Chapter 1
Adaptive Computing Systems and their Design Tools

Andreas Koch

Abstract While reconfigurable adaptive computing has many proven advantages over conventional processors, in practice, it is often limited to niche applications. This situation, which we aim to resolve with our research, is often linked to the lack of programming languages for adaptive computers that are familiar to software developers. We present a compile flow capable of translating general-purpose C programs to hybrid hardware/software applications for execution on an adaptive computer and give an overview of the required advances in compiler technology as well as in computer architecture and operating system design.

1.1 Introduction

As demonstrated numerous times, reconfigurable computing can have significant advantages over conventional processors for a wide range of applications [30]. Despite these advantages, however, it is only rarely employed outside of academic settings.

One of the key reasons for this discrepancy is the difficulty of actually programming a reconfigurable computer. Most commonly, this is done by designing a compute architecture for the algorithm from scratch, which is then described in a hardware design language such as VHDL or Verilog.

While this approach can result in very high-performance implementations, it requires programmers to be experienced in computer architecture, digital logic design and hardware design languages. Only very few software developers actually have these skills. Thus, the power of reconfigurable computing remains unavailable to most potential users.

Andreas Koch
Embedded Systems and Applications Group, Dept. of Computer Science, Technische Universität Darmstadt, Germany, e-mail: koch@esa.cs.tu-darmstadt.de

In recent years, many attempts have been made to close this gap and lift the abstraction level of reconfigurable computer programming to that of conventional software-programmable processors.

To this end, we have been working on Comrade, a compiler for automatically translating general-purpose ANSI C programs for computers containing both a conventional and a reconfigurable processor. As we will describe below, the choice of C with its pointers and possibly irregular control flow has significant effects on both the compile flow as well as the target computer architecture and its operating system.

Our DFG-funded project "Adaptive Computing Systems and their Design Tools" was initiated prior to the DFG Priority Programme 1148, but has been associated with the Programme right from the start due to the thematic closeness. Since the schedules of our on-going project and the concluding Priority Program are thus out-of-phase, this report will concentrate on the major results achieved during the era of the SPP. Sect. 1.7 will give some perspective on the issues we are addressing in our current research.

1.2 Execution Model

In contrast to traditional research on High-Level Synthesis (HLS) [10] our target architecture is assumed to always contain a conventional software-programmable processor (SPP) in addition to a reconfigurable processor. While the compute-intensive parts of a program can be implemented in a spatially distributed fashion for high-performance, other parts of the program that are either unsuitable (e.g., I/O using printf or similar functions) or that are only used rarely and would not justify the permanent allocation of computing area (e.g., error handling) are left in software on the SPP.

This target architecture also avoids a basic problem of High-Level Synthesis, which aimed to translate the *entire* program into hardware: If the input program contained a construct (e.g., function calls, irregular control flow, dynamic memory allocation, etc.) that the specific HLS algorithm could not handle, the translation was aborted completely. While we also intend to translate our input language to the widest practical degree, our flow can always fall back to the SPP to execute program parts that the flow cannot process yet due to implementation limitations, or that would exceed the capacity of the reconfigurable device. This allows incremental development of the compiler, with increasing parts of the profitable computations of a program being moved for acceleration to the reconfigurable device.

We call such an architecture an *adaptive computer system* (ACS). To be more precise, we differentiate between the underlying reconfigurable device (RD), which can be an either an FPGA or a coarse-grained reconfigurable array (CGRA), and the reconfigurable compute unit(s) (RCU) that can be mapped to it.

When combining multiple processing elements (such as the SPP and RCU of an ACS), the manner of their interaction must be specified. This is done by the *execution model* (discussed in greater detail in [24]).

Different ACS compilers employ different models, which differ mainly in the granularity of the SPP/RCU partitioning.

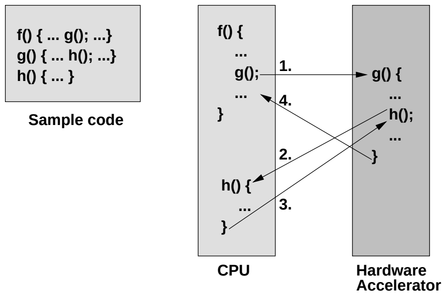


Fig. 1.1 ASH Execution Model [24]

The ASH system [3] can switch between the SPP and a hardware accelerator (note: not an RCU, ASH targets ASICs) only at procedure boundaries shown in Fig. 1.1). This rather coarse granularity leads to an *entire* procedure being ineligible for hardware acceleration even if only a single non-compilable construct is present. ASH exceeds classic High-Level Synthesis by allowing software functions to be called from the hardware accelerator, however.

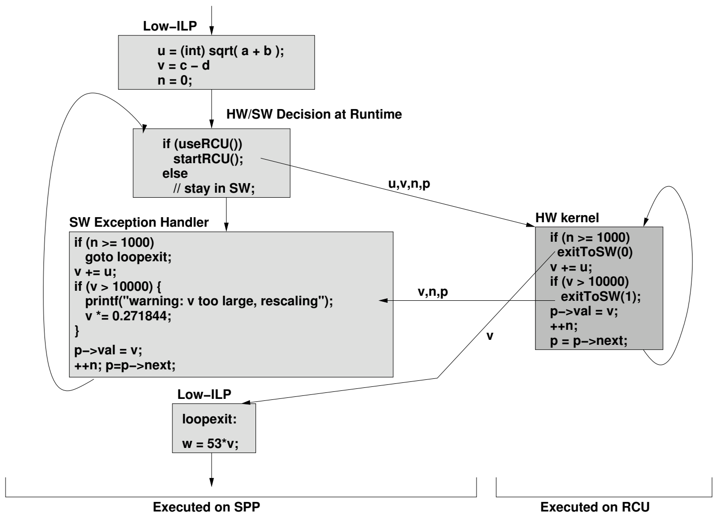
Other systems, such as GarpCC [4], Nimble [29] and our own Comrade, allow a finer-grained partitioning *within* of functions. For the following discussion, consider the sample program shown in Fig. 1.2.

```
...
u = (int) sqrt(a + b);
v = c - d;
for (n=0; n<1000; ++n, p=p->next) {
  v += u;
  if (v > 10000) {
    printf("too large, rescaling");
    v *= 0.271844;
  }
  p->val = v;
}
w = 53 * v;
...
```

Fig. 1.2 Sample program with RCU-infeasible statements

The program contains a typical hardware kernel (a loop) which is surrounded by code with a low degree of instruction level parallelism (ILP) statements. For this example, we assume that the functions sqrt and printf are not efficiently compilable to an RCU. Thus, they, as well as the rest of the low-ILP code, should be left on the SPP.

All three of these finer-partitioning compilers can successfully perform this operation. However, they differ in their handling of the RCU-unsuitable code within



```
u = (int) sqrt( a + b );
v = c – d
n = 0;
```

```
if (useRCU())
   startRCU();
else
   // stay in SW;
```

```
if (n >= 1000)
   goto loopexit;
v += u;
if (v > 10000) {
   printf("warning: v too large, rescaling");
   v *= 0.271844;
}
p–>val = v;
++n; p=p–>next;
```

```
if (n >= 1000)
   exitToSW(0)
v += u;
if (v > 10000)
   exitToSW(1);
p–>val = v;
++n;
p = p–>next;
```

```
loopexit:

w = 53*v;
```

Fig. 1.3 Example in the Nimble execution model [24]

the loop kernel. Using dynamic profiling, they might discover that the condition $v > 10000$ occurs only rarely, and thus moving the loop to the RCU is profitable despite the infrequent switches to the SPP (for the printf and the floating-point multiplication). Such a switch requires exchanging the live variables between the two processors and can have a significant overhead. After a switch to the SPP, GarpCC and Nimble then execute the entire remainder of the current loop iteration in software (they generate both hardware and software versions of each kernel), shown in Fig. 1.3. Only when re-entering the loop for another iteration is the decision made whether to continue on the SPP or switch back to the RCU.

