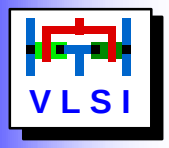


Compilation for Adaptive Computers

Experiences and Opportunities

Andreas Koch
Tech. Univ. Braunschweig
Integrated Circuit Design Unit (E.I.S.)

Outline

- High Level Language to Hardware compilation
- Target architectures
- Compiler processing steps
- Hardware generation
- Reconfiguration strategies
- Conclusion

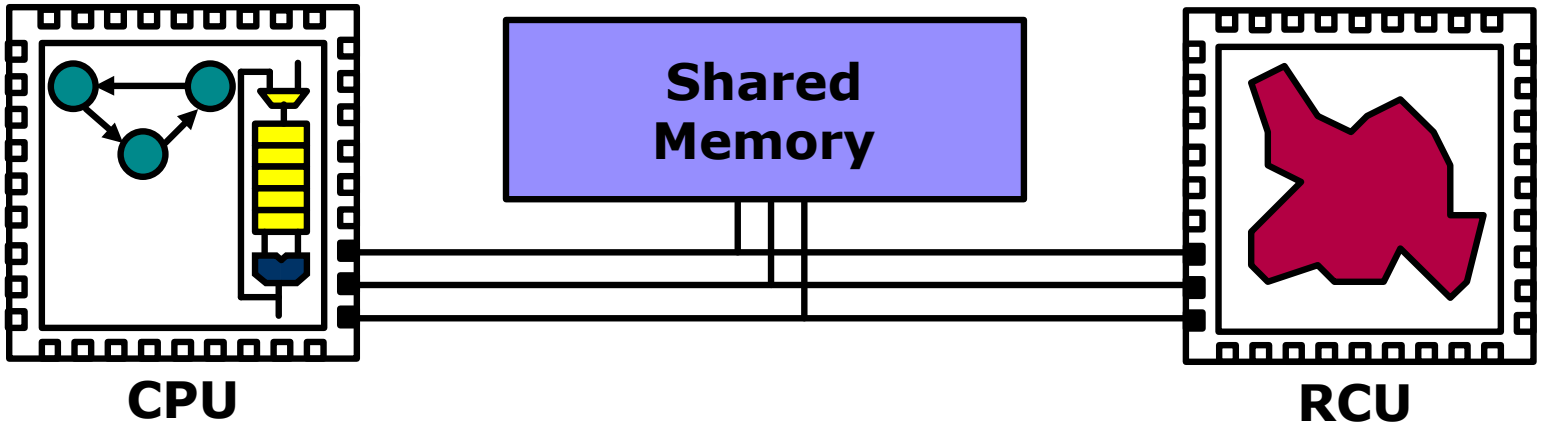
ACS Compilers

- **Experiences described based on work with**
  - **Adaptive Compilers**
    - **Garp CC (Tim Callahan @ UC Berkeley)**
      - Targets Garp simulator
    - **Nimble (joint effort with Synopsys et al.)**
      - Targets ACE-II and ACE-V
  - **Platforms**
    - **Garp (John Hauser @ UC Berkeley)**
      - Simulated, tightly couples MIPS-II core with RCU
    - **ACE-II (TSI-Telsys)**
      - Discrete, loosely couples microSPARC-IIep with 2x XC4085XL
    - **ACE-V (joint effort with Synopsys et al.)**
      - Discrete, loosely couples microSPARC-IIep with XCV1000
- **Outlook on COMRADE**
  - **Compiler under development at E.I.S.**
  - **Planned targets ACE-V and Xilinx ML300 (V2pro)**
    - **... but always looking for more suitable architectures**

HLL Compilation

- High-level software programming languages
  - Few to no extra user annotations required
- Here focus on imperative languages
  - Large user base: C
  - Easier to implement: Fortran, subset of Java
    - No pointers ...
- Concentrate on implementing loops
  - Require bulk of execution-time
- Hardware-infeasible constructs
  - Often floating point, I/O, memory management
  - Choices: skip loop or handle infrequent exceptions
  - Here: exceptions handled by switch to SW

Target Architecture



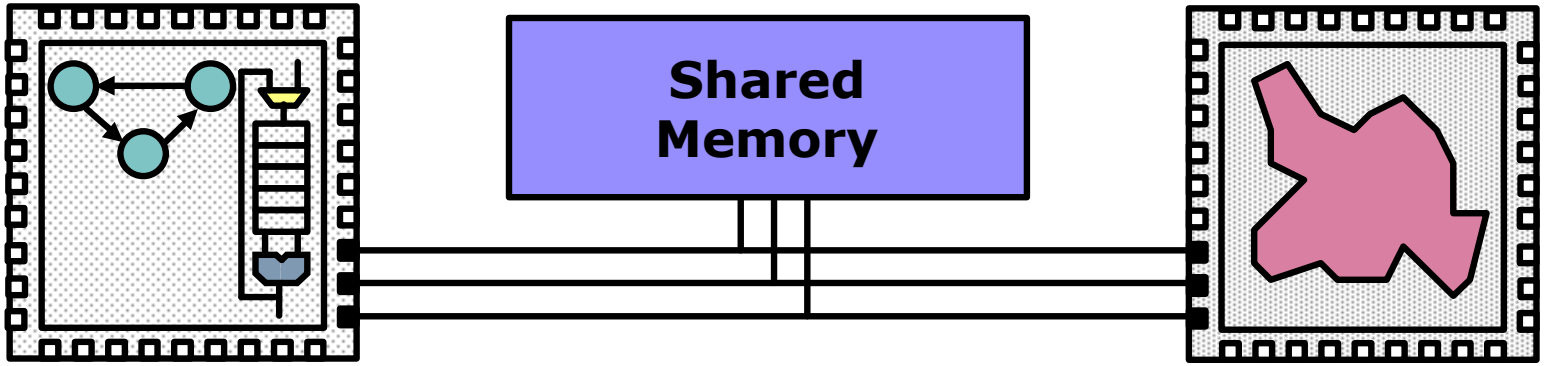
- **Processor**
  - Fixed Function
- **Executes**
  - Irregular sequences
    - System management
  - Hardware-infeasible ops
  - Small part of computation
    - Limited performance / power

