Hardware/Software Co-Compilation with the Nymble System

Jens Huthmann*, Björn Liebig*, Julian Oppermann†, Andreas Koch*
*Embedded Systems and Applications Group (ESA), Technische Universität Darmstadt
Email: {jh,bl,ahk}@esa.cs.tu-darmstadt.de
†Center for Advanced Security Research Darmstadt (CASED), Technische Universität Darmstadt
Email: julian.oppermann@cased.de

***Abstract*—The Nymble compiler system accepts C code, annotated by the user with partitioning directives, and translates the indicated parts into hardware accelerators for execution on FPGA-based reconfigurable computers. The interface logic between the remaining software parts and the accelerators is automatically created, taking into account details such as cache flushes and copying of FPGA-local memories to the shared main memory. The system also supports calls from hardware back into software, both for infrequent operations that do not merit hardware area, as well as for using operating system / library services such as memory management and I/O.**

I. Introduction

Adaptive computing systems (ACS) can improve the performance of many algorithms by combining a standard software-programmable processor (SPP) with a reconfigurable compute unit (RCU). Computing kernels are realized as dedicated microarchitectures on the RCU, while the SPP implements just non-performance critical control operations or operations that cannot be mapped efficiently to the RCU.

However, due to the specialized expertise required from the developer for an ACS, practical use of ACSs is still relegated to niche cases: Programming an ACS commonly requires experience in digital hardware design, computer architecture, and specialized programming languages and tool flows (Verilog/VHDL, place and route, simulation, ...). The skills will be unfamiliar to most software developers.

To open the potential of ACSs up to more users, considerable research effort has been invested in automatic compilers for such heterogeneous computers. Often, these tools are restricted to describing and generating the actual hardware accelerators. The software-hardware interfaces as well as the low-level communication mechanisms must be explicitly managed by the user. True hardware/software co-design tools exist, but they are often limited to a coarse partitioning granularity (often at the level of entire functions).

As an alternative, we present our hardware/software co-compiler system Nymble. Nymble aims not only to compile a large subset of C to hardware (including, e.g., pointer operations and irregular control flow in loops), but also automatically create the required software/hardware interfaces. In addition, Nymble supports advanced features such as pipelining execution even in the presence of nested loops and automatic hardware-to-software calls for rarely executed or difficult to accelerate operations.

II. Related Work

A growing number of design tools can translate (often differing) subsets of C into synthesizable HDL code [1], [2], [3]. However, co-compilation of C descriptions into *hybrid* hardware/software executables mainly remains the subject of academic research.

A. COMRADE

COMRADE 2.0 [4] was developed in the SUIF2 compiler framework [5]. It focused on the compilation of control-intensive C code into dynamically scheduled accelerators, providing support for aggressive speculative execution (e.g., early cancellation of mispredicted operators). The proposed compute model included finely granular hardware/software partitioning (at the level of loops), but these features were not robustly implemented. COMRADE did support shared-memory operation between the software-programmable processor (SPP) and the reconfigurable compute unit (RCU), with the latter being able to autonomously perform main memory accesses. Cache coherency was handled by a combined hardware/software approach: The RCU memory system tracked dirty cache lines, which were then explicitly invalidated in the SPP cache under software control. Small arrays were localized into on-chip memories, which could be moved back to SPP-accessible memory as specified by the programmer using a Local Paging Unit (LPU) on the RCU [6].

B. ROCCC

Like Nymble, ROCCC is a C to hardware compiler based on the LLVM compiler framework. In [7], ROCCC-generated hardware shows similar performance and 15x higher productivity relative to hand-written VHDL. However, ROCCC lacks support for many commonly used C constructs, such as pointers, loops other than for, variable-distance shifts, and targets pure hardware instead of performing hybrid hardware/software synthesis.

C. LegUp

LegUp [8] is an open-source high-level synthesis system. Introduced in 2011, it has reached its third major version release at the beginning of 2013 [9]. LegUp is similar to our approach as it supports automated partitioning of a C program (based on programmer directives) into a mixed software/accelerator setup and is also based on the LLVM framework. However, the architectures of LegUp and Nymble differ on multiple levels.

Both LegUp and Nymble support hardware/software co-execution with different SPPs. However, LegUp assumes separate memory spaces for SPP and RCU, requiring explicit copying of data and incompatible pointers (e.g., for PCIe-based systems), while the Nymble infrastructure allows fully shared

memories and the transparent exchange of virtual addresses between SPP and RCU.

Nymble offers a finer partitioning granularity, as it can generate hardware for single-entry regions of a function, as opposed to the whole-function granularity used in LegUp. Furthermore, Nymble supports calling software functions from within hardware kernels, preserving the current accelerator state.

In LegUp, control flow is modeled by an FSM whose states correspond to the basic blocks of the function. The instructions contained in each basic block form a dataflow graph that is scheduled independently from other blocks. In Nymble, the basic block structure is eliminated in favor of a combined control-dataflow graph (CDFG) for each natural loop nesting level. Scheduling at this per-loop scope potentially exposes more instruction-level parallelism (e.g., computing control conditions in parallel to speculative data operations).

Both compilers offer loop pipelining using an iterative modulo scheduler. LegUp's implementation can only operate on loops comprised of one basic block, while Nymble's is more general and can handle loops with internal control flow.

The LegUp memory system makes extensive use of the on-chip RAM blocks for storing local variables, but is limited to one concurrent memory access per accelerator. Nymble benefits from the MARC II configurable memory system [10] that allows multiple concurrent read/write accesses (with configurable coherency between them), and can relocate scalar *global* variables to local hardware registers for increased performance. Furthermore, scalar *local* variables are eliminated via exhaustive inlining and LLVM's memory-to-register promotion pass. In Nymble, arrays can be held in on-chip memories in an automated fashion: They are manually indicated by the programmer in the software part, and then automatically mapped into the shared virtual address space of SPP and RCU.

