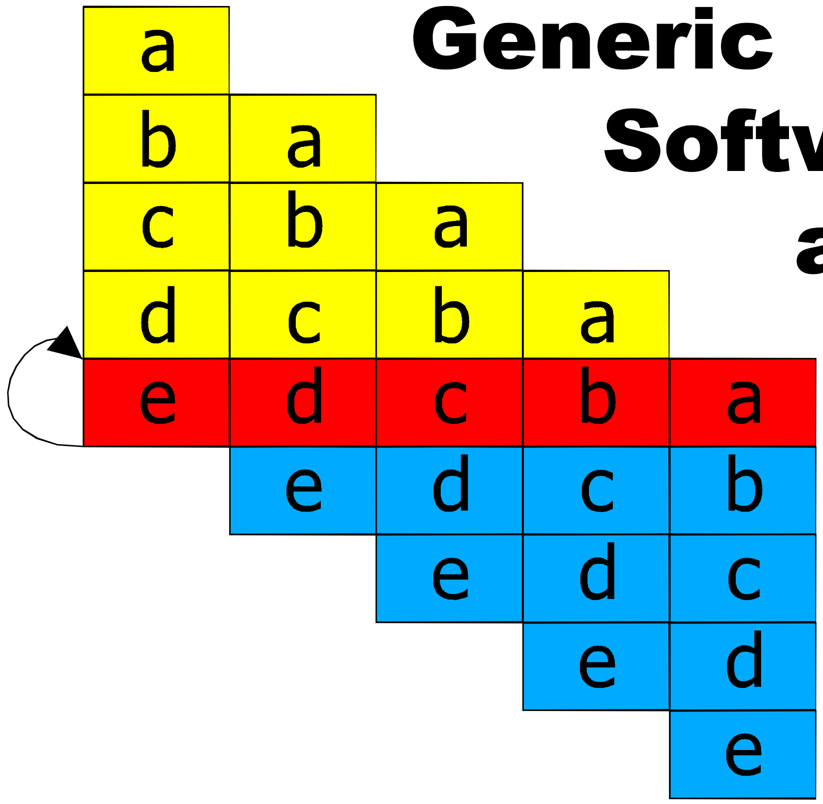
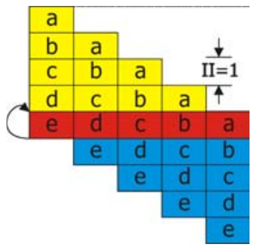
Generic Software Pipelining at the Assembly Level

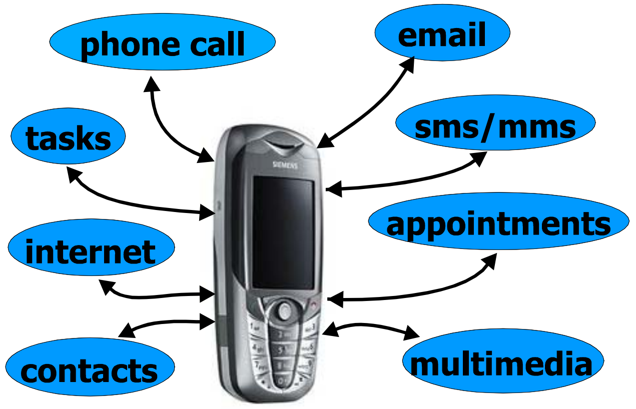


Markus Pister

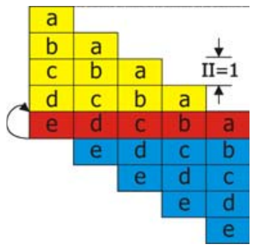
pister@cs.uni-sb.de



Embedded Systems (ES)



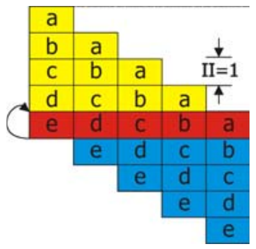
- Embedded Systems (ES) are widely used
  - Many systems of daily use: handy, handheld, ...
  - Safety critical systems: airbag control, flight control system,...
- Rapidly growing complexity of software in ES



Embedded Systems (2)

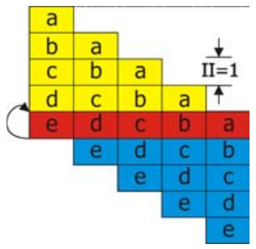
- Hard real time scenarios:
  - Short response time
    - Flight control systems, airbag control systems
  - Low power consumption and weight
    - Handy, handheld, ...

➢ Urgent need for fast program execution under the constraint of very limited code size



Code Generation for ES

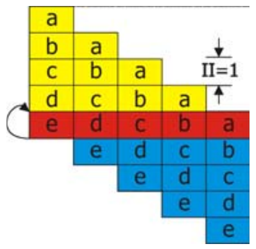
- Program execution times mostly spent in loops
- Modern processors offer massive instruction level parallelism (ILP)
  - VLIW architecture: e.g. Philips TriMedia TM1000
  - EPIC architecture: e.g. Intel Itanium



Code Generation for ES (2)