- **Reconfigurable unit**
  - Variable function
- **Executes**
  - Regular sequences
  - Bulk of computation
    - Memory access

RCU-CPU Coupling

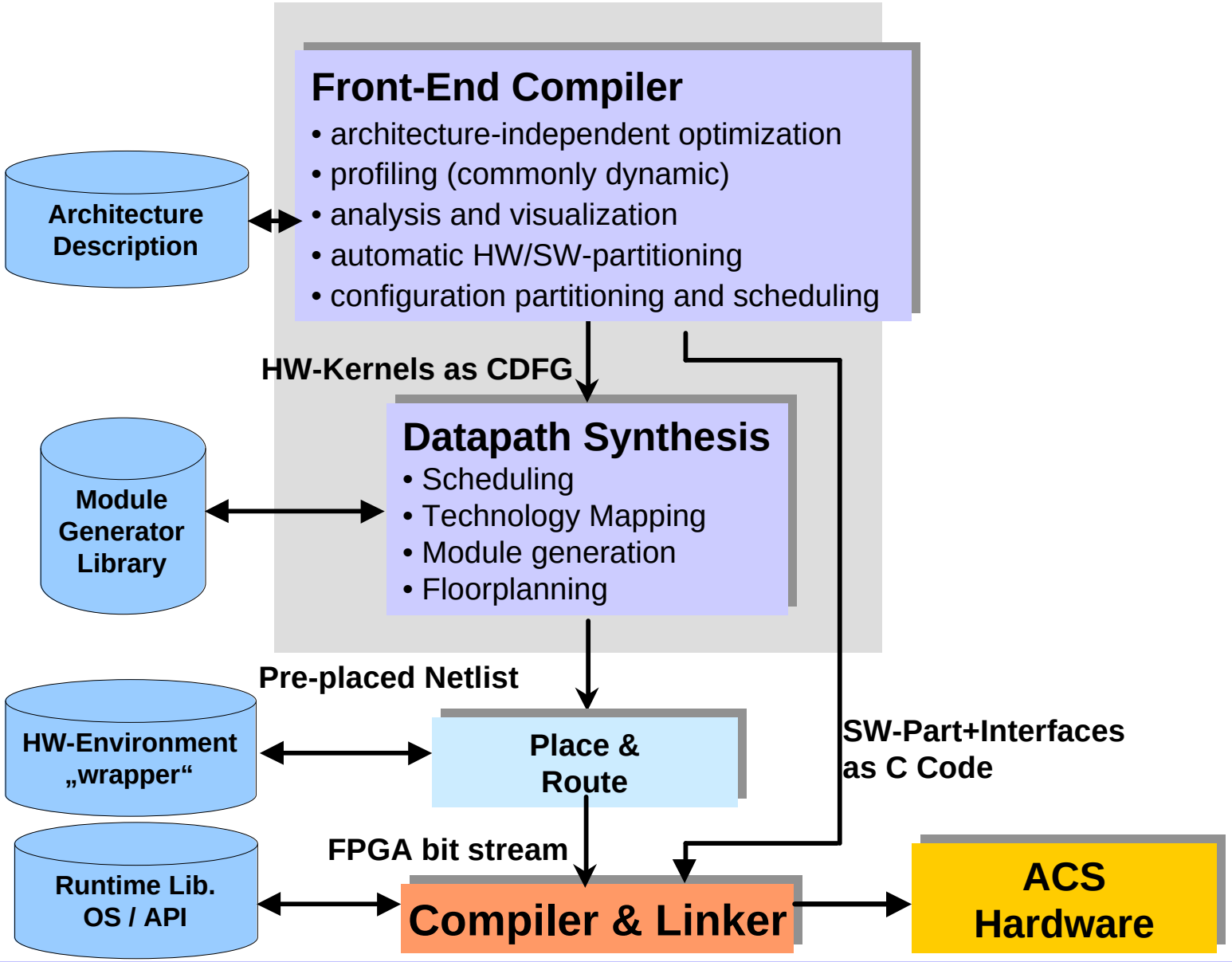
- Tight (short latency communication)
  - Frequent SW/HW switches affordable
    - More exceptions can be tolerated in HW blocks
  - Shorter blocks can be executed on HW
- Loose (long latency communication)
  - Fewer SW/HW switches affordable
  - Blocks must be longer for efficient HW execution
    - Amortize communication overhead over block run-time

Shared Memory



- Zero-copy data transfer between RCU and CPU
- Simplified memory management
  - No "sram_malloc()" etc.
- Homogeneous address space
  - Pointers freely exchangeable between RCU and CPU
- But: possibly cache coherency issues
- Optionally: RCU-local memory

