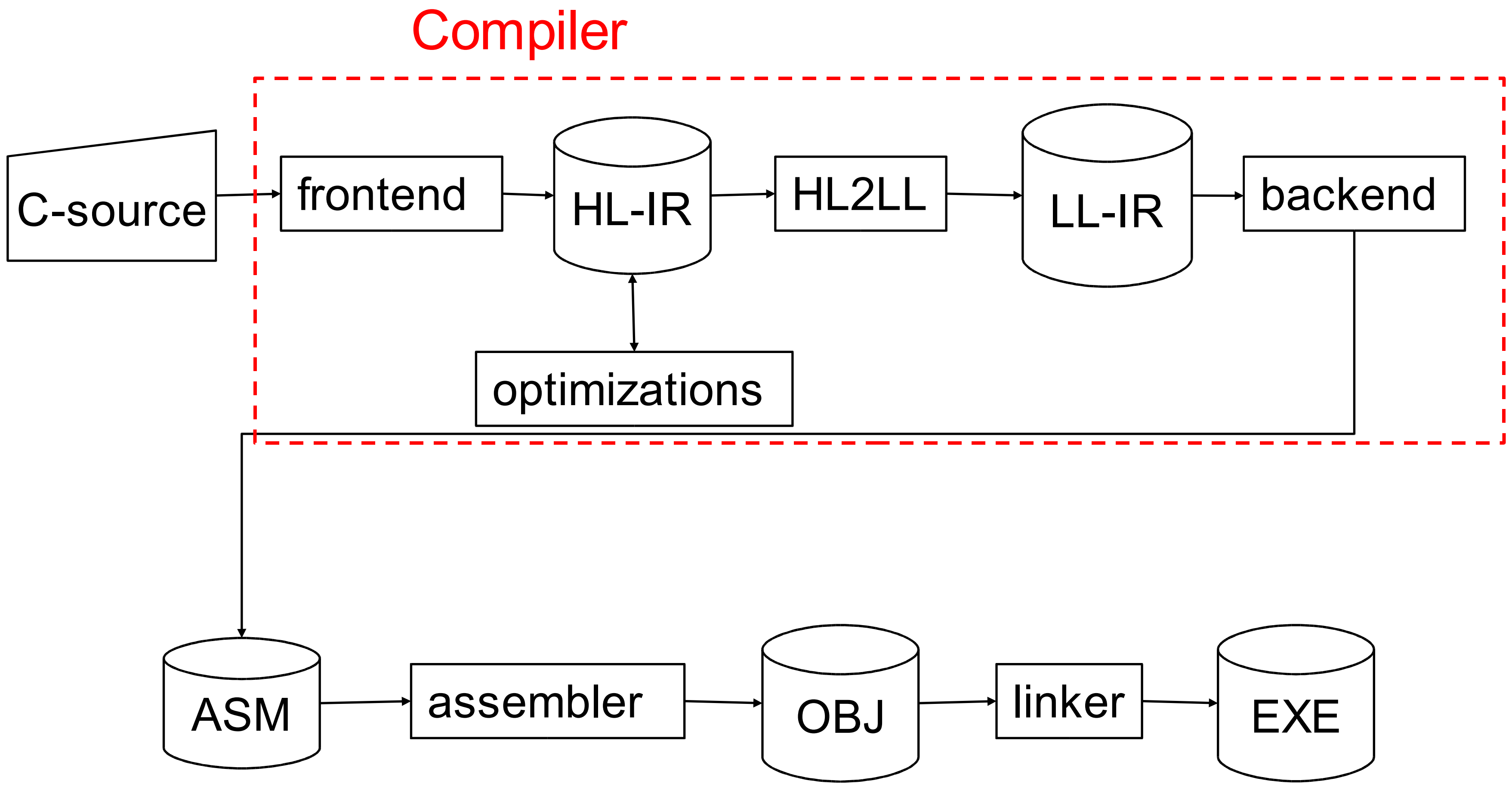
Building a compiler
(considering characteristics of embedded processors)

Peter Marwedel
University of Dortmund, Germany

Effort for building a compiler

- So far we assumed that all the optimizations can be added to some existing tool chain.
- Sometimes, a custom compiler is required, but: the effort for building a custom compiler is underestimated.
- It is not sufficient to design a processor and then think about the compiler later.
- Try to avoid the design of a full compiler. Approaches:
  - Modify an existing compiler.
  - Implement proposed optimizations as pre- or post-pass optimization.
  - Use existing standard software components
  - Use retargetable compiler (see below)
- What if we **have** to look at the entire compiler ? ☞

Anatomy of a compiler



Do not start from scratch!