The model we use in Comrade (shown in Fig. 1.4) is even more fine-grained: We can now switch between individual statements, moving just the printf and the float multiplication to the SPP as a so-called *software service*. After completion of the service, execution switches back to the RCU immediately. Note that, with the finer granularity, *fewer* live variables need to be transferred between the two processors. In this fashion, a single hardware kernel can access multiple software services, each with just the code for the requested function.

The exchange of live variables has far-reaching architectural consequences when *pointers* are to be supported (as we have to do for C in Comrade). In this case, address arithmetic performed on the SPP and RCU must be compatible, and both processors must share a coherent view of memory (reads and writes can be performed by both sides). This will be discussed in greater detail in the next Section.

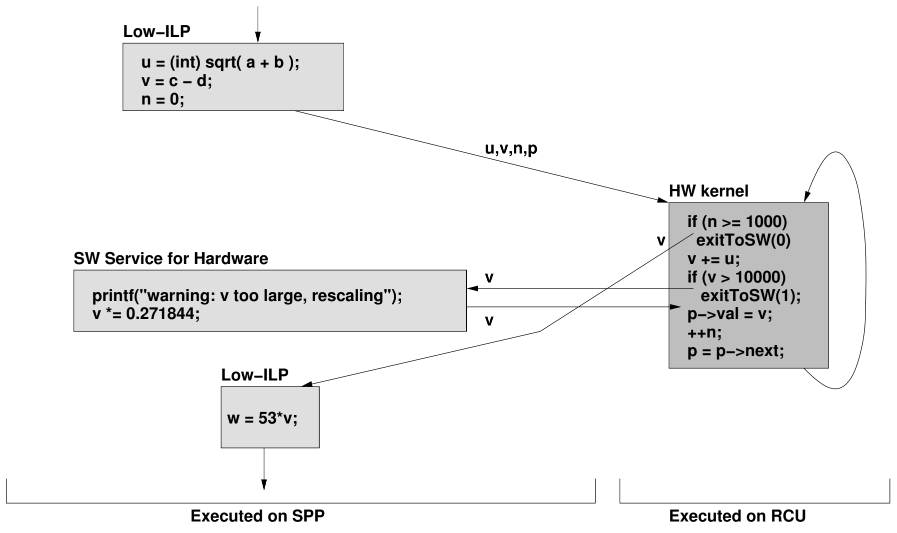


Fig. 1.4 Example in the Comrade execution model [24]

1.3 ACS Architecture

The Comrade model of execution requires a number of capabilities from its underlying computing platform to be practical. As explained above, our model requires low-latency SPP-RCU communication for the exchange of the live variables. Note that this does not have to be a high-bandwidth link, since generally only very few variables have to be exchanged (due to the small scope of our software services). Pointer addresses must be freely exchangeable across the SPP-RCU boundary, and both processors must have high-throughput access to a shared main memory, possibly in the presence of virtual memory (e.g., when running the ACS under a full-scale Linux OS). For security reasons, the RCU must of course respect the access permissions imposed by memory protection. If these apply at the process level (the general case for Unix variants), the code of the SPP software *within* the hybrid SPP/RCU process should also be inaccessible to a possibly rogue RCU. Furthermore, the RCU must be prevented from interfering with OS scheduling decisions, e.g., by denying other processes (or the OS itself) access to main memory. Finally, the software part of a hybrid SPP/RCU process should execute at full speed, without slow-down due to the RCU.

A much more detailed discussion of these aspects can be found in [23, 24, 25].

1.3.1 Reconfigurable System-on-Chip Architecture

To achieve these goals and actually make the Comrade execution model feasible on real hardware, we implemented suitable ACS architectures as reconfigurable systems-on-chip (rSoC), at first using the Xilinx Virtex II Pro-based ML310 platform. The II Pro FPGAs embed SPPs (300 MHz PowerPC 405 CPUs) into a reconfigurable logic array suitable as RCU. Since the interfaces, both between SPP and RCU as well as the rest of the system (memory, I/O, etc.) are mainly realized using reconfigurable logic, they can be changed to fit the requirements we formulated above.

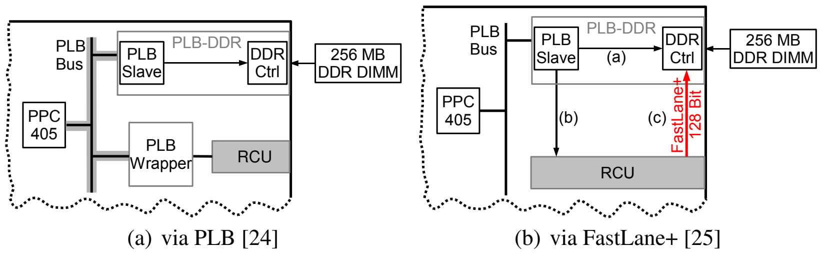


Fig. 1.5 RCU integration techniques

The default system-on-chip architecture supported by Xilinx EDK design tools is shown in Fig. 5(a). Note that both SPP-RCU communication as well as memory accesses from the RCU have to pass through the single Processor Local Bus (PLB). While the multi-purpose use of such standard busses allows the flexible composition of SoC architectures, the performance suffers: The standard approach achieves a transfer rate of just 213 MB/s (of the theoretically possible 1600 MB/s) for RCU memory accesses while the SPP executes a (mostly idle) Linux.

To improve RCU-to-memory bandwidth, we developed the FastLane+ architecture (Fig. 5(b)): Now, the RCU is attached directly to main memory using a dedicated 128b wide bus. This increases the bandwidth available to the RCU significantly (now to 1424 MB/s, again, while Linux is idling on the SPP), fulfilling one our requirements stated above. But in addition, FastLane+ ensures that the SPP has override priority when accessing memory and can even terminate RCU-initiated transfers. This guarantees that the OS (and programs chosen by the OS scheduler) are never starved by lack of memory bandwidth (a critical aspect for OS operations such as interrupt handling or time-critical disk or network I/O).

Since the RCU, acting as a slave, can be accessed from the SPP via the PLB, SPP-RCU transfers (such as the live variable exchange required when starting the RCU, or performing a software service on the SPP) can also be performed quite quickly (just 20ns/40ns per 32b variable read/written).

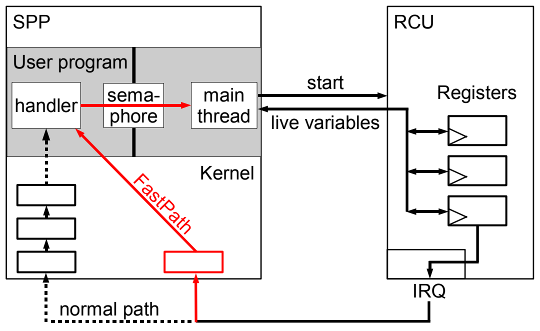


Fig. 1.6 RCU-SPP Signalling [25]

1.3.2 Operating System Integration

Improving the data transfer between SPP, RCU and memory is necessary to implement our execution model in a real system, but does not suffice on its own. To also operate securely under a protected virtual memory scheme while exchanging pointers, running software applications at full speed (unhindered by the presence of an RCU), and allowing low-latency RCU-SPP switches (for performing software services), the purely hardware architectural measures described above are inadequate: We now need support from the operating system. To demonstrate the feasibility of our approach even when running a full-scale OS, we chose Linux (which continues to grow market share in the embedded systems area) over a less demanding, but functionally more limited real-time kernel.

