.119	42	126
capacity		
get_freq.16	10	34
printbin16.8	6	17
printVHDL.29	4	13
printVHDL.22	6	24
create_input_file.14	8	21
create_output_file.32	6	17
create_output_file.25	6	24
pegwit		
gfInit.52	11	37
gfInit.44	18	53
gfAdd.51	20	4
gfReduce.23	30	95
gfMultiply.86	4	81
gfMultiply.59	3	52
gfSmallDiv.38	29	84
gfAddMul.64	19	62
gfAddMul.52	29	86
gfAddMul.39	16	52

TABLE III
CU DATAPATH AND CONTROLLER AREA REQUIREMENTS

Appl.	RF Area	Unmerged	Optimal		Heuristic	
	[Cells]	#Reconf.	#Rc.	Time [s]	#Rc.	Time [s]
versatility	1920	4304	4	2.59	5	< 0.01
	5120	11171	2689	0.01	3586	0.01
capacity	1920	2016	2	< 0.01	2	< 0.01
	5120	1473	1	< 0.01	1	0.03
pegwit	1920	4056	1285	7.69	1286	$< 0.01s$
	5120	8706	1919	8.15s	4733	0.01

TABLE IV
OPTIMAL AND HEURISTICAL CONFIGURATION MERGING

VII. RECONFIGURATION SCHEDULING

With the long reconfiguration times of current PLD (compared to the system clock rate), proper scheduling of reconfigurations is crucial. In earlier systems such as Nimble [20], only a single CU was loaded onto the RF (realized by a Xilinx Virtex 1000 FPGA) at a time, using full reconfiguration. In most cases, the glacial reconfiguration speed of the PLD allowed only a single kernel to be executed as CU. Even a single switch between different CUs would have obliterated any possible speed-up.

Given the relatively small area requirements of our CUs (see previous Section) compared to modern PLD capacities, we can gainfully employ the configuration merging approach to put multiple CUs into a single configuration. Often, this is all that is required to completely hold the CUs of an entire application. However, for more complex applications, the set of CUs has to be partitioned into individual configurations. To further reduce configuration times, the same CU may occur in multiple different configurations (overlap).

We have developed two methods for actually merging CUs into configurations [21]. The first one relies on the CU execution order, described as a trace collected during a program run. This trace is then evaluated using a dynamic programming algorithm that computes the optimal packing for the minimal number of reconfigurations (shown in Table IV).

While the savings in reconfigurations are already significant (e.g., reducing the number from 4304 of a single-CU version down to 4 for merged CUs), it becomes apparent that our approach of greedily building the largest possible CUs in the partitioning step (Section V) leads to inefficiencies when the the reconfiguration behavior is factored in. This occurs, e.g., in the versatility application when targeting a larger RF, having 5120 instead 1920 PEs. With the larger size, more paths could be moved to the CUs. However, the larger CUs allow less merging and require more reconfigurations to switch between them (2689 instead of 4).

Thus, it would be more efficient to actually consider reconfiguration behavior in the partitioning step. To this end, we have also developed a heuristic that can estimate configuration schedules from the static program structure and flat execution counts, without the need to consult a possibly very long execution trace. This heuristic can also reduce the number of reconfigurations significantly (in some cases closely approaching the optimal solution), but generally requires only a fraction of the execution time (in extreme cases three orders of magnitude less). It is thus suitable for inclusion in the cost function of the path construction partitioning algorithm. This refinement is planned for the next revision of our partitioning phase.

VIII. CU Architecture

After assigning CUs to configurations, the complete HW is generated for the RF. Shown in Figure 9, this not only includes the datapath and controllers for each CU, but also the multiplexer network for quickly switching between different CUs. This multiplexer network connects to the central CPU-RF interface (called *wrapper*) that is shared between the CUs in the configuration.

This interface consists of two main components. First, the control part is used for system management functions (e.g., selecting the active CU, starting CU execution, indicating HW-SW switch to CPU) as well as CPU-CU data transfer. The latter is currently realized by mapping the dedicated I/O registers that were earlier synthesized as part of the DFG into the CPU address space. In a more tightly integrated system, this might be realized differently, e.g., by treating the I/O registers as shared CPU registers. Second, the wrapper also holds the memory interface. It allows the CUs access to the shared memory in master-mode, supported by caches for irregular access patterns and buffered streams for regular accesses. This infrastructure is realized using the Memory Architecture for Reconfigurable Computers (MARC) [22], which provides them in a technology-independent and configurable fashion. The latter means, that the number and kind of logical and physical ports realized can be matched to the needs of the application. The configuration shown in Figure 9, connecting two cached ports and one stream port to four RF-local memories and the shared memory (via a BIU bus interface unit) is just one example and could be varied by different parameters.