Existing Compiler Frameworks: gcc

<u>G</u>NU <u>C</u>ompiler <u>C</u>ollection (GNU public license)

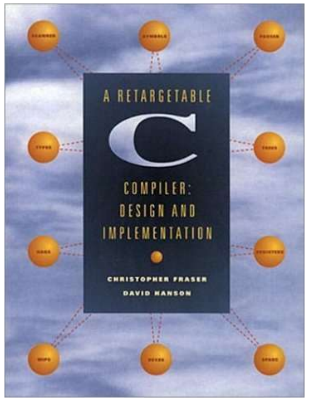
Family of C & C++ compilers
(also supports Java and Fortran)

Available for many different architectures
(e.g. Sparc, Mips, Alpha)

- Code-transformation into an IR (Intermediate Representation) called RTL (Register Transfer Language)
- Designed for homogeneous register machines
- No ideal fit for embedded processors (heterogeneous registers etc.)

based on slide by
Désirée Kraus, Inf 12, 2005

Existing Compiler Frameworks: lcc



<u>L</u>ittle <u>C</u> <u>C</u>ompiler (Princeton University)

„lightweight compiler“ (~ 13.000 lines of code)

„A retargetable C Compiler: Design and Implementation“ by C.W. Fraser & D.R. Hanson

- Limited code optimization capabilities
- Code quality generally lower than GCC‘s
- Translation of C-source into data flow graphs (C language operators + type and size information)
- Inappropriate for high-efficiency embedded code

based on slide by Désirée Kraus, Inf 12, 2005

Existing Compiler Frameworks: EDG

Edison Design Group

C++ frontend by EDG

Supports Standard C++, Microsoft C/C++, GNU C/C++ and other

- High-level IR keeping names, type information and line numbers
- **Loss of information about original source code**
- Inappropriate for source to source level transformations
- Backends for generating C and C++ code
  - transformation: C++ → C
  - Loss of code quality for C++ due to intermediate step

based on slide by Désirée Kraus, Inf 12, 2005

Existing Compiler Frameworks: Cosy

CoSy

Compiler system with frontends for C, C++, Fortran and Java

- Developed by ACE (Associated Compiler Experts)
- Common high-level Intermediate Representation
  - standard optimization passes
  - modular extensibility
- Lowering in further steps
- Commercial tool with professional support
- Uses C++ frontend by EDG (Edison Design Group)
- Significant costs

based on slide by Désirée Kraus, Inf 12, 2005

Existing Compiler Frameworks: SUIF

<u>S</u>tanford <u>U</u>niversity <u>I</u>ntermediate <u>F</u>ormat

compiler for ANSI-C and Fortran 77

- High-level IR: „high-SUIF“
- Output of high-level C-Code with minor changes
    - code-quality remains almost constant
    - appropriate for source to source transformations
- Reduction to lists of instructions („low-SUIF“)
- Optimizations
- Version problems SUIF 1 outdated, SUIF 2 never quite completed

based on slide by Désirée Kraus, Inf 12, 2005

Existing Compiler Frameworks: Trimaran

Trimaran

"An Infrastructure for Research in Instruction-Level Parallelism"

"For researchers investigating:

- *Explicitly Parallel Instruction Computing (EPIC)*
- *High-Performance Computing Systems*
- *Instruction-Level Parallelism*
- *Compiler Optimizations*
- *Computer Architecture*
- *Adaptive And Embedded Systems*
- *Language design"*

www.trimaran.org

Existing Compiler Frameworks: LANCE

<u>L</u>S12 <u>AN</u>SI-C <u>C</u>ompilation <u>E</u>nvironment

(University of Dortmund) designed for retargetable compilation and high-level code optimization

- HL-IR: 3-address-code;
  - Lowering of HL constructs into „primitive“ expressions
  - Subset of ANSI-C ⇒ can be compiled and executed
- Medium level effort for compiler generation
- Loss of code-quality by transforming source code into IR
- Low cost solution available from ICD

based on slide by D. Kraus, Inf 12, 2005

Existing Compiler Frameworks: ICD-C

2nd generation framework developed by ICD (Informatik Centrum Dortmund)