D. Garp CC

The Garp CC compiler [11] provides automatic partitioning of C programs to run on a MIPS processor augmented with a coarse-grained reconfigurable array (CGRA, having 32b arithmetic/logic operators as primitives). It is based on the SUIF compiler infrastructure [5] and uses a modified GCC toolchain. Garp CC attempts to perform the partitioning without user intervention by considering all loops as hardware kernels, and then incrementally excluding paths that are not suitable or beneficial for calculation on the reconfigurable hardware. Excluded paths would then be executed in software, either directly or by performing a hardware-to-software execution switch. These switches had a coarser granularity than in Nymble, since a switch back into hardware execution could occur only into the closest loop header, not back into the body of the loop. The intermediate representation used in Nymble is similar to Garp CC's dataflow graph. Garp CC also employs a shared-memory architecture, with the SPP and RCU even sharing the data caches. The underlying CGRA can have three incomplete memory operations in-flight, but can issue only one memory access per cycle due to only a single address bus being present in the fabric. As discussed above, Nymble's MARC II memory interface allows more parallel memory accesses to its distributed caches.

III. Nymble Compiler

Nymble is both a tool for the co-compilation of C code into hybrid hardware/software applications for execution on an Adaptive Computing System (ACS), as well as a framework for research on application-specific microarchitectures. An example for the latter is PreCoRe [12], [13], [14], the automatic synthesis of application-specific hardware for data-value speculation. Nymble stands in the tradition of Garp CC [11] and Nimble [15] for generating (mostly) statically scheduled microarchitectures using shared-memory communication between SPP and RCU, but is significantly more advanced than both, e.g., with regard to parallel memory accesses and fine granularity hardware/software partitioning.

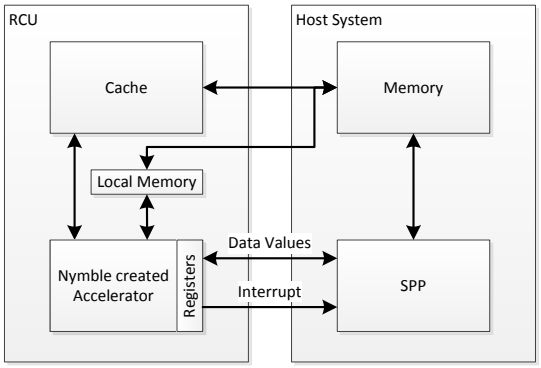


Fig. 1. Adaptive Computing System (ACS)

A. Hardware/Software Co-Execution Model

Nymble's hardware/software co-execution model requires a tight coupling between SPP and RCU. Communication must be possible in low-latency as well as high-bandwidth modes, but is not required to support both modes in parallel. The low-latency mode is used to transfer small-volume live variables and parameters as memory mapped registers for software/hardware and hardware/software execution switches. The high-bandwidth mode is for bulk data exchange and realized as high-performance shared memory. The model and its architectural implications is discussed in greater detail in [16].

B. ACS Architecture

Adaptive Computing Systems (ACS) provide the capabilities required for Nymble's co-execution model. Ideally, they would be realized in a fashion similar to Garp [11], which had common caches between the SPP and RCU. More common, however, are architectures as shown in Figure 1, with only the main memory actually being shared. However, current developments in reconfigurable systems-on-chip, such as the Xilinx Zynq-7000 series of devices [17], indicate a trend towards tighter integration. While Zynq-7000 does not yet support fully bidirectional cache-coherency between the SPP(s) and RCU, the required extensions to the AXI bus protocol have already been specified by ARM [18] and are expected to be implemented in future reconfigurable devices.

C. Concrete ACS Implementations

Such an ACS architecture can be implemented in many different fashions. As an example, Figure 2 shows how the Xilinx ML507 prototyping board with its Virtex 5 FX FPGA can be used to realize an ACS. While the embedded superscalar PowerPC 440 SPP cores of the FX-family device do not allow shared caches between SPP and RCU, shared main memory can be implemented very efficiently: The RCU has

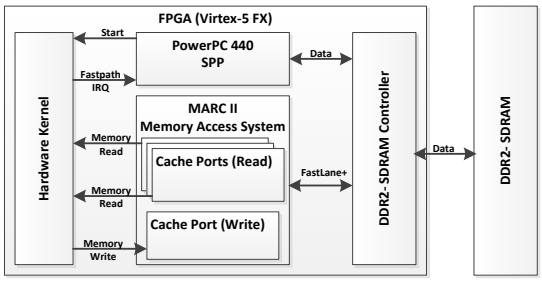


Fig. 2. ACS based on the Xilinx ML507 Virtex 5 FX FPGA Board

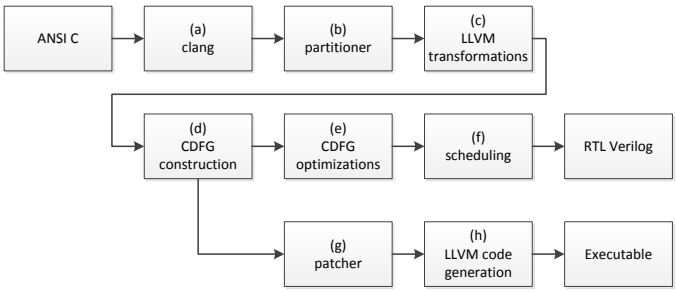


Fig. 3. Compile Flow

a dedicated FastLane+ high-bandwidth interface to the DDR2-SDRAM controller, which is considerably more efficient than the stock PLB-based memory interface commonly used on V5FX FPGAs.