Currently under development is an automatic floorplanning tool that can automatically integrate the various components and layout them in a regular manner, attempting to keep corner turns of the wide databusses to a minimum. For access to operator topology and timing data, it also relies on the GLACE/FLAME system.

IX. IP Embedding

We realize, that despite our best efforts, the CUs generated by the compiler will not be able to match the efficiency of those manually implemented by expert designers. Thus, we are currently extending COMRADE to automatically embed such externally created IP blocks into the compiled CUs. For the programmer, this will occur as transparently as the calling of an optimized assembler subroutine from C.

The interfaces between the IP block and CU, including access to MARC, will be generated automatically from a parametrized meta-data description of the IP block. We have already defined a parameter catalog and methodology for this approach, and demonstrated its feasibility by manually integrating a Xilinx CoreGen-created FFT core with the CU wrapper [2].

X. Conclusion and Discussion

While the COMRADE flow is not fully functional yet, with the back-end step of the floorplanner still under development, the earlier phases and subsystems are already operational and capable of generating simulatable models of the CUs.

Even in this early stage, the flow has already advanced the state of the art over predecessors such as GarpCC [23] and Nimble [20]. These improvements include the generalized model of computation, the use of the same parametrized module library at all stages of the compilation process, and the merging-based configuration scheduling.

The flow is also ripe with opportunity for further optimization. Currently, work is actively proceeding in the following areas:

- Improving the bit-wise optimization by better analysis methods, recognizing even more constant and discarded bits.
- Extending the GLACE module generators to exploit constant and discarded bits in an operator-dependent manner. E.g., breaking the carry chains of adders and subtractors in the presence of successive zero bits in operands.
- Adding more loop transformations, leading to more parallelism in the DFG. An appropriate analysis framework based on OMEGA [24] has already been added to COMRADE and the first transformation (scalarization) is already functional.
- Replacement of the current discrete ACS target platform with a chip-level solution (VirtexIIpro-based).

When the COMRADE flow is operational down to actual bitstream generation, it will be very interesting to observe the effect of these transformations on the system-level performance.

References

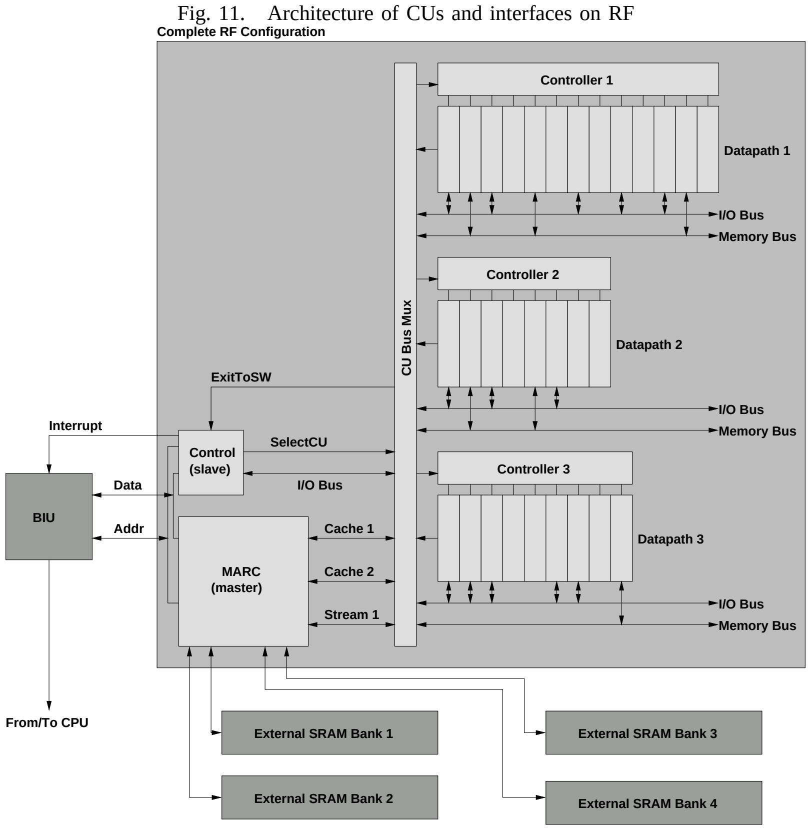
[1] Koch A. Golze, U., "A Comprehensive Prototyping Platform for Hardware-Software Codesign", *Proc. Workshop on Rapid Systems Prototyping*, Paris (F), 2000