- HL- IR: keeping original C-constructs, names, name scopes and file contexts
  - code quality remains constant
  - standard optimizations
  - extensibility
- Ideal for source to source code optimizations
- Industrial quality and licensing conditions

© Désirée Kraus, Inf 12, 2005

Lexical analysis (Recap)

- Lexical analysis is based on regular expressions, e.g.
  number = [0-9]+ // a sequence of digits
  identifier = [a-z][a-z0-9]* // sequence of lower case letters
  // and digits, starting with a lower
  // case letter
- Lexical analysis decomposes an input program into a sequence of tokens, e.g.
  a + 3 becomes *identifier operator number*
- Lexical analysis is usually based on finite state machines (FSMs or "automata"). Deterministic finite state machines (DFAs) are used to simulate non-deterministic finite state machines (NFAs).

lex: a lexical analyzer generator

The required automaton for lexical analysis can be generated with lex. Example*:

```
%{  /*C declarations */
include “tokens.h”
union {int ival; string sval; double fval;} yylval
int charPos=1;
#define ADJ (EM_tokPos=charPos, charPos+=yyleng)
%}
/*lex definitions*/
digits [0-9]+
%% /*regular expressions and actions*/
if                {ADJ; return IF;}
[a-z][a-z0-9]*    {ADJ; yylval.sval=String(yytext); return ID;}
{digits}          {ADJ; yylval.ival=atoi(yytext); return NUM;}
```

* A. W. Appel, Modern compiler implementation in C, 1998

Parsing (Recap)

Most computer languages can be described by context-free grammars (CFGs).

CFGs comprise derivation rules such as

expr = "(" *expr* "+" *expr* ")"

expr = *digits*

expr = *identifier*

" encloses characters which represent themselves (are not meta characters like in regular expressions).

These rules can be used recursively to build up complex structures.

Analysis of CFG-based languages require push-down automata (PDAs).

yacc (yet another compiler compiler)

yacc generates the tables required for parsing CFGs.

Sample input*:

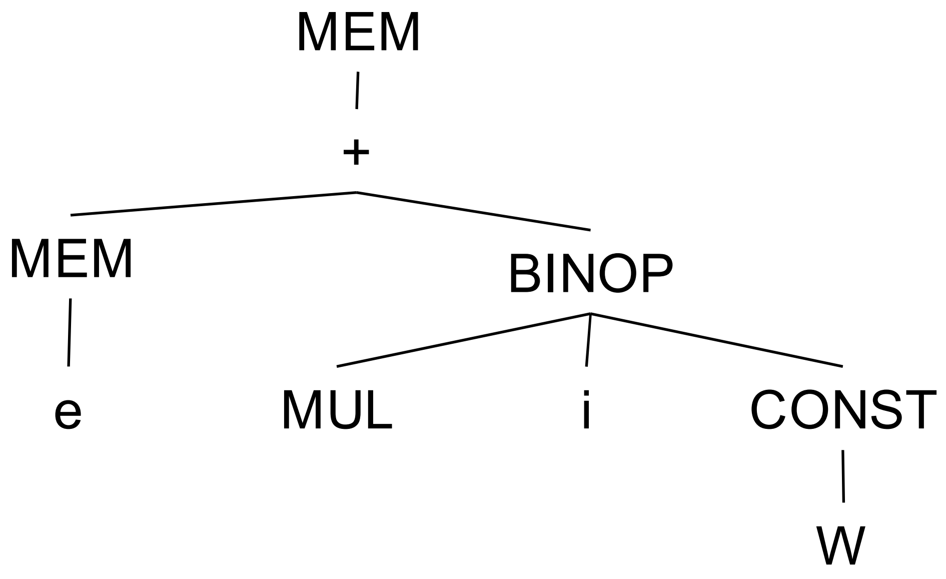
```
{ int yylex(void);
void yyerror(char *s) { Em_error(EM_tokPos, “%s”, s); }
%}
%token ID | WHILE  | BEGIN | END | DO | IF | THEN | ELSE | SEMI |
ASSI
%start prog
%%
prog: stmtlist
stm :    ID ASSI ID                 | WHILE ID DO stm
         | BEGIN stmlist END      | IF ID THEN stm
stmlist: stm                        | stmlist SEMI stm
```