In conjunction with other hardware measures and a customized version of a full-scale Linux operating system, RCU-SPP signaling latency using our FastPath kernel modifications has been improved by up to 23x, even over a Linux kernel extended with real-time patches. Furthermore, supported by our AISLE extensions, the RCU can access the shared memory in the same virtual address space as software on the SPP. All of these enhancements are described in greater detail in [16].

A hardware kernel executing on the RCU usually does not connect directly to the raw FastLane+ memory interface. Instead, the MARC II [10] memory system is inserted between the kernel and FastLane+ (Figure 2) to provide multiple caching/streaming memory ports with configurable coherency and organization (e.g., stream buffer size, cache lines and line length, etc.).

D. Compile Flow

TABLE I. LLVM TRANSFORMS/ANALYSIS PASSES USED IN NYMBLE[19]

Name	Description
-simplifycfg	Removes dead or unnecessary basic blocks.
-lowerswitch	Transforms switch instructions to a sequence of branches.
-loop-simplify	Guarantees that natural loops have a preheader block, their header block dominates all loop exits, and they have exactly one backedge.
-sccp	Sparse conditional constant propagation.
-instcombine	Algebraic simplifications.
-dce	Dead code elimination.
-mergereturn	Transform function to have at most one return instruction.
-basicaa, -scev-aa	Alias information for load and store instructions based on program independent facts and scalar evolution analysis.
LoopInfo	Mapping of basic blocks to natural loops.
DominatorTree	Dominance relation for basic blocks.

Nymble has been under continuous development for more than four years. Until recently, it relied on the Scale compiler framework [20] for its front- and middle-end operations. It has now been moved successfully to the LLVM infrastructure [21] for machine-independent passes in the compile flow (shown in Figure 3).

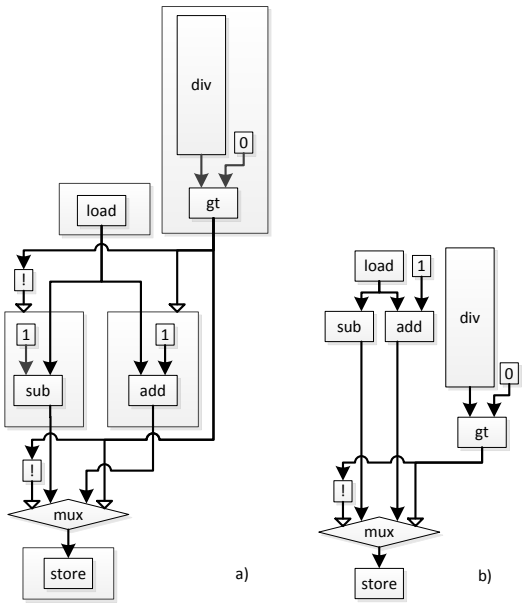
We modified the clang C/C++ front-end (Fig. 3.a) to accept custom pragma directives. clang translates input programs annotated by these pragmas to the LLVM intermediate representation (IR). The pragmas are then interpreted to define the hardware/software partitioning (Fig. 3.b), extract the future hardware part into a separate function, and guide the interface synthesis (see Sec. III-F1). Next, a series of LLVM optimization passes (Fig. 3.c, Table I) are applied in order to simplify and normalize to the hardware function. From its IR, the hierarchical CDFG for the actual hardware microarchitecture is constructed (Fig. 3.d, detailed in Sec. III-G), and optimized for hardware synthesis using non-LLVM passes (Fig. 3.e, described in Sec. III-H). As shown in Fig. 3.f, the optimized CDFG is scheduled using either ASAP or modulo scheduling [22], and exported as RTL Verilog for logic synthesis. Independently from this high-level hardware synthesis, the software parts of the program are patched with the hardware/software interface operations to realize our co-execution model (see Sec. III-F3). The completed software parts are then compiled to an executable using the traditional LLVM software flow (Fig. 3.h).

E. Nymble Microarchitecture Template

After our experiments in COMRADE with the fully dynamic COCOMA scheduling model, we wanted to explore the alternate use of a simpler, mostly statically scheduled model for code with simple control flow and few (if any) variable-latency operators (VLO).

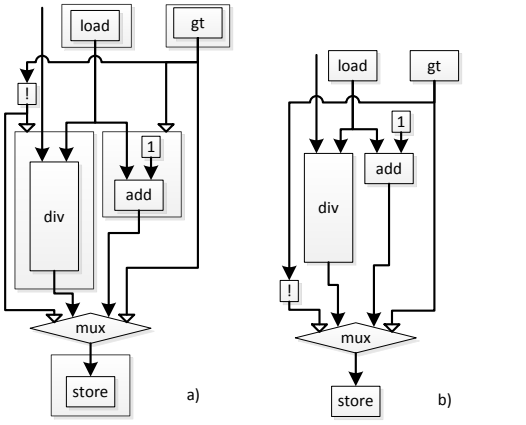
When designing an IR for the high-level hardware synthesis, two choices are commonly explored for the representation of data- and control flow as well as hierarchically nested control structures. One alternative transforms the operations in each basic block into separate DFGs (shown in Fig. 4.a and Fig. 5.a), and expresses the control flow using an external FSM triggering/enabling each basic block DFG (this is the approach used by LegUp). A second choice flattens the basic blocks and their control flow into a single CDFG (shown in the .b subfigures, respectively).