1.3.2.1 RCU-SPP Signalling

Normally, the RCU raises an interrupt to get the attention of the SPP (e.g., to provide a software service). Even in a Linux version patched for low-latency responses, the processing of such an interrupt takes 62 μs on the ML310 platform. The normal path (labeled as such in Fig. 1.6) sketches how an interrupt passes through numerous layers in the Linux kernel, before it finally reaches the handler in the software main thread of the hybrid HW/SW user program.

Our FastPath approach takes a number of measures to improve the interrupt response latency. First, a dedicated interrupt vector on the PowerPC 405 is assigned the RCU, allowing the bypassing of the kernel interrupt processing layers. Then, instead of handling the interrupt in a device driver and using mechanisms such as file descriptors or similar to forward it to the user program, FastPath can jump directly to a previously registered callback function which executes in context of the user program (has access to global variables and is subject to access limitations). The callback is executed in a separate thread in parallel to the existing threads of the user program. As usual, execution can be synchronized using a semaphore. But in FastPath, this semaphore is realized using a dedicated special SPP register (instead of an in-memory data structure that would also cause bus traffic). By proceeding in this fashion, we can reduce software service latency down to the 9.6...2.7 μs, an improvement of 6.5x to 23x over the original implementation. The FastPath latency on the comparatively slow 300 MHz PowerPC 405 outperforms even special sub-kernel facilities (such as RTAI and LXRT) running on multi-GHz desktop CPUs [28].

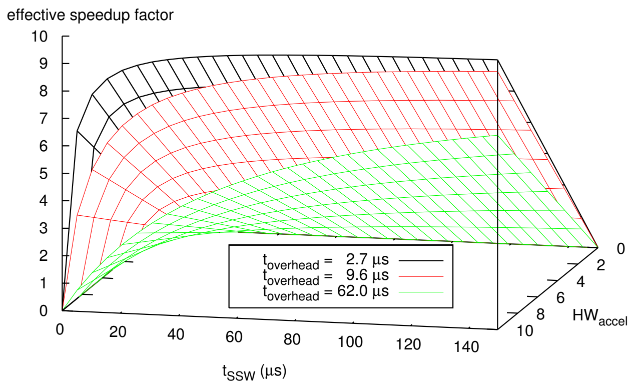


Fig. 1.7 Effective speed-up vs. RCU-SPP signalling delay [25]

The accelerated RCU-SPP signalling has a tremendous effect on the practical partitioning granularity for SPP-RCU execution (shown in Fig. 1.7). To give an example interpretation: Assuming the RCU executes an algorithm $HW_{accel} = 10$ times faster than the SPP, requires an average $t_{overhead} = 6.15\mu s$ for the RCU-SPP signalling, and disregarding the time for the software service itself (which would also need to be executed when running only on the SPP), we can accept a switch to a software service every $t_{SSW} = 55.4\mu$s and still achieve an effective acceleration of *9x* over the SPP. With FastPath, the quick SPP-RCU switches required by our execution model are demonstrated to be achievable on real hardware/software platform.

1.3.2.2 Shared Virtual Memory

Fast SPP/RCU accesses to a shared physical memory (as described in Sect. 1.3.1) can be implemented purely in hardware. But for safety and security reasons [34], protected virtual memory is becoming more common even in traditional embedded OS such as LynxOS, VxWorks, and of course embedded Linux. To support our model of execution even when running the ACS under such an OS, RCU memory addressing and accessing must be coordinate with the SPP-side memory management unit (MMU). We have implemented and evaluated two significantly different solutions to this problem.

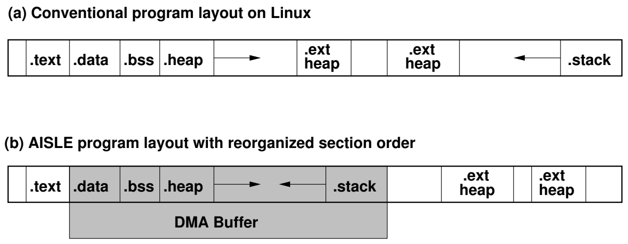


Fig. 1.8 Conventional and AISLE Program Layouts [24]

Figure 1.8 shows the simpler of the two approaches, the Accelerator-Integrating Shared Layout for Executables (AISLE) [24]. In this model, we map all data regions (uninitialized, initialized, heap, and stack) into a so-called DMA buffer by modifying the ELF program loader [1]. A DMA buffer is a contiguous address range accessible to the RCU that is locked into physical memory (not subject to being paged to and from disk). This also ensures that the same virtual address will always map to the same physical address. Thus, virtual $\leftrightarrow$ physical address translation consists of just adding/subtracting constant offsets. Analogously, addresses generated by the RCU can easily be constrained by a simple bounds check to always lie within the buffer, protecting other processes from a potentially rogue RCU. AISLE also protects the executable code (the so-called .text segment) of the user program itself from the RCU by keeping it *outside* the buffer. In this fashion, both SPP and RCU operate on the same virtual addresses and can transparently process data stored in the shared memory (no inefficient copying between SPP- and RCU-accessible memories is required). Reference [24] explains AISLE in greater detail, also covering topics such as cache coherency between SPP and RCU, and size management of the DMA buffer.

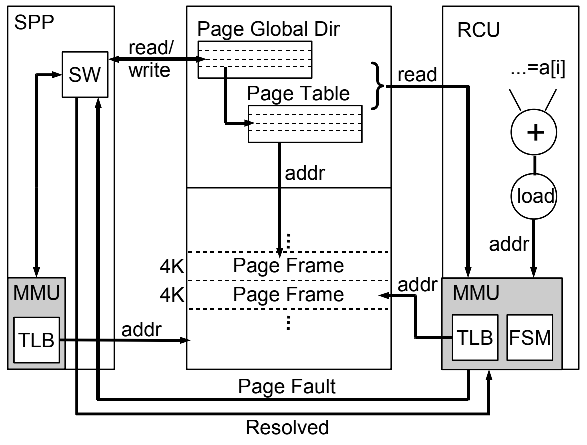


Fig. 1.9 SPP-RCU shared virtual addressing with PHASE/V [25]

While AISLE provides the desired functionality in a very efficient fashion (as will be shown below), it does have a number of limitations: Once the DMA buffer is allocated, it can be resized only with considerable overhead. In many cases, this will lead to the maximal size buffer being allocated to the RCU. Since, by its very nature, the DMA buffer does not participate in paging, it will always occupy its full size in physical memory.

As a second, potentially more flexible approach, we implemented the Processor-Hardware Accelerator Shared Environment with Virtual addressing (PHASE/V, shown in Fig. 1.9) [25]. Here, the RCU fully participates in all virtual memory operations, including demand paging, variable virtual-physical mappings, and discontiguous physical memory ranges. With demand paging now supported, physical memory is only allocated when needed and released when required otherwise (in contrast to the statically pre-allocated DMA buffer of AISLE). Analogously to the SPP, the RCU uses a translation-lookaside buffer (TLB) to cache virtual-physical translations (taking only a single clock cycle for a translation), but is fully capable of performing a walk of the page mapping tables itself to determine as-yet unknown mappings on a TLB miss. If no physical page is found, the RCU uses the FastPath signalling scheme to request the SPP to handle the page fault (e.g., load the missing page from disk). In the reverse direction, the OS on the SPP informs the RCU when it alters the mappings (e.g., when a memory page is swapped out to disk), causing the RCU to synchronize its TLB with the one in the SPP.