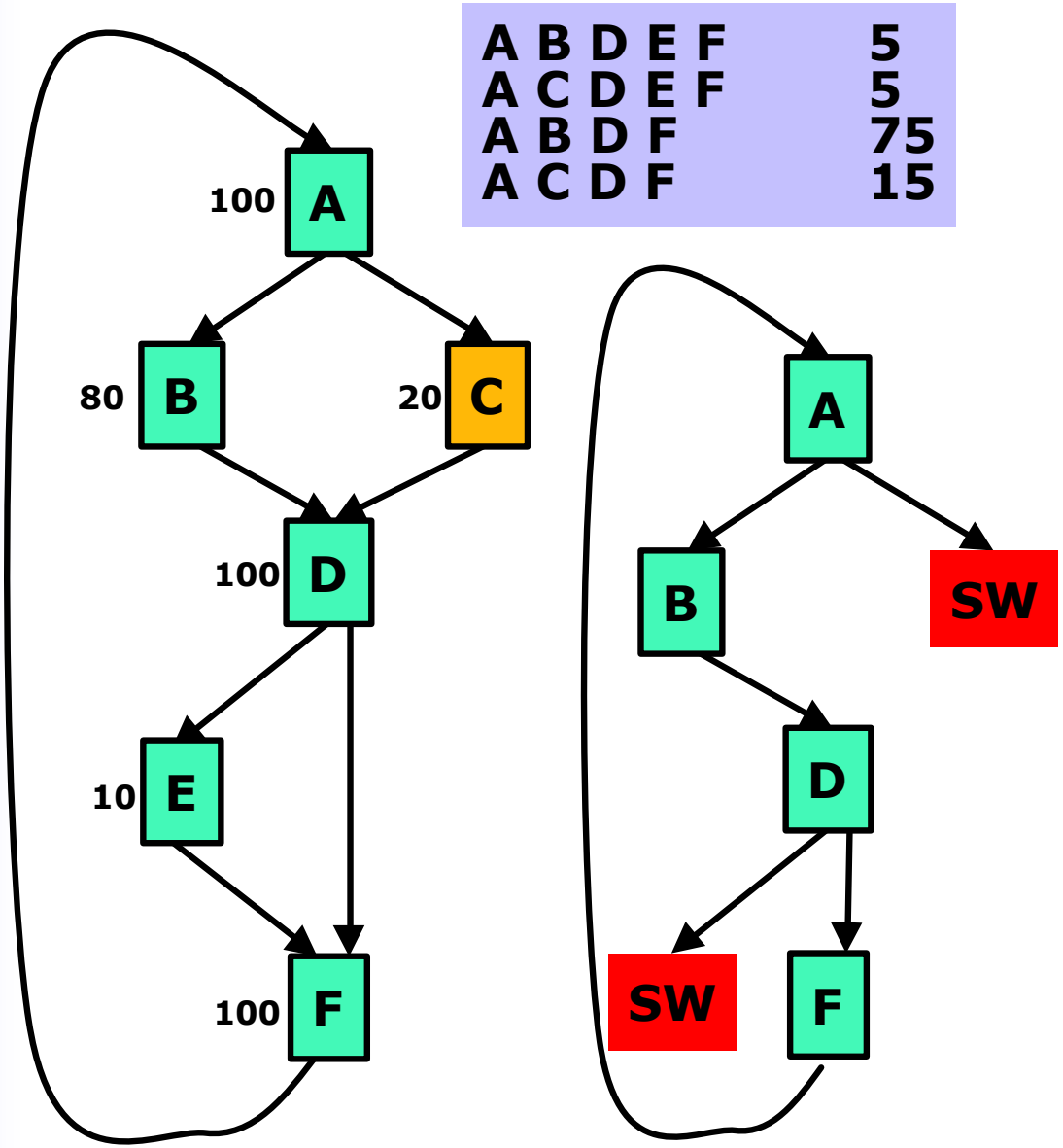
Anatomy of an ACS Compiler



Analysis Techniques

- **Traditional techniques include**
  - Control-flow analysis (recognize loops)
  - Alias analysis (disambiguate pointers)
  - Dynamic profiling (data set dependent)
    - Path profiling (also finds block execution counts) *
    - Performance profiling (block execution times)
- **High hardware relevance**
  - HW/SW-partitioning based on profiling data *
  - Data dependency analysis in loops allows *
    - Parallelization
    - Scalarization
  - Recognize potential use of HW memory streams
- **Reconfiguration Emphasis**
  - Loop Entry Profiling *
  - Loop-Procedure Hierarchy Graph *