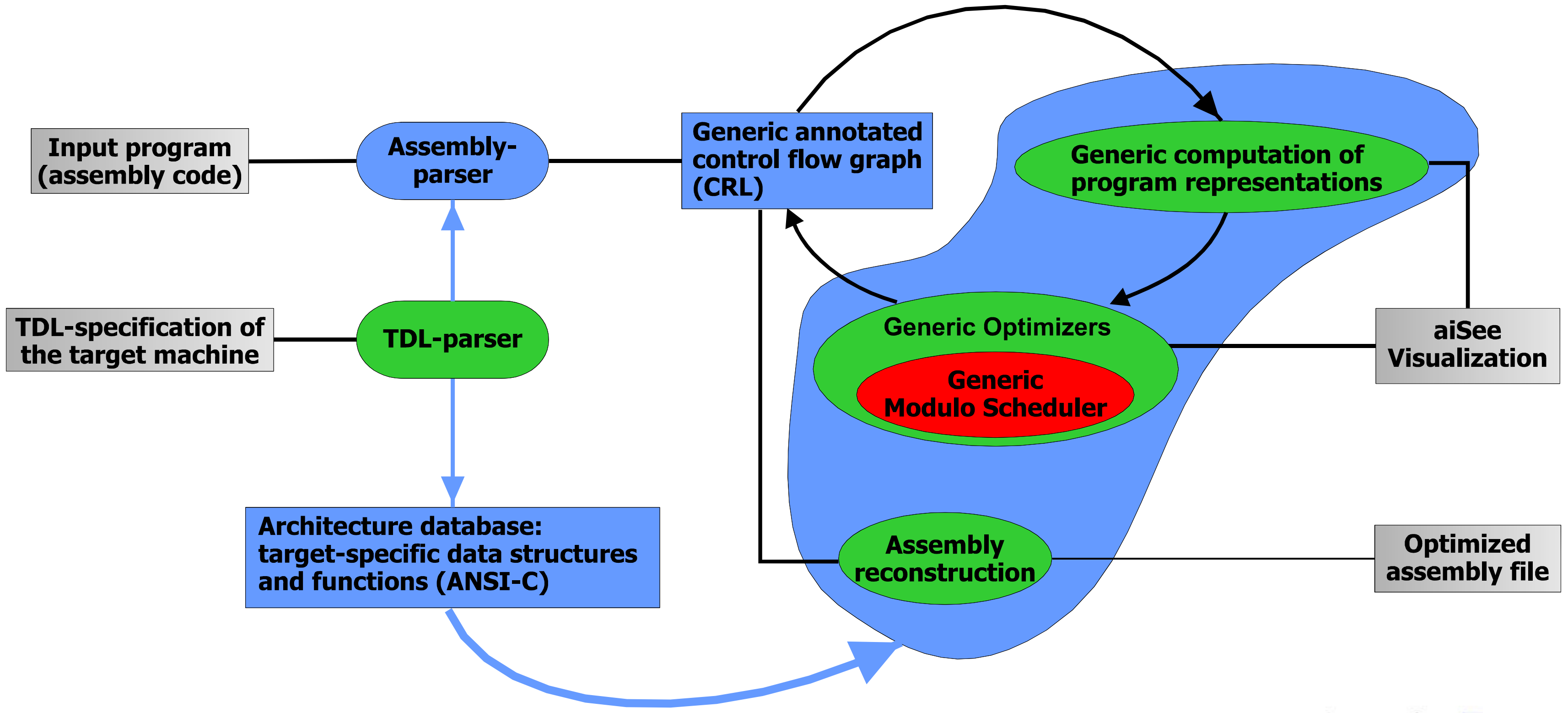
- Many existing compilers cannot generate satisfactory code (cannot exploit ILP)
- High effort enhancing them to cope with advanced ILP

➢ Improving the quality of legacy compilers by
  - Starting at the assembly level
  - Building flexible postpass optimizers
    - Can be quickly retargeted
    - Improve generated code quality significantly

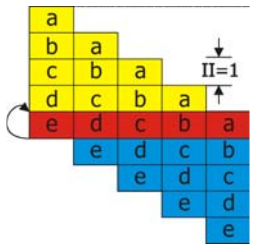


PROPAN-Overview

- Postpass-oriented Retargetable Optimizer and Analyzer



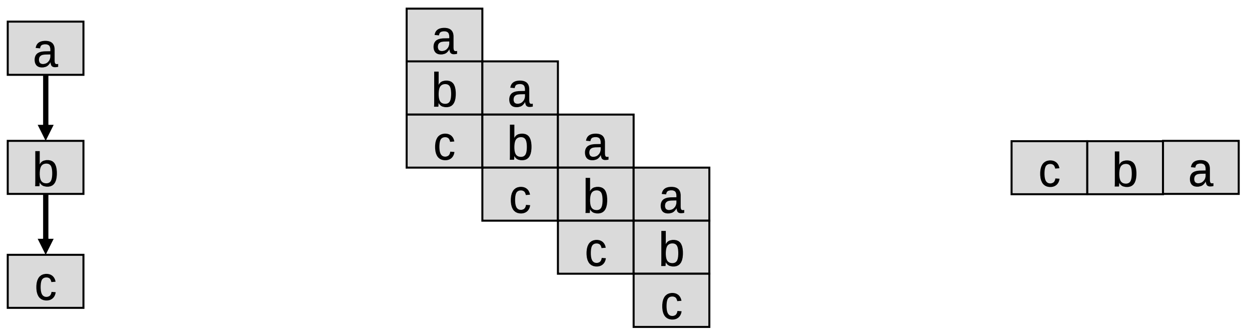
Input program (assembly code)
Assembly-parser
Generic annotated control flow graph (CRL)
Generic computation of program representations
TDL-specification of the target machine
TDL-parser
Generic Optimizers
Generic Modulo Scheduler
aiSee Visualization
Architecture database: target-specific data structures and functions (ANSI-C)
Assembly reconstruction
Optimized assembly file