The CDFG-based approach allows for potentially higher parallelism, as the individual DFGs may (at most) execute the operations of a single basic block in parallel. An example of this situation is shown in Fig. 4, where the CDFG allows the speculative side-effect free computation of both of the conditional branches in parallel with the computation of the control condition (the longer-latency division). When using separate DFGs, the division has to complete before the relevant branch can be executed (with side-effects, if necessary). On the other hand, if the branches have different latencies, the CDFG-based approach will need to always consider the longest one as the latency of the conditional. Such a scenario is shown in Fig. 5. Here, the separated DFGs are advantageous, since the conditional will be executed non-speculatively and only the DFG of the selected branch will determine the total latency. As Nymble is targeted specifically at the generation of efficient hardware for code with simple control flow, we chose the CDFG-based approach to expose more parallelism. For code with more




```
for(i=0;i < n;i++) {
  x = a[i];
  if(i / 3 > 0)
    x = x + 1;
  else
    x = x - 1;
  b[i] = x;
}
```

Fig. 4. Separate DFGs vs. single CDFG: Advantage CDFG




```
for(i=0;i < n;i++) {
  x = a[i];
  if(i>0)
    x = x + 1;
  else
    x = x / c;
  b[i] = x;
}
```

Fig. 5. Separate DFGs vs. single CDFG: Advantage DFGs

complex control (or wildly differing branch latencies), we can always fall back to COCOMA to schedule it dynamically. This is potentially even more powerful than separate DFGs, since COCOMA does support speculative execution of conditionals with *different* latencies, as well as early aborts of misspeculated branches. However, these capabilities induce a significantly higher hardware overhead.

Since we always incur the latency of the longest path through a conditional, we can easily extend our microarchitecture to fully pipelined execution by balancing the branch latencies (inserting register stages into the shorter branch). However, this approach becomes more complicated when extended to loops, as now all loop back-edges in the CDFG have to be balanced to the longest path through the loop body. The length of this path determines the number of cycles between successive iterations, also called the Initiation Interval (II) of the loop. We could keep extending this straightforward balancing approach to nested loops, but would then be faced with extremely large IIs (longest latency path through all nested loop bodies), which would quickly obliterate any performance gain we could hope for through pipelining. In order to avoid this effect, we do not fully flatten the operations of our hardware kernel into a single CDFG, but generate a hierarchy based on the nested loop structure: Each loop will be turned into a separate CDFG, with innermost loops executing in a fully pipelined fashion, while outer loops are stalled until inner loops complete. Note that this approach is amenable to transformations such as loop fusion to reduce the number of loops while increasing the number of operations within loops. The latter could then profit from the increased parallelism enabled by our per-loop CDFGs.

Our microarchitecture is not completely statically scheduled. Since we need to employ caches for efficient access to the shared main memory, we have to handle VLOs. To this end, we schedule memory operations for their expected latency (which varies for the different ACS platforms), and *stall* the entire computation if we detect violated latency constraints at run-time (e.g., due to a cache miss). A small dynamically scheduled controller, separate from the control FSM for the statically scheduled CDFG, is responsible for handling these occurrences. It is also able to deal correctly with multiple parallel latency constraint violations (e.g., separate cache misses for parallel memory accesses). The same mechanism is used to support VLOs as well as software service calls from hardware (see Sec. III-F2).

F. Interface Synthesis

Interface synthesis deals with both software-to-hardware as well as hardware-to-software interfaces.

1) Software-to-Hardware Interface: Using the `#pragma HARDWARE on/off` directives, which we added to the clang front-end (see Sec.III-D), we can mark arbitrary single-entry regions within a function for execution as hardware accelerators. The pragmas result in the creation of special marker instructions at the region entry and all of its exits when the clang generates LLVM-IR for the middle-end.

In the partitioning pass, we traverse the CFG in reverse postorder, and collect the basic blocks beginning from the entry marker until we reach an exit marker. If the markers are not at the beginning of a basic block, their block is split accordingly. Calls to other functions from within the marked blocks are

```
int func(int op, char *X, int N) {
  int j;
#pragma HARDWARE on
  for (j = 0; j < N; j++) {
    char tmp = X[j];
    if (op)
      tmp++;
    else
      tmp--;
    X[j] = tmp;
  }
#pragma HARDWARE off
  return j;
}
```

Fig. 6. Sample function with partitioning directives

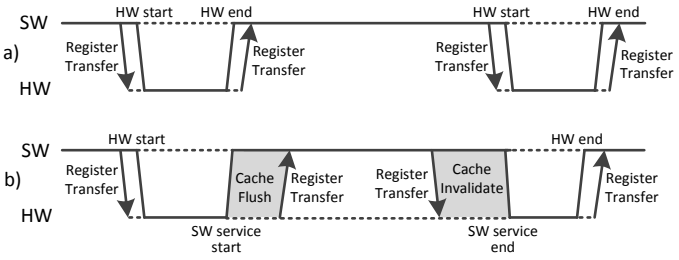
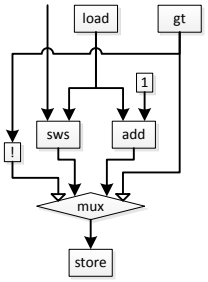


Fig. 7. Accelerator invocation protocol (solid line signifies control, dotted line is stored state while suspended)




```
for(i=0;i < n;i++) {
  x = a[i];
  if(i>0)
    x = x + 1;
  else
    #pragma SOFTWARESERVICE on
    x = x / c;
    #pragma SOFTWARESERVICE off
  b[i] = x;
}
```

Fig. 8. Software service for resolving the CDFG disadvantage of Fig. 5

exhaustively inlined [1].

We then leverage LLVM's CodeExtractor utility class to extract these blocks into a new *hardware function*. The SSA property in the LLVM-IR makes it easy to find values that cross the boundaries of the hardware part. Values originating from outside the accelerator are passed as arguments to the hardware function. Values that have uses outside of the accelerator become "out" arguments that point to a stack slot written inside of the hardware function.