1.3.3 Evaluation

To stress-test the performance of the different schemes, we use the traversal of a linked-list, randomly distributed in memory [25]. AISLE and PHASE/V have similar performance, until the number of virtual pages in the working set exceeds the RCU-TLB capacities (which occurs between 16K and 32K list elements). Then, PHASE/V begins to slow down (e.g., taking 2.3x the time of AISLE at 128K list elements). However, even in this extreme case, the RCU remains 23% faster than a pure-software version running of the SPP. Thus, the often quoted maxim that reconfigurable computing is mainly useful for stream processing has been invalidated by our research: RCUs can also be faster than SPPs even for highly irregular pointer-chasing applications. In such scenarios, performance will be limited by the memory bandwidth, which we can fully exploit with FastLane+ on the RCU.

More detailed evaluations are presented in [24, 25]. On of the experiments also demonstrates that an active RCU only marginally slows the pure software processes scheduled by the OS to execute on the SPP. In summary, by taking the appropriate architecture and OS measures, we have now created an environment which enables the practical use of the compute model of Sect. 1.2.

1.4 Hardware/Software Co-Compilation Flow

Now that we have defined and implemented a suitable reconfigurable target architecture, we can examine the programming tools we developed to make the technology accessible to software developers.

1.4.1 Overview

The heart of the flow is the Comrade compiler. As usual for modern compilers, it is organized in a multi-pass manner. First, common compiler operations are performed (lexing, parsing, machine independent optimization). At this step, the intermediate representation (IR) of the compiler front- and middle-end is exported from the system (as a C program) and subjected to dynamic profiling. As a result, actual execution frequencies can now be back annotated into the IR to guide further processing.

Based on the profiling data, program is partitioned for SPP and RCU execution (see Sect. 1.4.2 for more details). The software part destined for the SPP is enriched with interface code (to exchange the live variables at SPP/RCU execution boundaries and start the RCU), exported as a C program and fed into a conventional software C compiler.

The part to be executed on the RCU needs to be processed further by Comrade. It is first transformed into a control flow graph (CFG) in Static Single Assignment form, which forms the base of Comrades hardware-centric intermediate representation, the Control Memory Data Flow Graph (CMDFG, see Sect. 1.4.3). Hardware-specific optimizations (e.g., parallelizing memory accesses) are then performed by transforming the CMDFG. The hardware for the RCU is finally generated from the optimized CMDFG as separate data path and dynamically scheduled controller (see Sect. 1.4.4) in the form of RTL Verilog netlists. These are then synthesized together with the fixed rSoC architecture of the ACS (Sect. 1.3.1), and handed to the FPGA vendor tools for implementation (map, place, route, bitstream generation).

1.4.2 Profile-based Inlining and Partitioning

Dynamic profiling data is gathered early in the compile process after some machine independent optimizations (e.g., goto removal) have already been performed. This profile is first used to inline only the most heavily used functions. In this way, we avoid the code size explosion (specifically, the associated area requirements on the RCU) that can occur when inlining indiscriminately.

For partitioning, multiple versions of the loops making up the potential RCU kernels are created. This is done in an inside-out fashion, starting with just the innermost loop(s) and then widening the scope (possibly merging formerly separate loops into one kernel candidate) until the RCU area is exceeded. Figure 1.10 shows

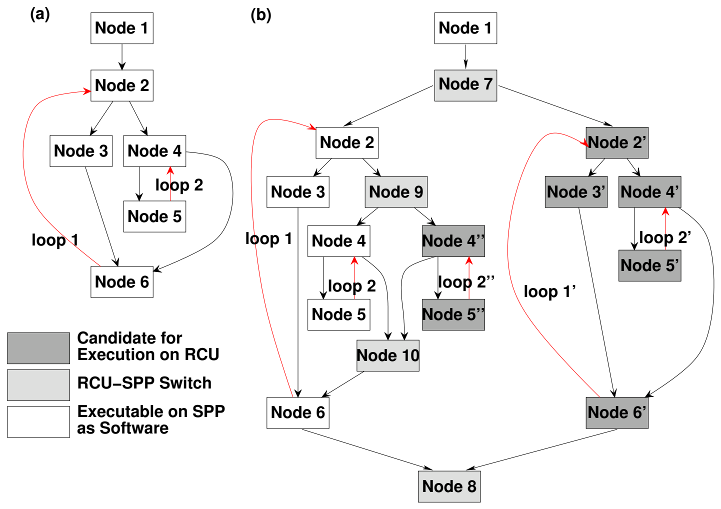


Fig. 1.10 Generating alternate SPP-RCU partitionings [15]

an example of this for two loops: The CFG shown in (a) is expanded into a CFG that has the entire loop nest executing on the SPP (left), the outer loop on the SPP and the inner on the RCU (middle), and the entire nest executing on the RCU (right).

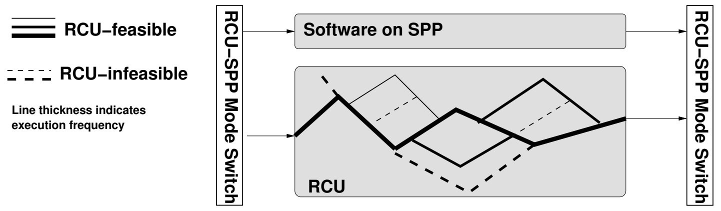


Fig. 1.11 RCU path construction [15]

We then examine each of these candidate RCU kernels by building valid *paths* of operations that can be natively executed on the RCU. These will generally bypass external I/O and all functions which were not inlined (either due to adverse profiling data or lack of source code). Such operations will be marked as potential software services. *Valid* means for path that it will pass *through* the candidate region (enter and leave it). Path construction first constructs a valid path and then expands it, adding rejoining sub-paths in order of decreasing execution frequency. This is done until the entire candidate is covered by RCU-suitable paths and software services, or the RCU area is exhausted. More details on these steps can be found in [13, 14]. As examples, 80% of the instructions of the Versatility wavelet image [33] compressor and the ADPCM audio coder can potentially be moved to on the RCU in this fashion.

1.4.3 CMDFG Intermediate Representation

We experimented with a number of intermediate representations for hardware/software co-compilation. This included various SSA forms, such as plain SSA [6], array SSA [18], and our own initial attempt for a data-flow controlled form heavily emphasizing parallel execution [16] (DFC-SSA, which later proved too difficult to schedule). None of these proved sufficiently expressive.

To fill this need, we developed Control Memory Data Flow Graphs (CMDFG), described in greater detail in [8, 9]. The CMDFG expresses control flow (extracted from the intermediate SSA-CFG created during compilation), inter-operator data flow (as a classical data flow graph) and memory dependencies (such as those computed by alias analysis). It is somewhat similar to the Program Dependence Web [5] and Program Dependence Graph [7], which are also low-level fine-grained intermediate representations. However, they do not express memory dependencies at all and rely on constructs such as γ and β functions or data-flow switches to guide computation. The CMDFG is closer to the actual hardware: It employs distributed multiplexers for this purpose, with control edges running from a condition to the multiplexer input nodes to determining the selected data.

```
unsigned int i, s;
s = 0;
for (i = 0; i < 6; i++) {
  if (i > 0) {
    if (i == 5) {
      s = s / i;
    } else {
      s = s + i;
    }
    s = s + i;
  }
}
printf("s = %d\n", s);
```

(a)