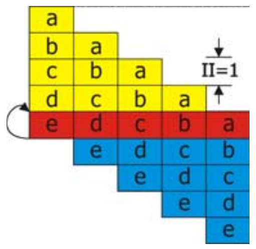


In this talk

- Software Pipelining as a post pass optimization
- ➢ Important technique to exploit ILP while trying to keep code size low
- Static cyclic and global instruction scheduling method
- Idea: overlap the execution of consecutive iterations of a loop

DDG **4x unrolled loop** **Kernel**

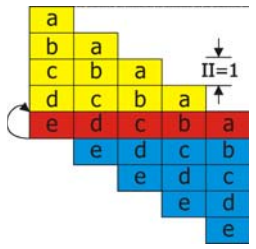




Software Pipelining

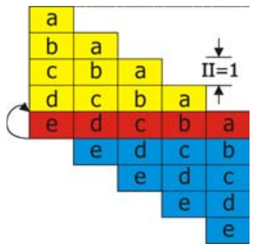
- Computes new (shorter) loop body
- Overlapping loop iterations
- Exploits ILP
- Modulo Scheduling
  - Initiation interval (II)
  - divides loop into Stages
  - Schedule operations modulo II
- Iterative Modulo Scheduling

$$\begin{tikzcd}
 & \text{original loop} \arrow[d] & \\
 & \text{Compute MII} \arrow[d, "II=MII"] & \\
 & \text{Scheduling(II)} \arrow[dl, "success"'] \arrow[dr, "failure"] & \\
\text{Modulo Schedule} & & \text{Increase II} \arrow[ul, bend right]
\end{tikzcd}$$



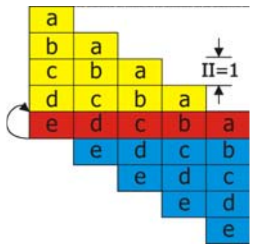
Minimum Initiation Interval

- Resource based: MII_{res}
  - Determined by the resource requirements
  - Approximation for optimal bin packing
- Data dependence based: MII_{dep}
  - Delays imposed by cycles in DDG
- MII = Max (MII_{res}, MII_{dep})
- Basis for Kernel (modulo) computation



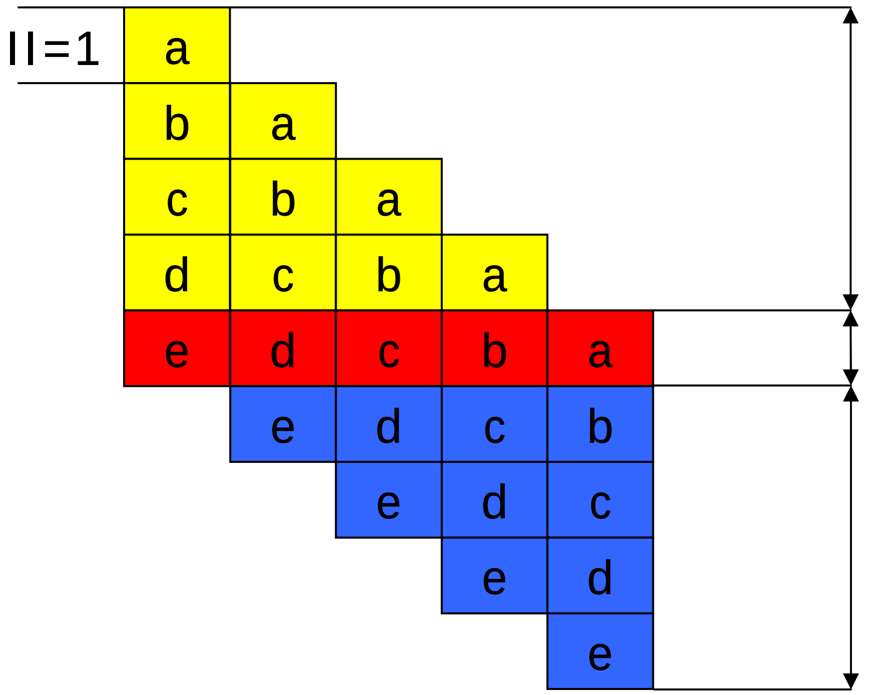
Scheduling Phase

- Flat Schedule
  - Maintain partial feasible schedule
  - Algorithm:
    - Pick next operation
    - Compute slot window [EStart,LStart]
    - Search feasible slot within [EStart,LStart]
    - Conflict: unschedule some operations and force current operation into partial schedule
- Kernel
  - Schedule operations from the Flat Schedule modulo II



Prologue / Epilogue

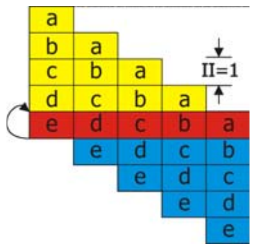
- „fills up“ or „drains down“ the pipeline respectively