Applied to the example function in Fig. 6, the CodeExtractor will determine that the original function's arguments are used in the hardware accelerator, and that the value `%j.0` (resulting from the source variable `j`) is alive after leaving the accelerator. The hardware function (shown in Figure 9) therefore has the signature `func_hw(i32 %op, i8 %X, i32 %N, i32 %j.0.out)`, with the last argument being an out argument.

2) Hardware-to-Software Interface: Our co-execution model supports the call of software functions from the hardware accelerator (so-called *software services*). This can be used to allow software co-execution for calls to functions that cannot or should not be inlined into the hardware accelerator. Another beneficial scenario is to replace long, but seldom used operations with a software service, aiming to keep the initiation interval of the *common case* short in the accelerator. Fig. 8 shows this approach for speed-up the common case of 5.b. If the accelerator and software service exchange data only explicitly via passed variables (instead of arbitrary shared memory locations), such calls have only very low overhead, as the accelerator cache(s) need not be flushed to main memory (see Fig. 7.a).

The extraction of a *software service function* is similar to the extraction of the hardware function surrounding it: We introduced the directives `#pragma SOFTWARESERVICE on/off` in clang to declare which parts of the hardware function should execute in software. Again, the function arguments represent the values that are alive across the HW/SW boundaries. From a scheduling perspective, the software services are considered to be VLOs and halt the accelerator until control returns to hardware (see Sec. III-E).

3) Accelerator Invocation Protocol: The normal control flow between software and hardware is shown in Figure 7.a. A dotted line denotes that, while the control has switched from the software or hardware, the state of the registers is maintained, so that execution can continue later. The hardware function arguments and the addresses of global variables used in the accelerator are passed in via registers (one for each argument/address), which is denoted by an arrow in the example.

At `HW_start`, control is transferred from the software to the hardware, while the software maintains its current state. Once the accelerator has finished, it raises an interrupt[2], denoted by `HW_end`. The values of out-arguments have been set by the accelerator and are read from the associated registers. Afterwards, the state of the hardware is reset to a known initial state.

In the presence of software services, the protocol is extended as follows (see Fig. 7 b)): For a hardware-to-software call, the accelerator writes the arguments of the software service to registers. The hardware execution is halted and its state is preserved. If necessary, a cache flush is issued (if the called service cannot be proven to be free of memory reads). The ID of the requested software service is passed along with the interrupt. Software then reads the appropriate arguments from the registers, and the service function is executed. On its return, the service's out-arguments are written to accelerator registers. If the service cannot be proven to free of memory writes, the accelerator's cache(s) are invalidated and hardware execution resumes.

G. Transforming LLVM IR to CDFGs

LLVM IR is an assembly-like intermediate representation in static single assignment (SSA) form, organized as basic blocks that form a CFG. Our CDFG is comprised of operation nodes and edges modeling the dataflow between them. All control flow is converted to conditional dataflow and predicated operations, making our IR similar to the one used in the Garp CC compiler [11]. In this section, we use the code of Fig. 9 and

[1] If a call cannot be inlined, we declare the call site as a hardware-to-software call (see Sec. III-F2)

[2] Note that we have significantly reduced IRQ processing latency in our ACS [16].