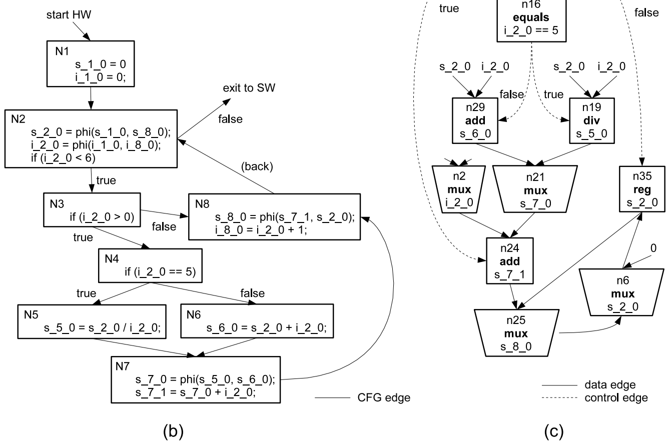
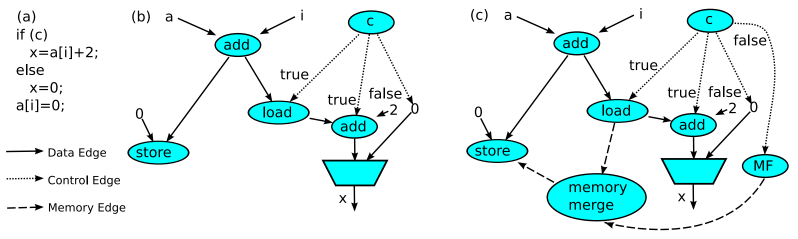


Fig. 1.12 (a) Sample program, (b) SSA-CFG, (c) CMDFG [8]

Figure 1.12 shows a very simple example of such a CMDFG (for simplicity, without memory edges). For the source code shown in (a), the SSA-CFG is shown in (b). The body of the loop is represented by the CMDFG shown in (c). For clarity, this omits the increment of the index variable and the test of the loop condition. Note that the true condition of n13 controls both the nested if statement as well as the assignment to s at the same level. In contrast to many previous hardware compilers, Comrade can easily handle even complex nested control structures in loops (including other loops!) using the CMDFG.



```
(a)
if (c)
  x=a[i]+2;
else
  x=0;
a[i]=0;
```

Fig. 1.13 (a) Sample program, (b) CDFG, (c) CMDFG

Figure 1.13 shows a simple example how memory edges explicitly express memory dependencies. In the CDFG without memory edges (b), the load and store of the sample program (a) could execute in arbitrary order and potentially violate the write-after-read requirement. The CMDFG (c) enforces correct execution, with the aid of two additional constructs: A Memory Forwarder (MF) executes if allowed by the incoming control edge, and allows execution of its successor memory nodes. The memory merge node allows execution of its successor memory nodes if at least one of its direct memory predecessors was executed. In this fashion, the write a[i]=0 will only execute if c was true, and the load in x=a[i]+2 has already completed, *or* c was false, and the MF node executed, leading to the execution of the memory merge node, which in turn allows the store node to execute.

By selectively modifying the memory dependence graph, we can easily express parallel memory accesses once we have proven their independence, e.g., by alias analysis methods. The memory aspects of the CMDFG are discussed in greater detail in [9].

1.4.4 CoCoMa Controller Model

Another core result of our research was creation of new dynamic scheduling semantics for the CMDFG. Beyond the capabilities of traditional dependency graphs (e.g., control data flow graphs), which have a node execute and produce a result as soon as all of its operands are available (and a possibly incoming control edge is also valid), we can now express the *deletion* of results. This can extend to stopping a calculation already in progress once it has been determined that the result will not be needed. Since this relation is transitive, an entire chain of unneeded calculations can be stopped, and the operators be made available for the next set of operands.

These semantics are defined by the Comrade Controller Micro-architecture (CoCoMa), which also describes their mapping into hardware. At the abstract level, we now a structure similar to a Petri net with two kinds of tokens: *Activate Tokens* (AT) indicate the presence of data at the source of an edge, Cancel Tokens (CT) erase an AT (and its associated data item) when they meet. ATs generally move in the direction of data flow, while CTs move in the opposite direction.

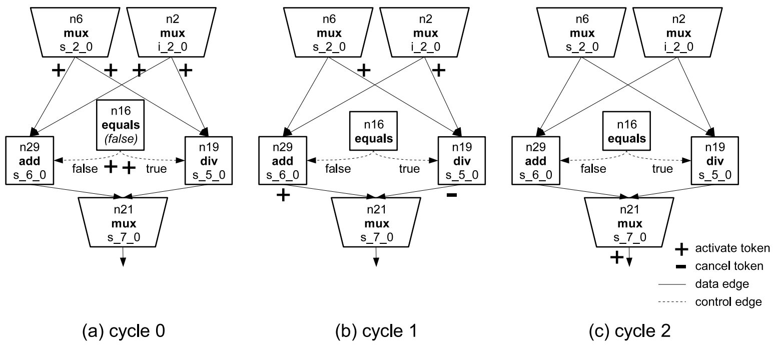


Fig. 1.14 Interaction of Activate and Cancel Tokens [8]

A simple example for the inner if of the Fig. 1.12.a is shown in Fig. 1.14 for three successive clock cycles (assuming the addition operator has a latency of one cycle, and the division takes longer). In cycle 0, the values for s_2_0 and i_2_0 have been computed (both of this happens in block N2 of the SSA-CFG). Thus, ATs are present on all outgoing edges of the associated multiplexers n6 and n2. Both the addition and division operators in CMDFG nodes n29 and n19 now start to compute (since all of their operands are available). But the condition i == 5 is assumed to have evaluated to *false* in this example. Thus, only the operator associated with this state, namely the addition, is allowed to complete (an AT travels along the control edge labeled *false* to node n29). On completion, the operator consumes its operands (and their associated ATs) on the incoming data edges from n6 and n2, and produces the result and an AT at its output in cycle 1. The other branch of the condition i == 5 is handled as follows: The division node at the destination of the control edge also receives an AT in cycle 0. However, since the condition state *false* does not match the *true* requirement on the control edge, the AT is turned into a CT at the output of the unneeded division operator in cycle 1. Note that the division has been started in cycle 0 (the same as the addition), but is now canceled in cycle 1. In cycle 2, the addition result and its AT travel *downward* through n21 for use in further computation. The CT at the division operator has now traveled upwards and erases the incoming operands (and their ATs) which were not consumed by the terminated division. Thus, the entire CMDFG subgraph is now available for the next loop iteration. In static scheduling, this would only happen after the division could be completed (even though its result would not be needed).

From this basic scheme, more complex behaviors can be derived. For example, the CMDFG memory dependencies (described in Sect. 1.4.3) are also modelled as ATs travelling along memory edges. Also, more complex transition rules are required when describing deeply nested structures, especially loops. The best formulation of these is still the subject of active research.

Despite its complexity, CoCoMa scheduling, which described in greater detail in [8], can lead to significant speedups, even when compared with other more advanced dynamic schemes such as lenient execution [3]. It does have additional cost when compiled into hardware, though: Each data register now requires an additional two bits to hold an AT and CT each. Furthermore, the token transition rules need to be implemented as logic networks, also requiring RCU area. As shown in [9], the CoCoMa overhead for realistic applications is in the range of 600-1700 slices, which should be considered acceptable given current RCU device sizes.

1.5 Infrastructure

Beyond the execution model, an appropriate practically realizable ACS architecture and the core compile flow, we developed a number of adjunct technologies.

1.5.1 Parametrized Module Library

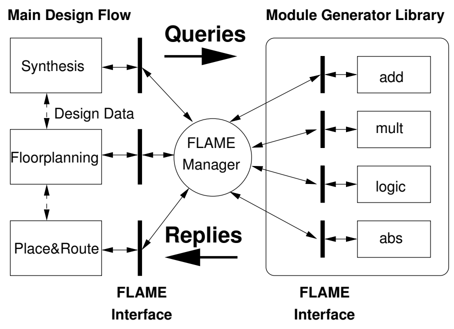


Fig. 1.15 GLACE Library and FLAME API [15]

Many steps of the Comrade compile flow need target hardware-specific data. This ranges from area/delay estimates for specific operations to information about their physical and logical interfaces down to (possibly pre-placed) structural netlists.