Prologue

Kernel

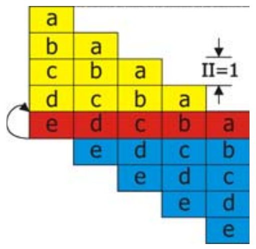
Epilogue



Characteristics of the Post pass approach

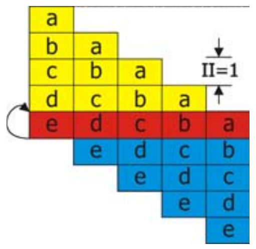
- Integration of the pipelined loop into the surrounding control flow
  - Modification of branch targets needed
- Reconstruction of the CFG is complex and difficult
  - Resolving targets of computed branches/calls and switch tables

```
ld32d(20) r4 → r34

ijmpt r1 r34
```



Characteristics of the Post pass approach (2)

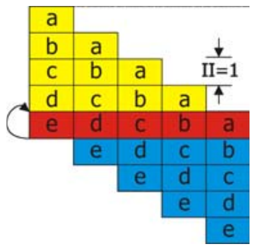
- Register allocation is already done
  - Assignment can be changed with Modulo Variable Expansion
  - Liveliness properties must be checked before register renaming
- Applicable for inline assembly and library code
- Data dependencies at the assembly level are more general
  - More generality leads to a more complex DDG
  - One single array access → multiple assembler operations



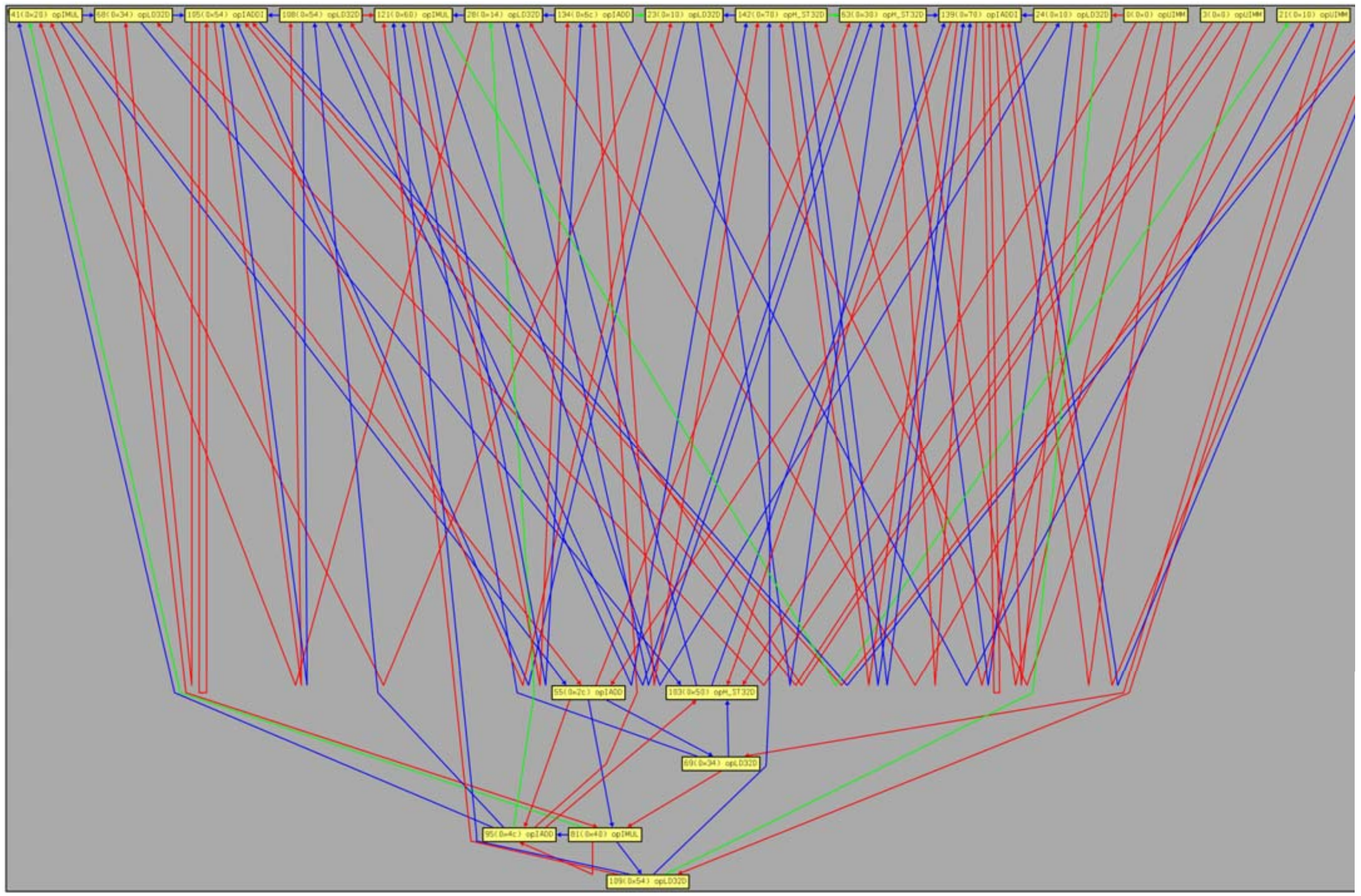
Data dependences at the assembly level

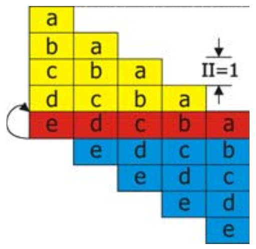
```
i=0;

…

i=i+1;

…

j=array[i];

…
```

```
ld32d(8) r6 → r7

…

iadd(1) r7 → r8

…

ld32d(20) r4 → r10
iadd r10 r8 → r9
ld32d r9 → r11
```