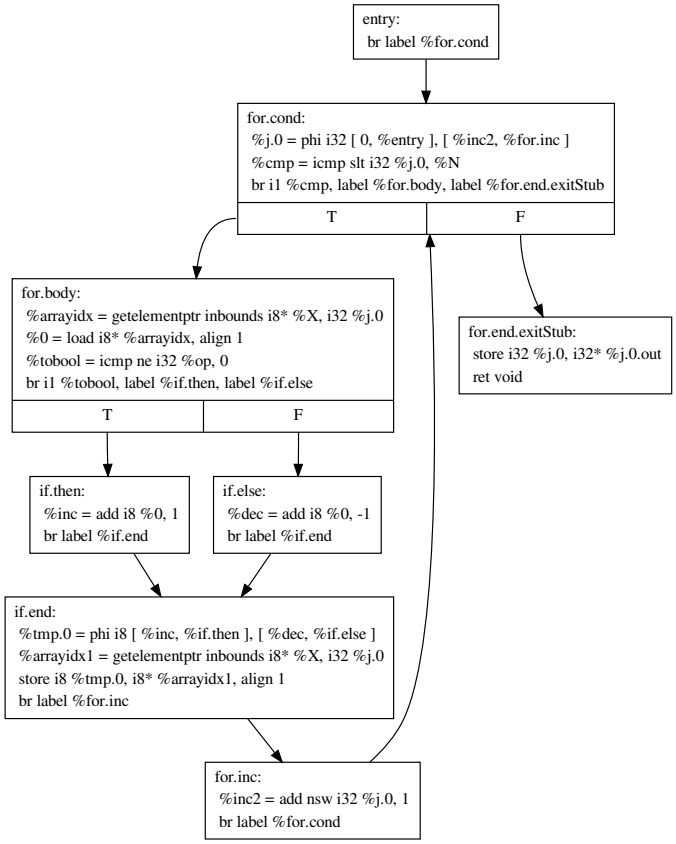


Fig. 9. CFG of the hardware function from Fig. 6

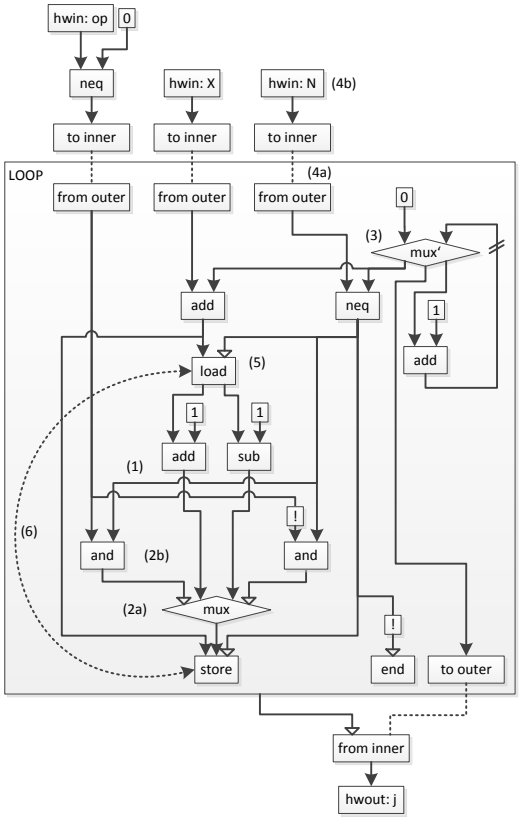


Fig. 10. CDFG of the hardware function from Fig. 6

its resulting CDFG in Fig. 10 as example for the transformation process.

We construct a graph for each natural loop (as determined by LLVM's loop analysis), beginning with the most deeply nested loops.

The first step is to perform a simple dataflow analysis on the basic blocks. For each basic block, we calculate the condition that controls its execution in the original program. Such a condition is a propositional formula whose elements are the condition values used by branch instructions. In the example, we would, e.g., determine **`true`** for the entry block, **`%cmp`** as the condition for the block **`for.body`**, and **`%cmp`** $\wedge\neg$ **`%tobool`** for **`if.else`**.

Then, all LLVM values and instructions are translated to the appropriate operations in the CDFG while iterating over the basic blocks of the current loop. The mapping of LLVM values to CDFG operations is stored in a symbol table per graph. Beyond nodes for common arithmetic and load/store operations (1), our IR contains several special nodes.

Multiplexers are constructed for SSA *Phi* instructions and come in two flavors: The first one is denoted by mux and is used to model conditional dataflow resulting from branches in the original program (2a). These operations have pairs of inputs, specifically a data input and a one-hot predicate input. Upon activation, the mux operation forwards the single data input whose predicate input is true. Additional operations that calculate the predicate value are constructed based on the analyzed condition of the predecessor block referred to in the phi instruction (2b). Note that in the example, the "**`!`**" symbol represents a logical negation.

The second kind of multiplexer is used to model the flow of values across loop iterations and is labeled as mux' in the example (3). The left input takes an initial value in the first iteration. The right input is passed through the operation in all subsequent iterations. This kind of node is also known as the "θ" operation in dataflow representations [23]. In the example CDFG, the two diagonal dashes indicate the boundary between loop iterations.

Loop and *SoftwareService* nodes represent nested graphs. Pairs of transfer nodes model the exchange of values from and to nested graphs (4a), and into/out-of the accelerator (4b). If a value contained in a different loop is used, the operation is looked up from the corresponding graph's symbol table and connected by a pair of these transfer nodes.

The stateful operations (load, store, loop, software service) have an additional predicate input that controls their execution (5). The operations calculating the predicate value are constructed based on the analyzed condition of the containing block or the loop's header block, respectively. Special memory dependency edges are inserted between load and store operations and nested loop nodes in case we cannot prove that they are independent by means of LLVM's alias analysis (6).

As our execution model facilitates loop pipelining, two issues must be addressed: First, memory dependencies are bidirectional. For the example, this implies that the read operation cannot be scheduled before the store operation of the previous iteration (inter-iteration RAW dependency). Second, we support unstructured control statements in the original C program, such as a **`return`** from within a nested loop. This extends predicates at the current nesting level to include condition values from *inner* loops, which are exchanged analogously to data values by inter-loop transfer nodes. Similar care must

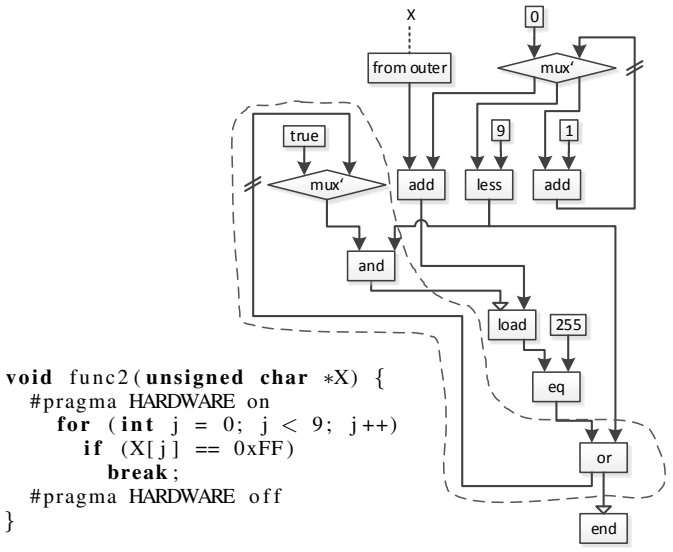


Fig. 11. Sample code and CDFG of a **`for`** loop requiring guard conditions.

be taken for loops with data-dependent exit conditions, such as the example shown in Fig. 11: We have to ensure that the stateful load of **`X[j]`** is only executed if the *previous* iteration did not abort the loop due to **`X[j] == 0xFF`**. To this end, execution of the load node is conditional not only on the current iteration's loop predicate, but also on the previous iteration's exit condition (shown in the dashed area of the figure).