All of these aspects are encapsulated in the Generic Library for Adaptive Computer Environments (GLACE), which we introduced in [32].

GLACE provides all of the basic operators (arithmetic, logic, memory) as well as the system interfaces (I/O registers, SPP-RCU signalling) we need for C-to-hardware compilation in a parametrized fashion [21]. For example, we can retrieve from GLACE data about a 27x18 bit unsigned multiplier. The underlying data is organized and accessed by the Flexible API for Module-based Environments [20] (FLAME, shown in Fig. 1.15), which defines both passive (design data model) and active components (query/reply scheme for interacting with the module library). The data model is organized into different *views*, each representing a subset of aspects relevant for a specific stage of the compile flow. Examples include *behavior* (which lists operator semantics), *synthesis* (giving are/delay and interface data), *topology* (holding placement information), and *place* (which encompasses pre-placed EDIF netlists).

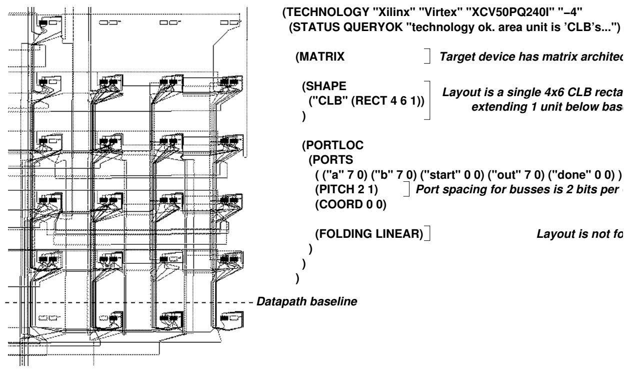
GLACE data, accessed via FLAME, is used in Comrade e.g., for the hardware/software partitioning step. It guides both the expansion of nested kernels from pure software to pure hardware as well as the construction of RCU-executable paths through the candidate kernels. It also provides the pre-placed operator netlists for the datapath, which are then combined with the CoCoMa controller and the system interface into a single RCU configuration.

GLACE has been continuously updated for the duration of the project, including adding support for new hardware blocks on target FPGAs (e.g., hardwired multiplier blocks). Due to the generality of the FLAME API [20] and the Library Specification [21], such extensions are completely transparent to the core compile flow. Furthermore, the development environment for new modules, which is based on Brigham Young University's JHDL [2], has also been updated to support all basic blocks of XIlinx Virtex 5 devices (new memories, digital signal processing, etc.) [31].

1.5.2 Physical Design Aspects

We have always also considered the physical design aspects of reconfigurable computing. This began with Structured Design Implementation methodology [19], which described how to assemble regular datapaths from pre-placed and pre-routed parametrized modules. We have carried forward this intent into the *topology* and *placed* views of FLAME.

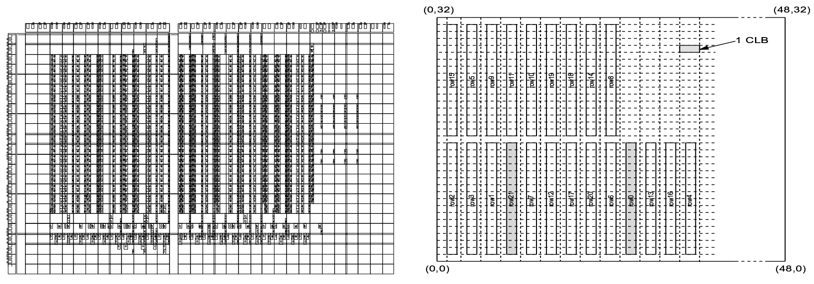
As an example, Fig. 1.16 shows the regular layout of an 8 bit unsigned multiplier on a Xilinx Virtex FPGA and the associated FLAME *topology* view. This describes the extent of the layout (separating regular and irregular parts) as well as the locations of external ports (which are organized here at a pitch of two bits per CLB height). The regular parts of a module have a consistent horizontal data flow, consisting of regularly spaced bits. Above and below this regular area, irregular components (such as module-local controllers and overflow computation) can be arranged. A purely-module based approach is inefficient, however, when compos-



```
(TECHNOLOGY "Xilinx" "Virtex" "XCV50PQ240I" "–4"
  (STATUS QUERYOK "technology ok. area unit is 'CLB's...")

   (MATRIX                  ]  Target device has matrix architec

    (SHAPE                  ]  Layout is a single 4x6 CLB recta
     ("CLB" (RECT 4 6 1))   ]        extending 1 unit below bas
    )                       ]

    (PORTLOC
     (PORTS
      ( ("a" 7 0) ("b" 7 0) ("start" 0 0) ("out" 7 0) ("done" 0 0) )
      (PITCH 2 1)        ] Port spacing for busses is 2 bits per
      (COORD 0 0)

      (FOLDING LINEAR)]                         Layout is not fo
     )
    )
   )
```

Fig. 1.16 Pre-placed 8-bit multiplier and FLAME description [32]

ing more complex logic expressions from individual modules: Each operator would be mapped to its own separate module (which would at least be one LUT column long), even though multiple operators could conceivably be mapped together into a single LUT column (as long as the number of LUT inputs is not exceeded). To solve this problem, we have developed a universal generator for logic modules that accepts arbitrarily complex logic expressions and performs inter-operator logic optimization and mapping to generate a single regularly pre-placed module for the entire expression [37, 22].



(a) Purely horizontal (b) Clustered horizontal sub-datapaths

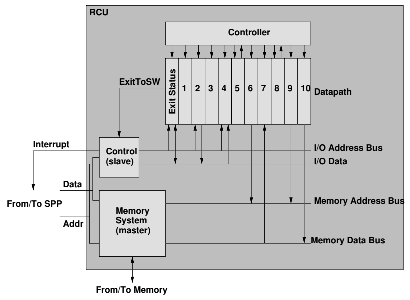
Fig. 1.17 Regular placement of Wavelet datapath [36]

For high performance, the regular layout style also should be preserved when assembling an RCU datapath from multiple modules. This requires custom tools, since the vendor CAD tools are not specialized for datapath layout. An easy way to achieve regular inter-module placement is a linear arrangement of modules, this is an approach that has already been used in earlier projects [29]. An example of such a linear layout for a kernel of a Wavelet image compression algorithm [33] is shown in Fig. 17(a).

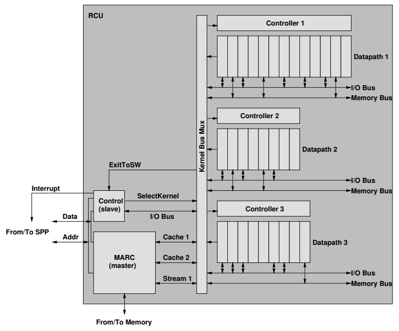
However, this technique obviously does not scale with increasing datapath complexity: At some point, the linear distances become excessive and the performance decreases (despite the regular linear layout). To this end, we have developed a combined approach ClaP [36] that partitions the datapath into clusters of linear sub-datapaths (shown in Fig. 17(b) for the same circuit). Within each cluster, a strictly horizontal data flow is preserved, but less tightly connected modules can be placed in different clusters, allowing the exploitation of the vertical dimension. The core of ClaP is a simulated annealing algorithm with different moves (inter-cluster, intra-cluster, move entire clusters, . . .). For the given example, this vertical stacking of horizontally arranged clusters leads to a delay reduction of ca. 20% over the purely linear solution.