Analysis for Partitioning

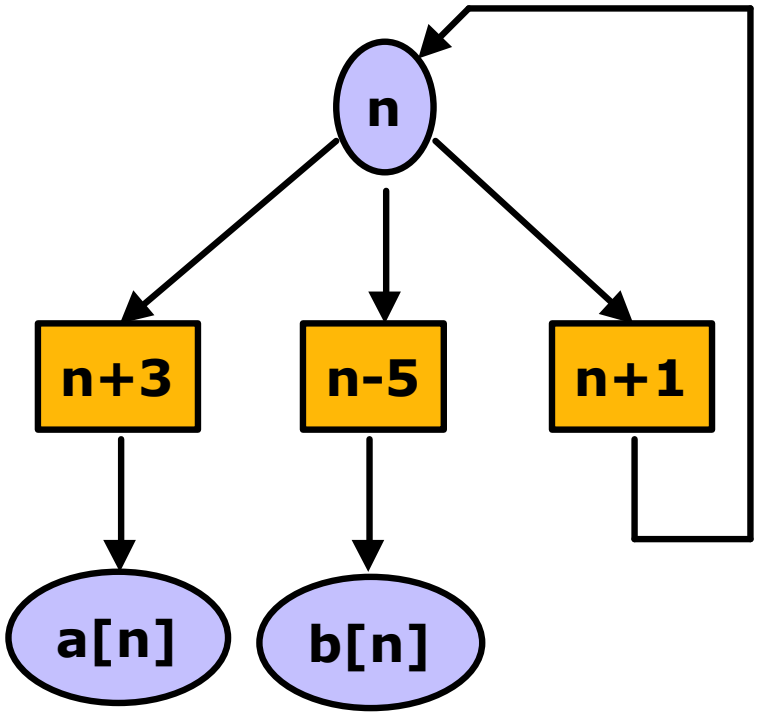


- Relies on path profiling data
  - Block and path execution counts
- Find HW-infeasible constructs (C)
  - If infrequent, handle via SW exception
- Find HW-inefficient constructs (E)
  - If infrequent, prune and handle in SW

Analysis for Parallelization

- Data-dependence analysis for later parallelization
- Dedicated hardware operators
  - Spatially distributed computation

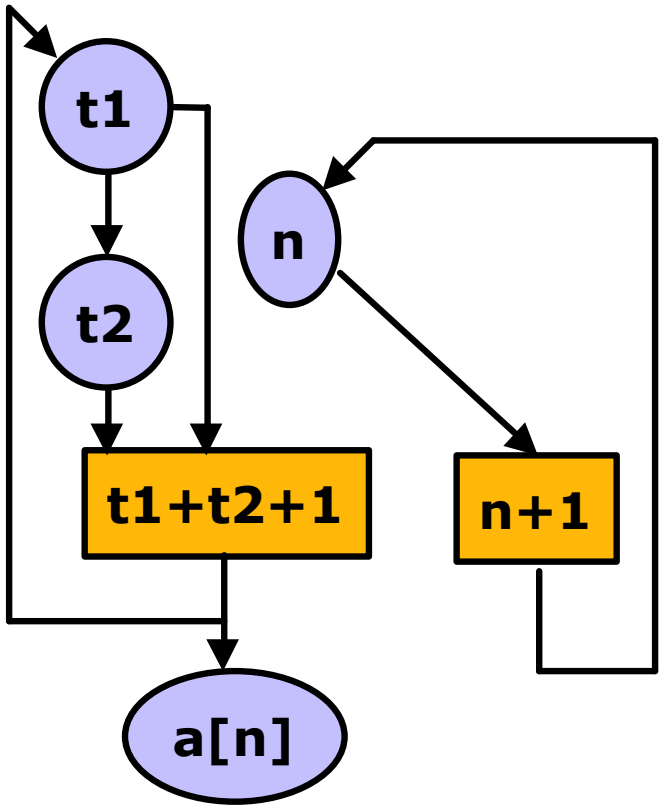
```
for (n=0; n<32; ++n) {
  a[n] = n + 3;
  b[n] = n - 5;
}
```



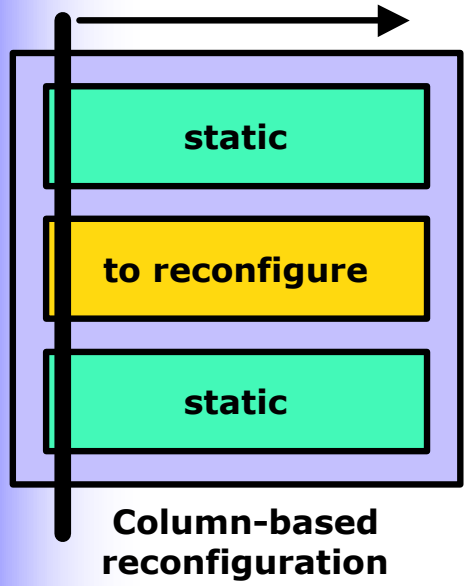
Analysis for Scalarization

- Data-dependence analysis for later scalarization
  - Reduction of memory accesses
- Very efficiently realizable in hardware
  - Multi-tap shift-registers, primed in software

```
for (n=2; n<32; ++n) {
  a[n] = a[n-1] + a[n-2] + 1;
}
```