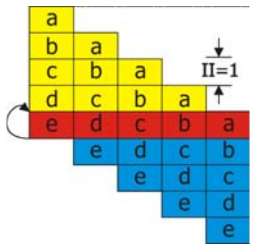
DDG at the assembly level



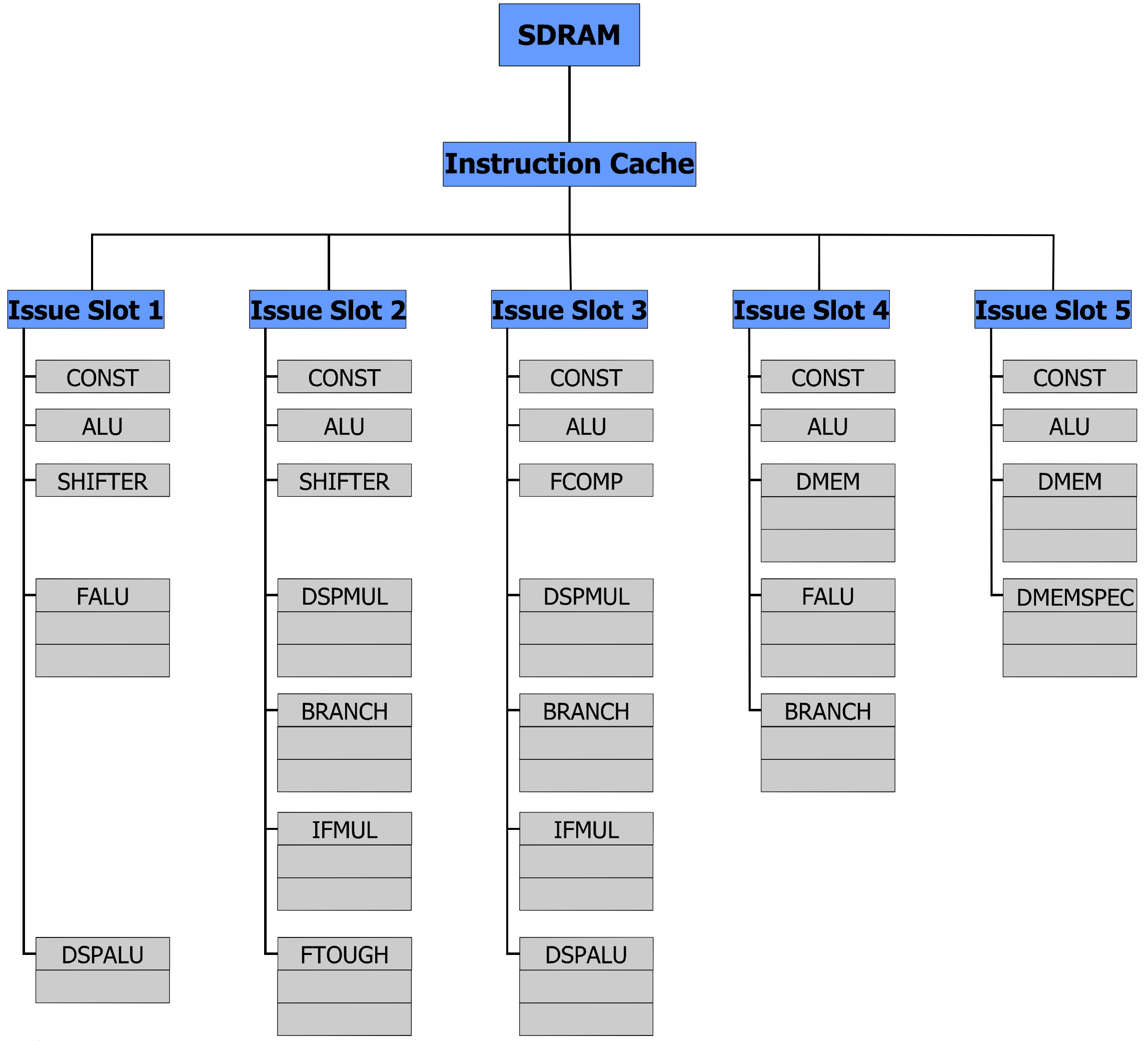


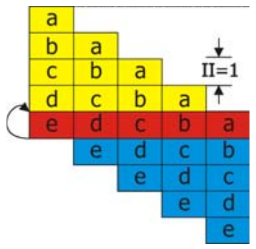
TriMedia TM1000 - Overview

- Digital Signal Processor for Multimedia Applications designed by Philips
- 100 MHz VLIW-CPU (32 Bit)
- 128 General Purpose Registers (32 Bit)
- 27 parallel functional units