1.5.3 Reconfiguration Scheduling

The kernels currently extracted by Comrade from real C applications (without exploiting alias or loop iteration space analysis) have a maximum size of a few hundred operators and often take up just 3000. . . 6000 cells (4-LUT + flip-flop) of RCU area. Considering that even medium-sized commodity FPGAs have more than 17,000 cells, a study of how to use all of this space has much potential.



(a) One kernel per configuration



(b) Multiple kernels per configuration

Fig. 1.18 (a) Initial and (b) new on-chip architecture

Figure 18(a) shows the initial RCU architecture generated by Comrade. It consists of interfaces to the rest of the system (a slave to exchange live variables with the SPP and a master to perform independent memory accesses) and the controller / datapath for one kernel.

Orthogonally to trying to extract more complex kernels from the C programs (see Sect. 1.7), we can employ the unallocated RCU area to hold the controllers/datapaths for *different* kernels on the RCU. In this manner, we avoid potentially very slow reconfigurations and simply *switch* between kernels that have been merged into a single configuration. Such an architecture is shown in Fig. 18(b). The SPP can now initiate an appropriate switching of the kernel bus multiplexer by performing a slave mode write to the RCU. Thus, a switch between different kernels requires just a single clock cycle instead of the many milliseconds of a reconfiguration.

While Comrade does not yet exploit this scheme, we have already developed the algorithms for computing how the kernels should be merged into configurations so that the number of reconfigurations is minimized over the entire program execution [17]. For rapid calculation of an estimated solution (independent of the detailed execution trace of the program), we use a heuristic that constructively builds clusters of configurations following the nested loop structure of the program. To evaluate the quality of the heuristic and to compute the optimal solution when algorithm run-time is less critical, an alternate exact approach using dynamic programming was also designed. Since the exact approach evaluates the complete trace of kernel execution order during a program run, its own run-time can be lengthier for more complex programs. In general, the heuristic computed results close to the optimal solution for our experiments, but required less than 1/10 s execution time. The optimal algorithm generally had run-times between 1/10 s and 100 s.

The results of configuration merging are extremely promising. From the Wavelet image compression application Versatility [33], we can extract eight kernels. When using separate configurations for these kernels, the execution of the hybrid hardware/software application will require 5381 reconfigurations. After performing configuration merging, the heuristic will reduce that to five reconfigurations, while the optimal solution can even get it down to just four reconfigurations. Given that reconfiguration times in many cases dominate the entire application run-time, configuration merging will be an essential part of a refined compile flow.

1.6 Lessons Learned

The sheer implementation complexity of a compile flow from a high-level language down to hardware is tremendous. Even for pure software compilers, the significant infrastructure requirements to get a flow working at all (not innovating yet) are considered serious impediments to current research by notable compiler experts [12]. The efforts to bring up a software compiler are dwarfed by the *additional* hardware architecture and design tasks necessary for a *hardware/software* co-compiler such as Comrade. The compiler part alone of Comrade currently consists of more than 300,000 lines of C++ and Java code, excluding the Stanford SUIF2 compiler framework [35] which is used as the front-end. Also not included are the system interfaces (SPP, memory, including caching and streaming) that were formulated in Verilog and VHDL and the operating system modifications (see Sect. 1.3). Our research continues to be significantly hampered by our unfortunate choice of SUIF2 as the base for the compiler. While SUIF2 was being hailed as a future-proof successor of the earlier SUIF1 system, which was successfully used in the Nimble project [29], SUIF2 development quickly faltered and even today the system does not have all of the capabilities of its less modular predecessor. In retrospect, we should have chosen the much more capable Open64 system [11], which was already available at the project's start, but was discarded due to its convoluted implementation and steep learning curve Today, more alternatives are available: Open64 continues to be developed and is very powerful especially in the area of parallelizing loop transformations. While the more recent LLVM [27] is not yet as advanced as Open64 in that area, it offers a cleaner internal structure and a well-integrated compile flow encompassing many scalar optimizations and even just-in-time compilation. Especially the later would allow for interesting research in just-in-time hardware generation.

The choice of a suitable intermediate representation for the RCU part of the compile flow also turned out to be pivotal for the project. Existing representations, both for pure software (e.g., SSA-CFG) or pure hardware (e.g., DFG and CDFG) lacked the expressive power to handle the complex RCUs capable of accessing memory in their own, which were our aim with Comrade. While the CMDFG has fulfilled that role admirably (we have yet to discover practical limitations), it took much experimentation (e.g., with DFC-SSA and Array-SSA forms) to develop it. For similar research, we highly recommended to spend significant effort developing an IR suitable for *all* phases of the project before beginning to work on individual passes.

Analogously, the CMDFG execution semantics and their realization in the dynamic CoCoMa controller have also advanced the project significantly. In contrast to pure DSP or scientific computing C code, the general purpose C code we are aiming to compile is not amenable to efficient static scheduling: In much practical code, simply too many data, control and (possibly hidden) memory dependencies exist. Modern out-of-order SPPs handle such temporally distributed programs by discovering these dependencies at run-time and re-ordering execution around them. The CMDFG model allows a similar approach for our spatially distributed computations: The CoCoMa dynamic scheduling resolves dependencies at run-time (but see Sect. 1.7).

As described in the preceding Sections, our core focus is still the compiler and the ACS architectures supporting its execution model. However, when compiling more complex programs potentially containing many RCU kernels (something we have not done yet), it is obvious that the reconfiguration overhead on current reconfigurable devices (generally FPGAs without configuration caches etc.) will become excessive and most likely negate any possible speed-up actually achievable by RCU execution. At this stage, our currently under-utilized work on reconfiguration scheduling will become crucial for high performance.

Over the course of the project, we have worked predominantly with Xilinx FPGAs. Starting with the XC4085XL in 1998, we progressed through the Virtex XCV1000 (our long-time work horse), Virtex II Pro and now Virtex 5 FX FPGAs. Up to and including the Virtex device, we managed to keep our physical design tools (floorplanning, pre-placed module generation) synchronized with the latest FPGA features. However, with FPGA complexity increasing (more and more special purpose blocks becoming available), and the quality of conventional logic synthesis and design implementation tools finally improving (e.g., good support for physical synthesis), we believe that it is no longer practical for academic research to cover the entire flow from high-level input language down to layout generation within a single project. We have thus downsized our own efforts in the physical design area (which achieved speed-ups in the 20...30% range) to focus on the compiler's optimization and hardware-generation stages (where we often see speed-ups of 2x or more). In practice, this means that the compiler will no longer generate pre-placed floorplanned EDIF netlists for the RCU (datapath, controller, system interface), but instead stop at a RTL description that is exported to a commercial logic synthesis system for actual mapping.

1.7 Future Work

Now that the compile flow and the associated ACS architecture are fully operational, we can let the initial results of front-to-back compilation experiments guide our future research (some of which has already started).

First, it becomes obvious that we should move larger parts of an input program to the RCU in order to profit from acceleration. While our fine-grained execution model does support this already, the scope of the current SPP/RCU partitioner turns out to be to restrictive. Instead of the current profile-based inlining / profile-based path construction, we are currently implementing full-scale path profiling to discover critical paths flowing through multiple functions [26] without the need for inlining. These whole-program paths can then be used as the basis for RCU kernels.