Analysis for Reconfiguration



Column-based reconfiguration

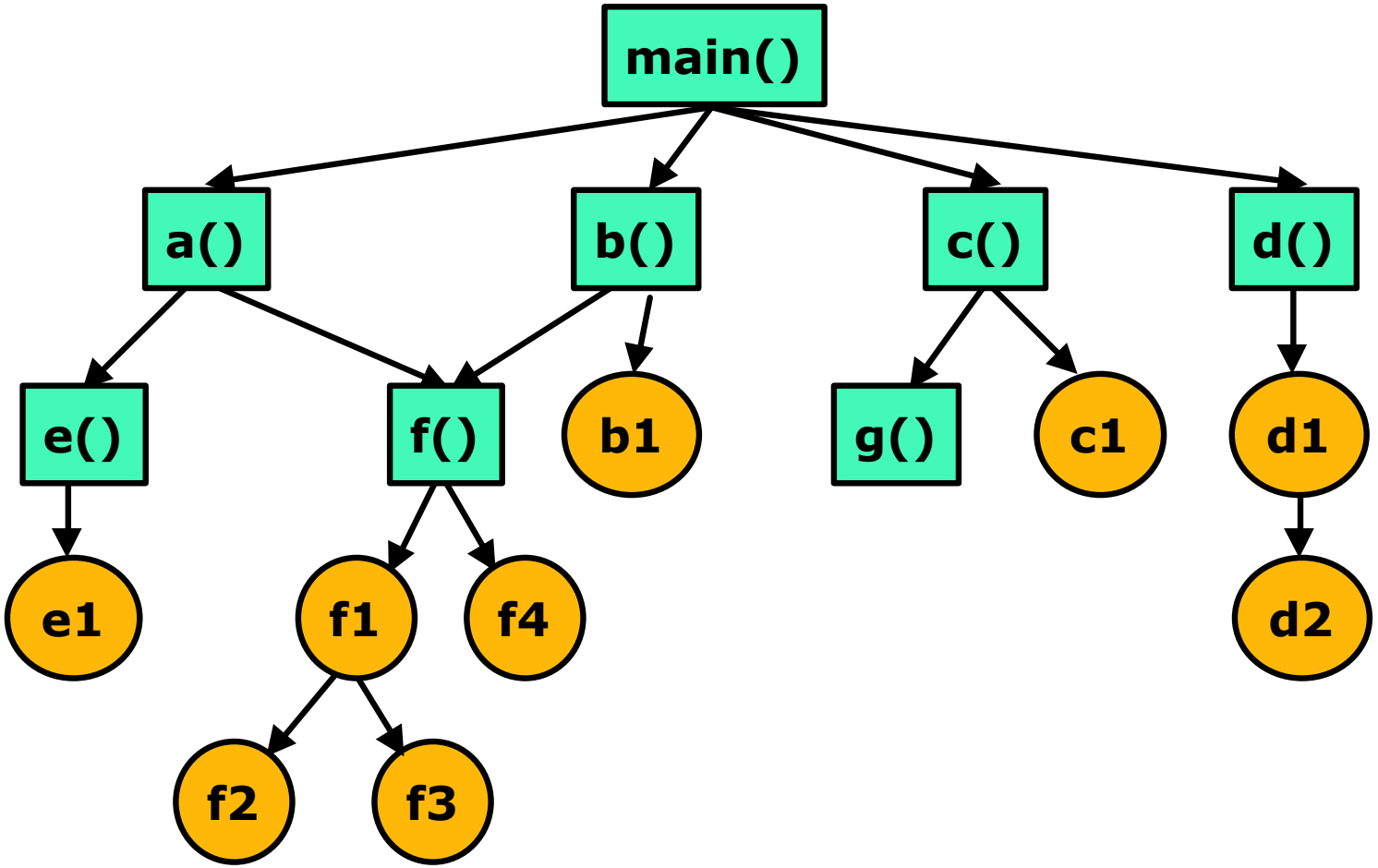
- Kernel: A loop or loop nest
  - Smallest unit considered in partitioning
  - Nimble limitation: Only inner loops
- Questions
  - Which kernels to actually put in HW?
  - Which kernels to put in a configuration?
    - Partial reconfiguration inefficient
    - Applies to many current devices
  - Nimble: Only 1 kernel per configuration
- Naive approach: Put everything in HW
  - Even for fast (10's of cycles) configuration
  - ✗ Slowdown of 10x vs. selective approach
- ➡ Minimize inter-kernel reconfigurations

Selective RCU Use

- Approach in Nimble Compiler
  - Li, Callahan, et al. (DAC2000)
  - Can be generalized beyond Nimble limitations
- Requires two kinds of analysis
  - Static (data-independent)
    - Procedure call tree
    - Loop nesting tree
  - Dynamic (precision depends on quality of input data)
    - Per-block execution time (from profiling)
    - Iteration count for loops (- " -)
    - Loop execution entry sequence

Static Analysis

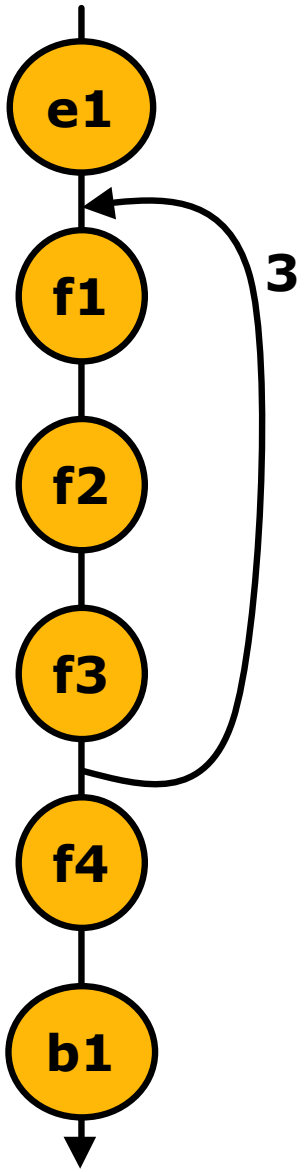
- Loop Procedure Hierarchy Graph (LPHG)
  - Calling structure of functions
  - Nesting structure of loops



Dynamic Analysis

- **Loop Entry Profiling**
  - **Determines temporal order of loop entries**
    - But not iterations!
  - **Dynamic profiling, quality dependent on data set**