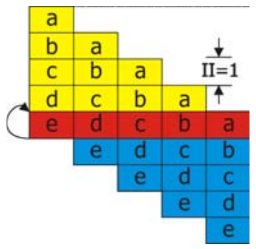
TM1000 – VLIW-Core





TriMedia TM1000 - Properties

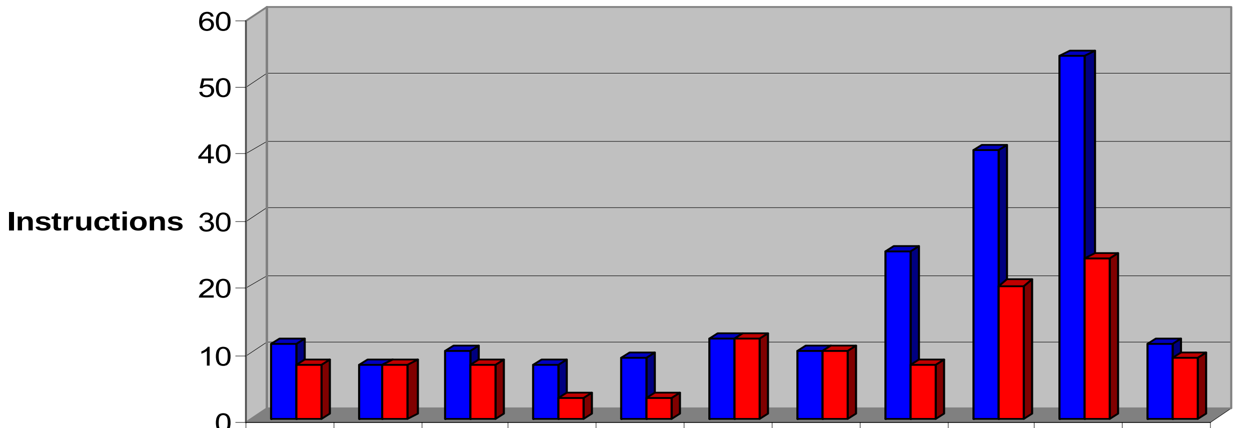
- Instruction set
  - Register-based addressing modes
  - Predicative execution: register-based
  - load/store architecture
  - Special multimedia operations
- 5 Issue Slots, 5 Write-Back Busses
- Irregular execution times for operations
  - ➢Write-Back Bus has to be modeled independently



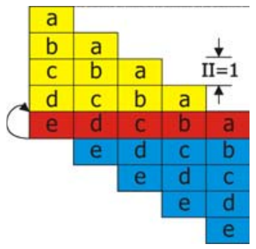
Experimental Results

- Files from DSPSTONE- and Mibench-Benchmark
- Best performance gains for chain like DDG‘s (up to 3,1)

Performance increase for the execution of 100 iterations

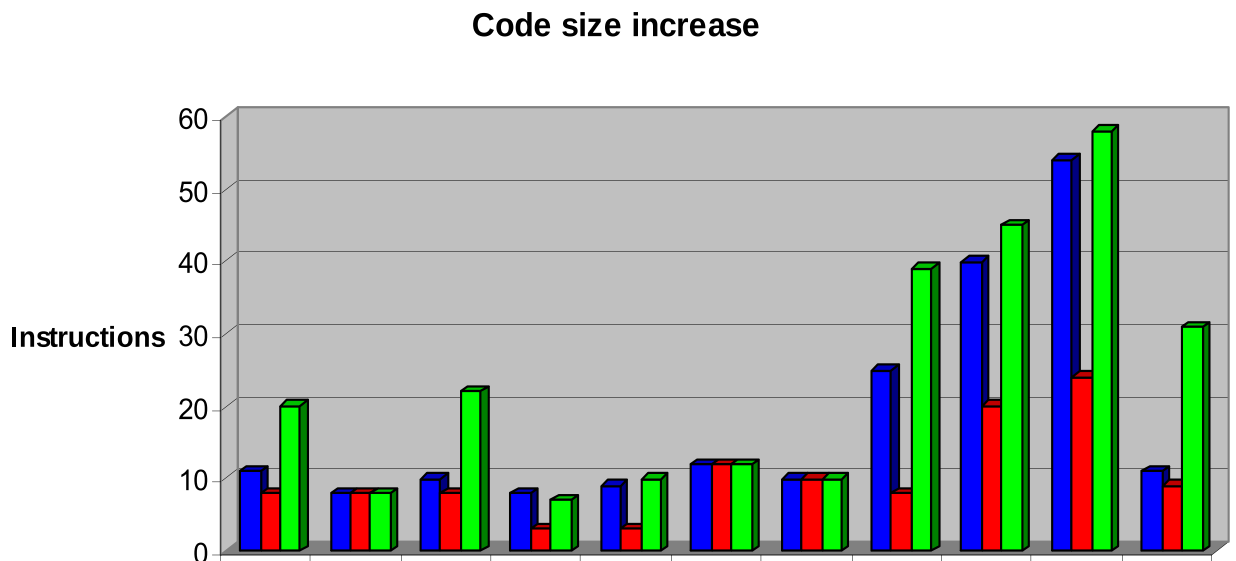


	basic math 1	basic math 2	bitco unt	chain	chain 2	dfir	dlms	dmat 1x3	dmat rix1	FFT	mam u2
🟦 Original	1100	800	1000	800	900	1200	1000	2500	4000	5400	1100
🟥 Pipelined	796	792	798	292	295	1200	1000	799	1985	2386	895