H. Hardware-Specific Optimizations

In addition to the machine-independent optimizations performed by LLVM, we apply a number of hardware-specific optimizations in the Nymble back-end:

1) Operator Chaining: Simple operations (e.g., adds, subtracts, shifts, etc.) are so short that multiple of them may be *chained* within a single clock cycle. However, since this optimization relies on timing estimates for the different operators, overly ambitious chaining may lead to a slow-down during actual design implementation (map, place, route).

2) Memory Port Multiplexing: While the MARC II memory system (see Sec. III-C) supports distributed parallel caches, actually using a separate cache for each individual static access site in the program would be excessive. The number of actual caches is limited by two measures: First, since only a single of the hierarchical CDFGs for nested loops can be active at the same time, distributed caches are shared among different CDFGs. Second, a larger CDFG may contain so many access sites that it is no longer economical to provide each of them with a separate cache. As an area-performance trade-off, we can set an upper bound on the number of distributed caches (specific for the current target platform) and re-use caches for multiple access sites within a single CDFG. This is transparent to the datapath, as each access site is treated as a VLO and dynamically scheduled (see Sec. III-E). The datapath is restarted only if *all* required accesses have actually been completed, regardless of their static scheduling.

3) Local Memory: In general, Nymble memory accesses are cached to external DDR1/2/3-SDRAM memory (see Sec. III-C). However, the system also supports automation for directly using the on-chip BlockRAMs. This not only allows further speed-ups, but also reduces area (since BlockRAMs do not require the caching/coherency mechanisms). To this end, the user can mark arrays for BlockRAM storage. The compiler will then automatically create BlockRAM read and write operations instead of cache ports for each array access. Furthermore, since BlockRAMs on the target devices (currently Xilinx Virtex 5 and 6) have only two ports, these must be time-multiplexed if a CDFG has more accesses to the array. The required multiplexing and scheduling logic will be automatically created by the compiler.

Pointers present a more difficult problem: Once static LLVM alias/"points-to" analysis has determined that a pointer points only to a single BlockRAM-stored array, it will be mapped to that BlockRAM too, using either an exclusive or shared BlockRAM memory port. If a pointer points to multiple BlockRAM-stored arrays, all of these arrays will be stored in the same BlockRAM (start addresses are automatically assigned by the compiler). Finally, if a pointer can point both to main memory and to arrays for which on-chip storage has been requested, the latter request is ignored and the array will be held in main memory.

Note that the automation provided by Nymble extends to localizing arrays that are shared between SPP and RCU: Once Nymble determines that arrays are live between software and hardware, specialized data-movement units are synthesized by the compiler which, when the entering and exiting of the accelerator, copy the contents of the local arrays from/to the main memory shared with the SPP.

IV. Experimental Evaluation

For evaluating the Nymble system, we examined the CHStone benchmark suite [24]. We used the architecture of the Xilinx ML507 prototyping board as the base of our ACS (especially the memory system and SPP-RCU interface). However, the results presented here were determined from *post-layout simulations*, as the Virtex 5 FX70 device present on the ML507 was too small for some of the larger benchmarks. Our "virtual" ACS substitutes the larger FX200 FPGA, which can easily hold all of the generated circuits. We performed true hardware/software co-simulation, with the software parts of the applications executing natively on the x86 simulation host and communicating with the simulated RCU via shared memory. The simulations do include exact cycle-accurate models for memories and cache behavior.

TABLE II. Hardware partitions in the benchmark programs

Benchmark	HW pragmas around...	in loop
dfadd/mul/div	float64_add/mul/div	yes
mips	do-while loop in main	no
adpcm	adpcm_main	no
gsm	Gsm_LPC_Analysis	no
motion	motion_vectors	no
blowfish	BF_cfb64_encrypt	yes
sha	sha_stream	no

Pragma directives bracket the call sites of the actual benchmark routines, leaving initialization and the self test code to be executed in software where possible (see Table II). In the cases where the hardware region is in a loop, we accumulated the cycle counts of all invocations including the respective cache flushing phases.

Table III shows the cycle counts for accelerator execution, clock frequencies (post-placement), area requirements,

TABLE III. BENCHMARK RESULTS ON VIRTEX 5

	Baseline						Chaining					
Benchmark	**Cycles**	**Freq. [MHz]**	**Slices**	**BRAM**	**RP**	**muxed**	**Cycles**	**Freq. [MHz]**	**Slices**	**BRAM**	**RP**	**muxed**
dfadd	5940	96.93	10758	90	1	17	4481	98.63	9475	99	2	15
dfmul	2366	87.14	9542	97	1	10	1770	95.62	8479	99	2	9
dfdiv	6762	99.89	18233	88	1	11	5925	98,92	17774	113	2	10
mips	53996	105.73	9385	93	1	41	42545	102.57	8791	99	2	28
adpcm	133003	72.73	20468	127	2	89	109714	66.97	19554	159	4	60
gsm	33973	81.18	15480	116	3	52	31553	79.63	15635	110	3	52
motion	1223	70.72	22742	215	3	68	1142	70,69	21055	233	4	57
blowfish	1845551	77.57	10221	98	2	86	1237477	87.07	10111	131	4	52
sha	737735	101.50	18043	112	1	22	555349	97.82	16917	133	2	18

RP = Number of read ports to the MARC2 controller muxed = Maximum number of memory accesses multiplexed to any read port

and memory system configuration (number of read and write ports). For the memory system configuration, we also indicate the maximum degree of time-multiplexing of each distributed cache port. For the larger applications, only this multiplexing allowed the accelerators to fit into the FX200T, a fully spatial approach (one distributed cache port per access site) would not have been feasible. However, the memory port multiplexing reduces the clock rate. On the other hand, performance can be gained for some applications by using chaining, which can reduce cycle counts. Due to inaccuracies in the current Nymble delay estimators, however, some chained circuits suffer from a lowered clock frequency.