Example:
e1 f1 f2 f3 f1 f2 f3 f1 f2 f3 f4 b1



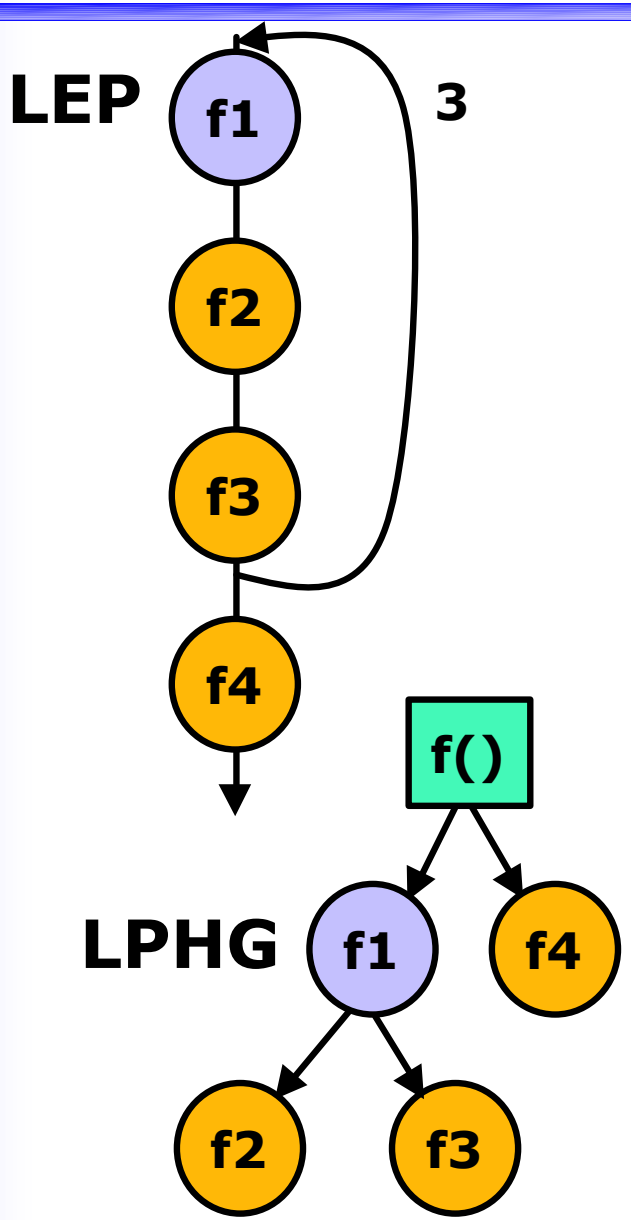
Assign Kernels to RCU

- **Problem:**

  Find program-wide best assignment of *n* kernels to RCU, minimizing reconfigurations

- **Heuristic for tackling this $O(2^n)$ problem**
  - **Cluster all kernels in LPHG sharing predecessor**
    - Assumption: Clustered kernels compete for RCU, no interference between different clusters
    - Predecessor: Outer loop or enclosing function
    - Limit cluster size (e.g., to 5), split larger clusters
  - **For all possible combinations of *cluster* contents**
    - Compute # of required reconfigurations from LEP
  - **Pick per-cluster optimum selection of HW kernels**
    - At least one HW-candidate kernel from each cluster
  - **Each of these kernels will become an RCU configuration**

Nimble Example



On RCU	Reconfigurations
-	0
f1	outer loop, not hw-feasible
f2	1
f3	1
f4	1
f2,f3	6
f2,f4	2
f3,f4	2
f2,f3,f4	7

- **Next step**
  - **Consider total performance**
    - Estimated speed-up by HW loops
    - Slow-down by reconfigurations
  - **Relies on dynamic profiling**

Work in COMRADE

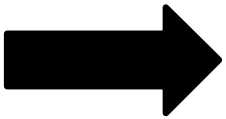
- Nested loops are now valid
- Merge multiple kernels into a single configuration
  - Significantly reduces number of reconfigurations
- Try to preload configurations

Transformations

- **Traditional**
  - Common sub-expression elimination
  - Constant folding and propagation
  - Dead code elimination
- **Hardware relevant**
  - Function inlining *
    - Specialization of constants (especially loop bounds)
  - Loop transformations to expose parallelism
    - Unrolling (increases RCU area requirements)
    - Software pipelining (small to no RCU area growth)
- **Hardware emphasis**
  - Bit-width reduction
  - Unroll & squash *
  - Embedding of external IP blocks
    - Disguised as function calls

Function Inlining

```
for (n=0; n < 4; ++n) {
        f(a, n);
}

f(char *p, int i) {
        p[i] = 2*i;
}
```



```
for (n=0; n < 4; ++n) {
        a[n] = 2*n;
}
```

- Function calls are HW infeasible
  - Prevent HW execution of kernel
- Inlining inserts function code directly at call
  - But calling block can become larger
- Questions
  - What to inline?
  - How deep a function hierarchy to inline?