[2] Lange H., Koch A., "Hardware/Software-Codesign by Automatic Embedding of Complex IP Cores", *Proc. Intl. Conf. on Field-Programmable Logic (FPL)*, Antwerp (BE), 2004

[3] Yu C.W. et al., "A Smith-Waterman Systolic Cell", *Proc. Intl. Conf. on Field-Programmable Logic (FPL)*, Lisbon (PT), 2003

[4] Jarvinen K.U. et al., "A Fully Pipelined Memoryless 17.8 Gbps AES-128 Encryptor", *Intl. Symp. on Field-Programmable Gate Arrays (FPGA)*, Monterey (CA, USA), February 2003

Fig. 11. Architecture of CUs and interfaces on RF



[5] Rabaey J., "Pleiades: Ultra Low Power Hybrid and Reconfigurable Computing", *Presentation at Berkeley Wireless Research Center Retreat*, June 1999

[6] Gädke H., Koch A., "Wavelet-based Image Compression on the Reconfigurable Computer ACE-V", *Proc. Intl. Conf. on Field-Programmable Logic (FPL)*, Antwerp (BE), September 2004

[7] Kumar S., Pires L., Ponnuswamy S. et al., "A Benchmark Suite for Evaluating Configurable Computing Systems - Status, Reflections, and Future Directions", *Proc. Eighth International Symposium on Field-Programmable Gate Arrays (FPGA)*, Monterey (USA), 2000

[8] Barwood G., `http://groups.yahoo.com/group/pegwit/`

[9] `www.tensilica.com`

[10] `www.arc.com`

[11] `www.arm.com`

[12] Blodget B., Bobda C., Huebner M., Niyonkuru A. "Partial and Dynamically Reconfiguration of Xilinx Virtex-II FPGAs", *Proc. Intl. Conf. on Field-Programmable Logic (FPL)*, Antwerp (BE), 2004

[13] Bobda C., "ESM: The Erlangen Slot Machine - Architecture and Development Tools", *Proc. Design Automation and Test in Europe (DATE) Conference*, Munich (DE), 2004

[14] Neumann T., Koch A., "A Generic Library for Adaptive Computing Environments" *Proc. Workshop on Field-Programmable Logic (FPL)*, Belfast (UK), 2001

[15] Koch A., "On Tool Integration in High-Performance FPGA Design Flows", *Proc. Intl. Workshop on Field-Programmable Logic (FPL)*, Glasgow (UK), 1999

[16] Kasprzyk N., Koch A., "Verbesserte Hardware-Software-Partitionierung für Adaptive Computer", *Proc. Conf. on Architecture of Computing Systems (ARCS)*, Augsburg (DE), 2004

[17] Kasprzyk N., Koch A., Golze U., Rock M., "An Improved Intermediate Representation for Datapath Generation", *Proc. Intl. Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, Las Vegas (USA), 2003

[18] Muchnik S. S., "Advanced Compiler Design and Implementation", *Morgan Kaufmann Publishers*, San Francisco (USA), 1997

[19] Sentovich, E.M. et al., "SIS: A System for Sequential Circuit Synthesis", *Electr. Res. Lab. Memo No. UCB/ERL M92/41*, Dept. of EE and CS, UC Berkeley (USA), 1992

[20] MacMillen D., "Nimble Compiler Environment for Agile Hardware, Vol. 1+2", *Report AFRL-IF-WP-TR-2002-1536*, Information Directorate, Air Force Research Laboratory, USA, 2001

[21] Kasprzyk N., van der Veen J. C., Koch A., "Configuration Merging for Adaptive Computer Applications", *Proc. Intl. Workshop on Field-Programmable Logic (FPL)*, Tampere (FI), 2005

[22] Lange H., Koch A. "Memory Access Schemes for Configurable Processors", *Proc. Intl. Workshop on Field-Programmable Logic and Applications (FPL)*, Villach (AT), 2000

[23] Callahan T., Hauser J., Wawrzynek J., "The Garp Architecture and C Compiler", *IEEE Computer*, Vol. 33, No. 4, April 2000

[24] Pugh W., "Uniform techniques for loop optimization", *Proc. Intl. Conf. on Supercomputing*, 1992