Second, we need to improve the degree of ILP in the generated hardware. While our current CMDFG-to-CoCoMa translation does support speculation within loop iterations, we do not yet speculate across iteration boundaries. To resolve this, we can introduce *token queues* that allow the computation of predecessor nodes (e.g., the loop index calculation) to re-start in a new iteration even though not all of their successors have consumed their tokens (and the associated data items) yet. Different branches of the CMDFG can thus execute different loop iterations. Other techniques which will be investigated are loop transformations for increased parallelism (using the different cost models of spatial computation, e.g., availability of many registers compared to SPP) and speculative memory accesses (both read and write). The latter capability does require active support from the memory system. Thus, we will continue our research in ACS architecture issues.

1.8 Conclusions

With the Comrade compiler, our DFG project demonstrates the feasibility of compiling from general-purpose ANSI C into hybrid applications, executing both in software on a conventional processor and hardware-accelerated kernels on a reconfigurable compute unit. The flow draws on our advances in compiler construction, high-level synthesis, computer architecture and physical design automation to achieve this goal. Now that the system is functional and can be evaluated on real hardware fully supporting its execution model, we can proceed to actually improve the quality of results. We have already discovered a number of research areas, ripe with potential for significant improvements, that will be tackled in the next stage of work.

Acknowledgements This work was supported by DFG grant Ko 1721/1-2, project *Adaptive Rechensysteme und ihre Entwurfswerkzeuge*, associated with the Priority Programme 1148, *Reconfigurable Computing Systems*.

References

1. Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2. TIS Committee (1995)
2. Bellows, P., Hutchings, B.: Jhdl - an hdl for reconfigurable systems. In: FCCM '98: Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines, p. 175. IEEE Computer Society, Washington, DC, USA (1998)
3. Budiu, M.: Spatial computation. Ph.D. thesis, Carnegie Mellon University, Computer Science Department (2003). Technical report CMU-CS-03-217
4. Callahan, T.J., Hauser, J.R., Wawrzynek, J.: The Garp architecture and C compiler. Computer **33**(4), 62–69 (2000). DOI http://dx.doi.org/10.1109/2.839323
5. Campbell, P.L., Krishna, K., Ballance, R.A.: Refining and defining the program dependence web. Cs93-6, University of New Mexico, Albuquerque (1993)
6. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst. **13**(4), 451–490 (1991). DOI http://doi.acm.org/10.1145/115372.115320
7. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. **9**(3), 319–349 (1987)
8. Gädke, H., Koch, A.: Accelerating speculative execution in high-level synthesis with cancel tokens. In: Proc. Intl. Workshop on Applied Reconfigurable Computing (ARC) (2008)
9. Gädke, H., Stock, F., Koch, A.: Memory access parallelization in high-level language compilation for reconfigurable adaptive computers. In: (Proc. Intl. Conf. on Field Programmable Logic and Applications (FPL) (2008)
10. Gajski, D.D., Ramachandran, L.: Introduction to high-level synthesis. IEEE Design and Test **11**(4), 44–54 (1994). DOI http://dx.doi.org/10.1109/54.329454
11. Group, O.S.: Open64 – the open research compiler. URL http://www.open64.net/
12. Hall, M., Padua, D., Pingali, K.: Compiler research: the next 50 years. Commun. ACM **52**(2), 60–67 (2009). DOI http://doi.acm.org/10.1145/1461928.1461946
13. Kasprzyk, N.: COMRADE – Ein Hochsprachen-Compiler für Adaptive Computersysteme. Ph.D. thesis, Technische Universität Braunschweig (Germany) (2005)
14. Kasprzyk, N., Koch, A.: Verbesserte Hardware-Software Partitionierung für Adaptive Computer. In: Proc. Conference on Architecture of Computing Systems (ARCS) (2004)
15. Kasprzyk, N., Koch, A.: High-level-language compilation for reconfigurable computers. In: Proc. Intl. Conf. on Reconfigurable Communication-centric SoCs (ReCoSoC) (2005)
16. Kasprzyk, N., Koch, A., Golze, U., Rock, M.: An improved intermediate representation for datapath generation. In: Proc. International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA) (2003)
17. Kasprzyk, N., van der Veen, J., Koch, A.: Configuration merging for adaptive computer applications. In: Proc. Intl. Conf. On Field-Programmable Logic (FPL) (2005)

18. Knobe, K., Sarkar, V.: Array SSA form and its use in parallelization. In: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 107–120 (1998). URL http://doi.acm.org/10.1145/268946.268956
19. Koch, A.: Regular datapaths on field-programmable gate arrays. Ph.D. thesis, Tech. Univ. Braunschweig (Germany) (1997)
20. Koch, A.: FLAME: A flexible API for module-based environments (EIS TR 2004-01). Tech. rep., Tech. Univ. Braunschweig, Dept. of Integrated Circuit Design (E.I.S.) (2004)
21. Koch, A.: FLAME library specification (EIS TR 2004-02). Tech. rep., Tech. Univ. Braunschweig (2004)
22. Kunz, J.: Eine placer-modul-erweiterung für den „universal generator for logic circuits on fpgas". Master's thesis, Tech. Univ. Darmstadt (2007)
23. Lange, H., Koch, A.: Design and system level evaluation of a high performance memory system for reconfigurable SoCi platforms. In: Proc. HiPEAC Workshop on Reconfigurable Computing (2007)
24. Lange, H., Koch, A.: An execution model for hardware/software compilation and its system-level realization. In: Proc. Intl. Conf. on Field Programmable Logic and Applications (FPL) (2007)
25. Lange, H., Koch, A.: Low-latency high-bandwidth hw/sw communication in a virtual memory environment. In: Proc. Intl. Conf. on Field Programmable Logic and Applications (FPL) (2008)
26. Larus, J.R.: Whole program paths. SIGPLAN Not. **34**(5), 259–269 (1999)
27. Lattner, C.: Llvm: An infrastructure for multi-stage optimization. Master's thesis, University of Illinois at Urbana-Champaign (USA) (2002). URL http://www.llvm.org/
28. Laurich, P.: A comparison of hard real-time linux alternatives (2004). URL http://www.linuxdevices.com/articles/AT3479098230.html
29. Li, Y., Callahan, T., Darnell, E., Harr, R., Kurkure, U., Stockwood, J.: Hardware-software co-design of embedded reconfigurable architectures. In: DAC '00: Proceedings of the 37th conference on Design automation, pp. 507–512. ACM, New York, NY, USA (2000)
30. Lysaght, P., Rosenstiel, W. (eds.): New Algorithms, Architectures and Applications for Reconfigurable Computing. New Algorithms, Architectures and Applications for Reconfigurable Computing (2005)
31. Mitlehner, S.: Portierung eines java-frameworks zur erzeugung von paramterisierten hardware-strukturen auf eine neue generation von rekonfigurierbaren logikbausteinen. Master's thesis, Tech. Univ. Darmstadt (Germany) (2008)
32. Neumann, T., Koch, A.: A generic library for adaptive computing environments. In: International Conference on Field Programmable Logic and Applications (FPL) (2001)
33. Ponnuswamy, K.P., Kumar, S., Pires, L., Ponnuswamy, S., Nanavati, C., Golusky, J., Vojta, M., Wadi, S.: A benchmark suite for evaluating configurable computing systems - status, reflections, and future directions. In: in Proc. ACM/SIGDA Int. Symposium on Field Programmable Gate Arrays (FPGA'00, pp. 126–134 (2000)
34. Rose, G.: Using the microprocessor MMU for software protection in real-time systems. Tech. rep., LynuxWorks, Inc. (2009)
35. Stanford: The SUIF2 compiler system. URL http://suif.stanford.edu/suif/suif2/
36. Thorns, F.: ClaPi – clustering and placement. Master's thesis, Tech. Univ. Braunschweig (Germany) (2002)
37. Wewetzer, C.: A universal generator for logic circuits on FPGAs. Master's thesis, Tech. Univ. Braunschweig (2005)