Improving Inlining

- **Experience from Garp CC**
  - Simple static rules allowing only inlining of leaf functions insufficient
- **Better approach**
  - Should be profiling directed
    - Rely on dynamic call tree and execution time data
  - Only inline at execution time hot-spots
  - Hierarchical inlining for chain of simple functions
  - Recognize “near-leaf” functions
    - Handle rare cases in software
- **Effects**
  - Larger number of kernels (=hardware area)
    - One for each caller
  - Opportunities for specialization by constant propagation

Inlining Example

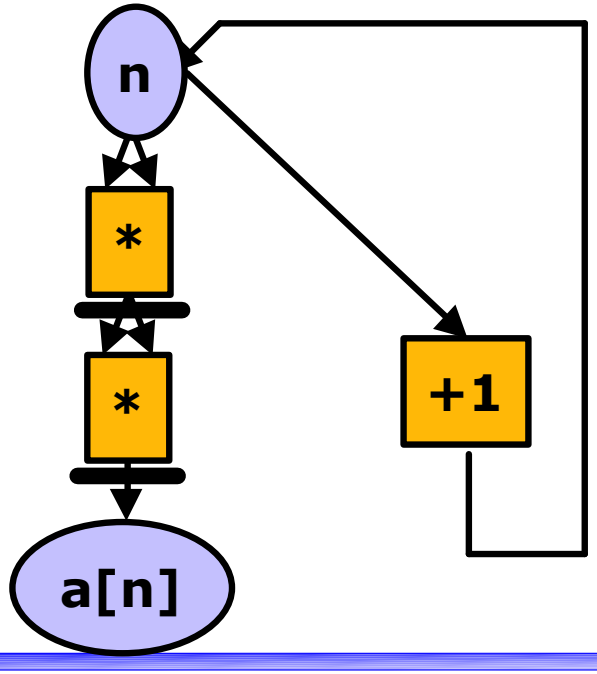
```
for (n=0; n<32; ++n) {
        a[n] = f(n, 4);
}

int f(int i, int j) {
        int k = 1;
        if (j < 0)
                printf("error");
        for (int l = 0; l<j; ++l)
                k *= i;
        return k;
}
```

Inlining "near-leaf" procedures

```
for (n = 0; n < 132; ++n) {
        int k = 1;
        if (4 < 0)
                printf("error");
        for (int l = 0; l < 4; ++l)
                k *= n;
        a[n] = k;
}
```

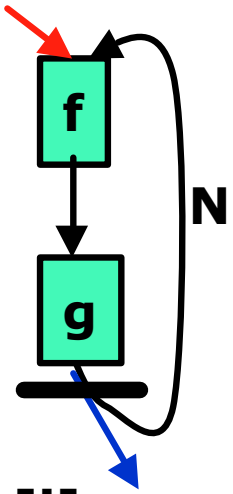
Dead code eliminated
Loop unrolling
Constant propagation
Copy propagation
Algebraic simplification

```
for (n = 0; n < 32; ++n) {
        a[n] = n*n*n*n;
}
```



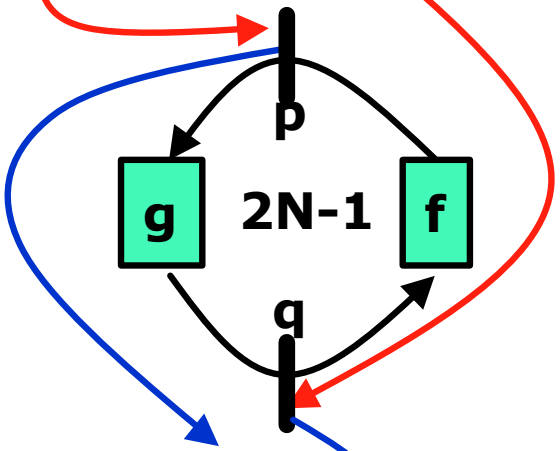
Unroll & Squash in Nimble

```
for (i=0; i<M; ++i) {
        x = a[i];
```




```
        b[i] = x
}
```

```
for (i=0; i < M; i+=2) {
        x1 = a[i]; q=a[i+1];
        p = f(x1);
```



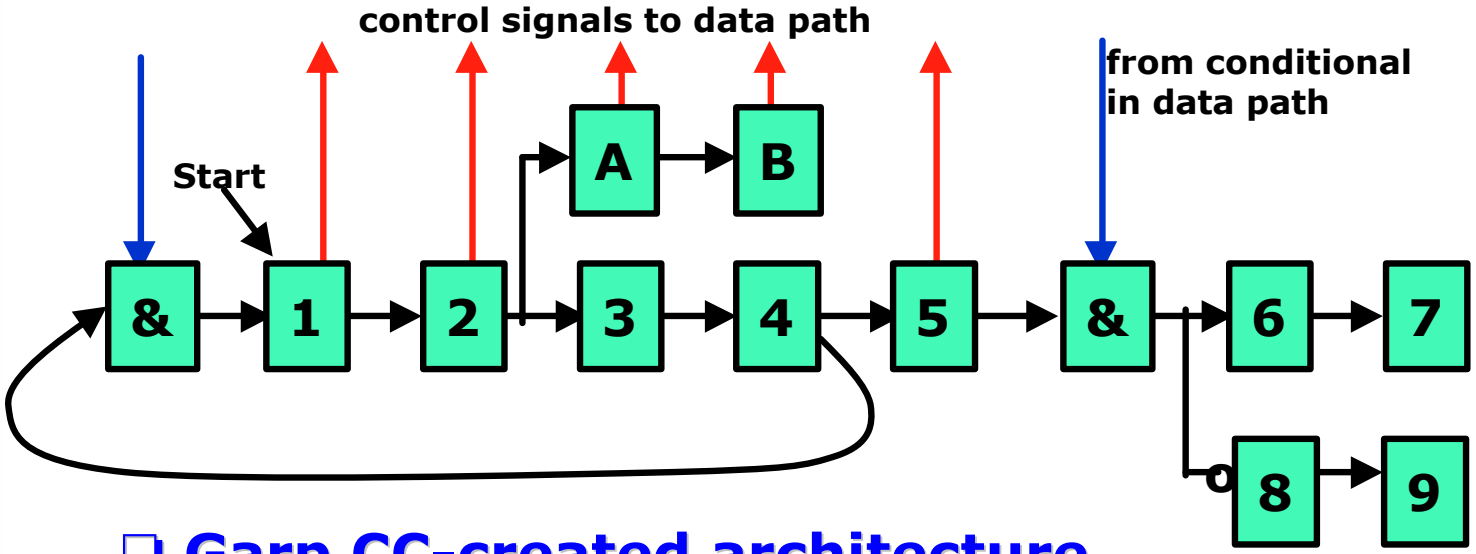

```
        x1 = g(p);
        b[i] =q; b[i+1]=x1
}
```