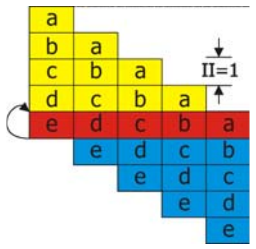


Experimental Results (2)

- Moderate code size increase (average: 1,42)



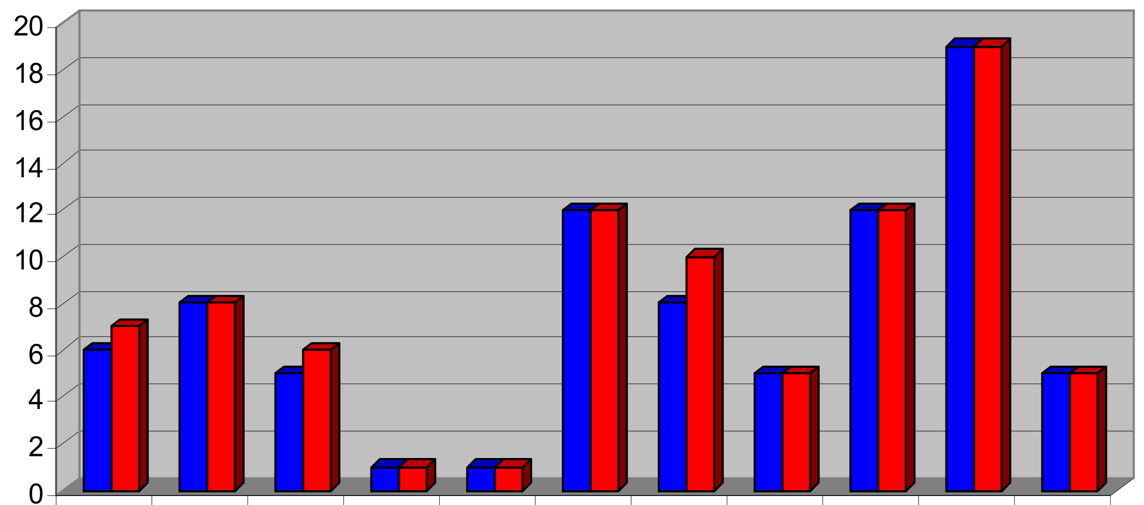
	basic math1	basic math2	bitcou nt	chain	chain2	dfir	dlms	dmat1 x3	dmatri x1	FFT	mamu 2
🟦 Original	11	8	10	8	9	12	10	25	40	54	11
🟥 Kernel	8	8	8	3	3	12	10	8	20	24	9
🟩 Overall	20	8	22	7	10	12	10	39	45	58	31



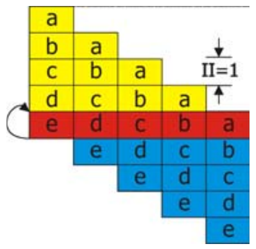
Experimental Results (3)

- Computed MII mostly is already feasible (73%)

Feasibility of the MII

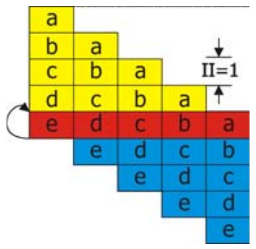


	basic math1	basic math2	bitcou nt	chain	chain2	dfir	dlms	dmat1 x3	dmatri x1	FFT	mamu 2
🟦 MII	6	8	5	1	1	12	8	5	12	19	5
🟥 II	7	8	6	1	1	12	10	5	12	19	5



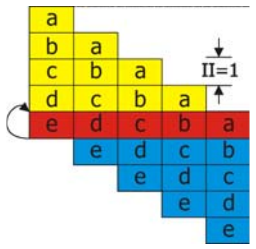
Future Work

- Nested loops:
  - Process loops from innermost to outermost one
  - Treat an inner loop as one instruction (“meta-instruction”)
- Parallelize Prologue and Epilogue code with surrounding code
  - Can be done by existing acyclic scheduling techniques like list scheduling
- Delay Slot filling



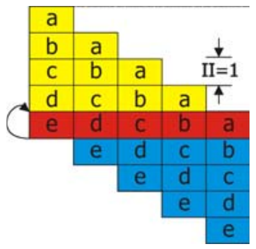
Conclusion

- Embedded Systems creates need for fast program execution under constraint of very limited code size
- Overcome limitation of existing compilers by retargetable postpass optimizer
- Fast program execution by exploiting ILP with Software Pipelining
- Iterative Modulo Scheduling at the Assembly level
  - Characteristics of the Postpass approach
- Experimental results show
  - a speedup of up to 3,1 with
  - an average code size increase of 1,42



Hardware-Support

- Predicated execution
  - ➢possible to omit prologue and epilogue
- Rotating Register Files
  - ➢No Modulo Register Expansion needed
- Speculative Execution
  - ➢Arbitrary number of iterations possible without a copy of the original loop at the expense of code size