V. CONCLUSION AND FUTURE WORK

The Nymble compile flow is able to handle a wide spectrum of applications from the CHstone benchmark suite, reaching from DSP over cryptography to an entire microprocessor, and demonstrates the feasibility of fine-grained hardware/software partitioning with automatic interface synthesis. However, the flow does have significant potential for optimization, both in the performance and area domains.

Details that will be addressed include both peephole issues, such as sub-optimal control paths (e.g., wasted cycles between computation and use of a predicate) and improved accuracy of the chaining delay estimator, as well as system-level topics. One example for the latter is the analysis performed to allow selective cache flushes/invalidates when switching between hardware/software execution modes, which currently is very pessimistic. A more detailed analysis (e.g., combined with per-function annotation of C library calls) could significantly reduce the costly cache operations.

ACKNOWLEDGMENT

The authors would like to thank Xilinx for supporting their work.

REFERENCES

[1] Xilinx, Inc, *Vivado Design Suite User Guide – High-Level Synthesis*, 2012.

[2] M. Fingeroff and T. Bollaert, *High-Level Synthesis Blue Book*. Mentor Graphics Corp., 2010.

[3] Synopsys, Inc, *Synphony C Compiler User Guide*, 2011.

[4] H. Gädke-Lütjens, "Dynamic Scheduling in High-Level Compilation for Adaptive Computers," Dissertation, TU Braunschweig, 2011.

[5] S. Amarasinghe, J. Anderson, C. Wilson, S.-W. Liao, B. Murphy, R. French, M. Lam, and M. W. Hall, "Multiprocessors from a software perspective," *Micro, IEEE*, vol. 16, no. 3, pp. 52–61, 1996.

[6] H. Gädke-Lutjens, B. Thielmann, and A. Koch, "A Flexible Compute and Memory Infrastructure for High-Level Language to Hardware Compilation," in *Proc. Intl. Conf. on Field Programmable Logic and Applications (FPL)*, 2010, pp. 475–482.

[7] J. Villarreal, A. Park, W. Najjar, and R. Halstead, "Designing modular hardware accelerators in c with roccc 2.0," in *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual Int. Symp. on*. IEEE, 2010, pp. 127–134.

[8] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "LegUp: High-Level Synthesis for FPGA-based Processor/Accelerator Systems," in *Proc. Intl. Symp. on Field Programmable Gate Arrays (FPGA)*, 2011, pp. 33–36.

[9] *LegUp Documentation, Release 3.0*, 2013. [Online]. Available: http://legup.eecg.utoronto.ca/docs/3.0/legup-3.0-doc.pdf

[10] H. Lange, T. Wink, and A. Koch, "MARC II: A Parametrized Speculative Multi-Ported Memory Subsystem for Reconfigurable Computers," in *DATE'11*, 2011, pp. 1352–1357.

[11] T. Callahan, "Automatic compilation of C for hybrid reconfigurable architectures," Ph.D. dissertation, UC Berkeley, 2002.

[12] B. Thielmann, T. Wink, J. Huthmann, and A. Koch, "RAP: More Efficient Memory Access in Highly Speculative Execution on Reconfigurable Adaptive Computers," *2011 Intl. Conf. on Reconfigurable Computing and FPGAs*, pp. 434–441, 2011.

[13] B. Thielmann, J. Huthmann, and A. Koch, "Evaluation of speculative execution techniques for high-level language to hardware compilation," *6th Intl. Workshop on Reconfigurable Communication-Centric Systems-on-Chip (ReCoSoC)*, pp. 1–8, 2011.

[14] ——, "Precore - A Token-Based Speculation Architecture for High-Level Language to Hardware Compilation," in *2011 21st Intl. Conf. on Field Programmable Logic and Applications*. IEEE, 2011, pp. 123–129.

[15] Y. Li, T. Callahan, E. Darnell, R. Harr, U. Kurkure, and J. Stockwood, "Hardware-software co-design of embedded reconfigurable architectures," in *Proc. Design Automation Conference (DAC)*, 2000, pp. 507–512.

[16] H. Lange and A. Koch, "Architectures and Execution Models for Hardware/Software Compilation and their System-Level Realization," *IEEE Trans. Comput.*, vol. 59, no. 10, pp. 1363–1377, 2010.

[17] V. Rajagopalan, V. Boppana, S. Dutta, B. Taylor, and R. Wittig, "Xilinx Zynq-7000 EPP – An Extensible Processing Platform Family," in *23rd Hot Chips Symposium*, 2011, pp. 1352–1357.

[18] Ashley Stevens, "Introduction to AMBA 4 ACE," 2011.

[19] *LLVM's Analysis and Transform Passes*, 2013 (accessed April 30, 2013).

[20] James Burill et al., "A Scalable Compiler for Analytical Experiments," 2007. [Online]. Available: http://www.cs.utexas.edu/users/cart/Scale/

[21] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis and transformation," in *CGO*, 2004, pp. 75–88.

[22] B. R. Rau, "Iterative modulo scheduling," in *Proc. Intl. Symp. on Microarchitecture - MICRO 27*, 1994, pp. 63–74.

[23] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner, "Equality saturation: a new approach to optimization," in *ACM Proc. Symp. on Principles of Programming Languages (POPL)*, 2009, pp. 264–276.

[24] Y. Hara, H. Tomiyama, S. Honda, and H. Takada, "Proposal and quantitative analysis of the chstone benchmark program suite for practical c-based high-level synthesis," *Journal of Information Processing*, vol. 17, pp. 242–254, 2009.