- Here: Up to 2x performance with same HW area
  - Clocks: 2 M N vs. (M/2) (N*2) = M N
  - by Petkov, idea: Leiserson's c-slow execution

Work in COMRADE

- Profile-based inlining
- Combined loop transformations
  - Software pipelining and unrolling
  - Unroll&squash and unroll&jam
- Bit-width reduction for logical operators
- Exploit novel loop restructuring techniques
  - Aimed specifically at hardware implementation
    - Example: Aggressive Tail Splitting
- Automatic embedding of external IP blocks
  - Interface wrapper generation

Controller



- Garp CC-created architecture
  - N-hot controller (branching shift-register)
  - Allows pipelining
  - Fast and compact
- Limitations
  - Allows only single thread of execution
    - Memory stalls halt entire sequencer
  - Assumes fixed (worst-case) schedule
    - Longest computation path determines decision

Work in COMRADE

- Basic structure unchanged
- Supports fully dynamic schedules
  - More data flow-like
- Short circuit evaluation
  - After condition is valid, wait only for the actually selected computation
- Cancel mis-speculated computation in progress
  - Restart with next set of input values
- But: Much larger and more complex hardware
  - Two nested loops, four conditionals: >50 flip-flops
- To do: Trade-off simple vs. complex controller
  - Possibly complex controller only for innermost loops

Dynamic Reconfiguration

- Exploitable by automatic compilers
  - Plea to vendors: Just give us suitable devices
  - … but there seems to be hope ☺
- Single-cycle reconfiguration not required
  - Due to focus on longer running loops
- … and seems to be rather wasteful
  - 30%-50% expected area increase vs. 10's of cycles
    - John Hauser, architect of Garp
- But configuration caches are effective
  - Mean cache miss rate over 13 real applications
    - Compiled by Garp CC
    - 1 plane: 35%, 4 planes: 4%, 8 planes: 1%
  - Area efficient: 4 → 8 planes = +15% area (Hauser)
    - On Garp: hit=10's of cycles, miss=384 cycles
  - Even better cache usage using improved management

Reconfiguration Strategies 1

- **Established (Nimble via LEP+LPHG)**
  - **One kernel per configuration**
  - **Fully exploit available hardware area for realization**
    - Maximum parallelism and speculation
  - **Only feasible for**
    - Reasonably fast configuration switches (100's of cycles)
    - Or very few kernels actually selected for HW execution
- **Under development (COMRADE, extended LEP+LPHG)**
  - **Allow multiple kernels per configuration**
  - **Compensate for glacial configuration speeds**
    - Suitable for current fine-grained devices (FPGAs)
  - **But trade-off becomes more complex**
    - Number of reconfigurations
      vs. area per kernel (less parallelism and speculation)

Reconfiguration Strategies 2

- **Future: Support for dynamic partial reconfiguration**
  - **Multiple kernels resident on device**
  - **Kernels can be individually loaded**
  - **Profiling data used as hints for kernel pre-loads**
  - **But mispredictions can be dynamically corrected**
    - **All feasible kernels actually *have* HW realizations**
    - **Compare with Nimble/COMRADE: Miss → SW execution**
  - **Novel degree of support in physical design tools**
    - **Estimate time to configure a specific function**
    - **Estimate length of configuration data**
    - **Both dependent on**
      - Complexity of function (hardware area)
      - "Wildcarding" (configuration compression for regular circuits)

Configuration Size

- Becomes problem with growing number of kernels
  - Especially with fast reconfiguration
  - More kernels can be configured onto hardware
  - ... but now more configuration data has to be stored
  - ➡ Problem especially for embedded systems
- Configuration size for XCV1000: 768KB
  - 150-300x 32b operators+control+memory interface
  - Even after LZO compression ~100KB per kernel
  - Garp CC can find 147 HW-feasible kernels in GCC
- Alternatives
  - Denser configurations
    - Garp 32x 32b operators+control+mem.intf.: 6144 Bytes
  - Usable partial reconfiguration
  - "Wildcarding" to describe regular structures
    - Replication of configuration data across a larger area

Conclusion

- **Overview of an ACS Compiler**
  - Hardware effects of traditional steps
  - ACS-specific steps
  - Large as-yet untapped performance potential
- **Dynamic reconfiguration**
  - Automatically exploitable by compiler
  - But currently no practically useful devices
  - Problem of increasing configuration data size

➡ **More suitable device architectures sorely needed**