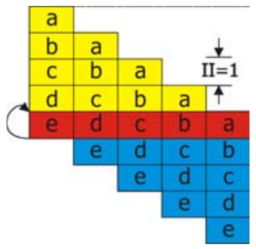


Slot Window

- Early Start
  - Earliest possible schedule time w.r.t. to the data dependencies

$$\mathrm{EStart}(p) = \max_{q \in Pred(op)} \begin{cases} 0 & \text{q is unscheduled} \\ \mathrm{T(q)} + \mathrm{Delay(q,p)} & \text{otherwise} \end{cases}$$

- Late Start
  - Analog to Early Start the latest possible schedule time

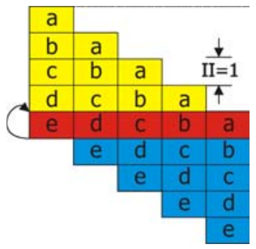


Highest-level-first Priority

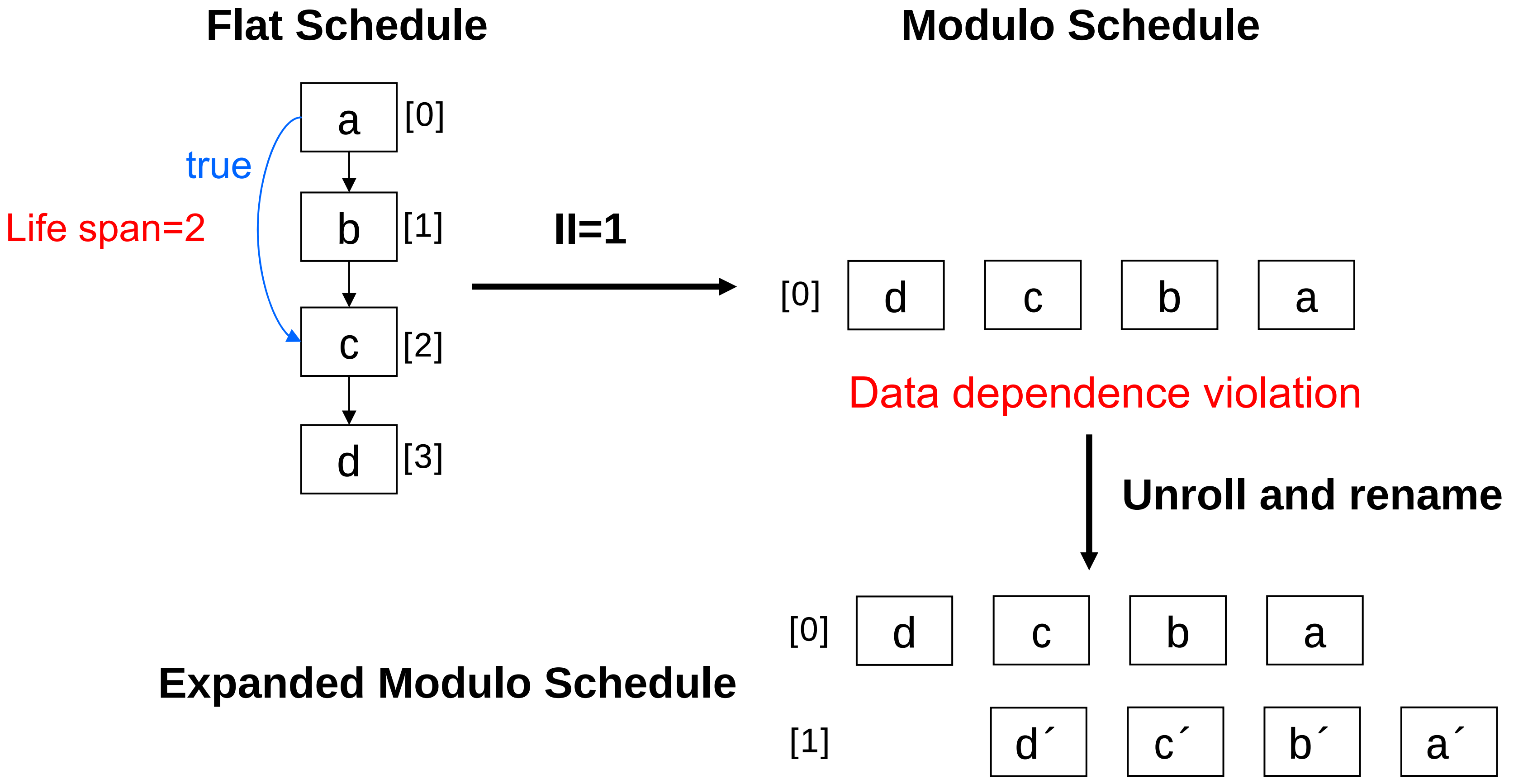
- Larger priority ➔ smaller slack available for the operation w.r.t. to the critical path

$$HeightR(p) = \begin{cases} 0 & p = PSEUDO\_STOP \\ \max_{q \in Succ(p)} (HeightR(q) + EffDelay(p,q)) & otherwise \end{cases}$$

$$\mathrm{EffDelay}(p,q) = \mathrm{Delay}(p,q) - II * \mathrm{Distance}(p,q)$$



Modulo Register Expansion



Flat Schedule
Modulo Schedule
a [0]
true
Life span=2
b [1]
c [2]
d [3]
II=1
[0] d c b a
Data dependence violation
Unroll and rename
Expanded Modulo Schedule
[0] d c b a
[1] d´ c´ b´ a´