More recent implementations: Bison, occs

* A. W. Appel, Modern compiler implementation in C, 1998

Abstract syntax trees (ASTs)

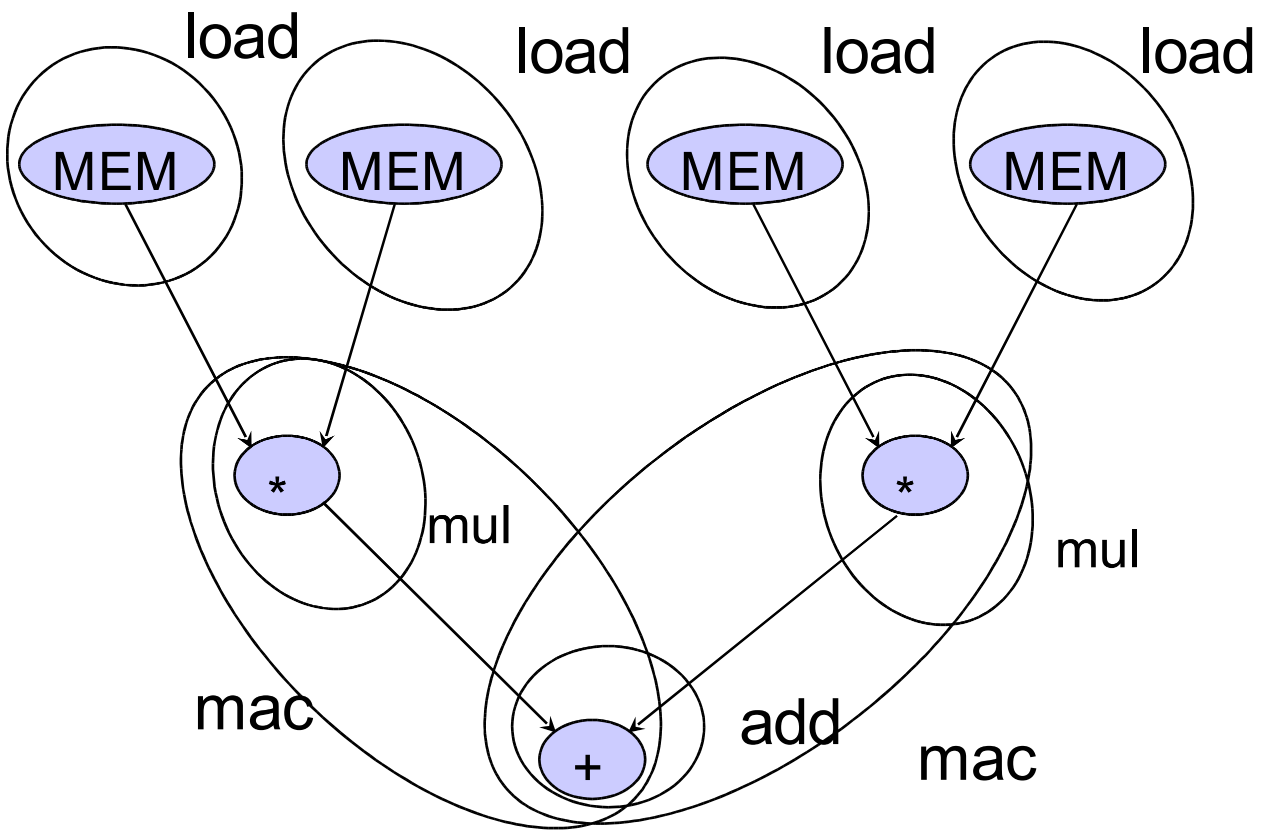
Abstract syntax trees are abstract representations of the input program. Example:



Abstract syntax trees are generated by tools such yacc provided appropriate actions are defined for all derivations

Code selection

Code selection is the task of covering operations of the AST with appropriate operations/instructions of the real machine. Example:



Does not yet consider data moves to input registers.

Code selection (CS) by tree parsing (1)

Instructions are specified as a grammar.
CS = parsing AST with respect to this grammar.
Example (input for iburg tree parser generator):

terminals: {MEM, *, +}
non-terminals: {reg1,reg2,reg3}
start symbol: reg1
rules:

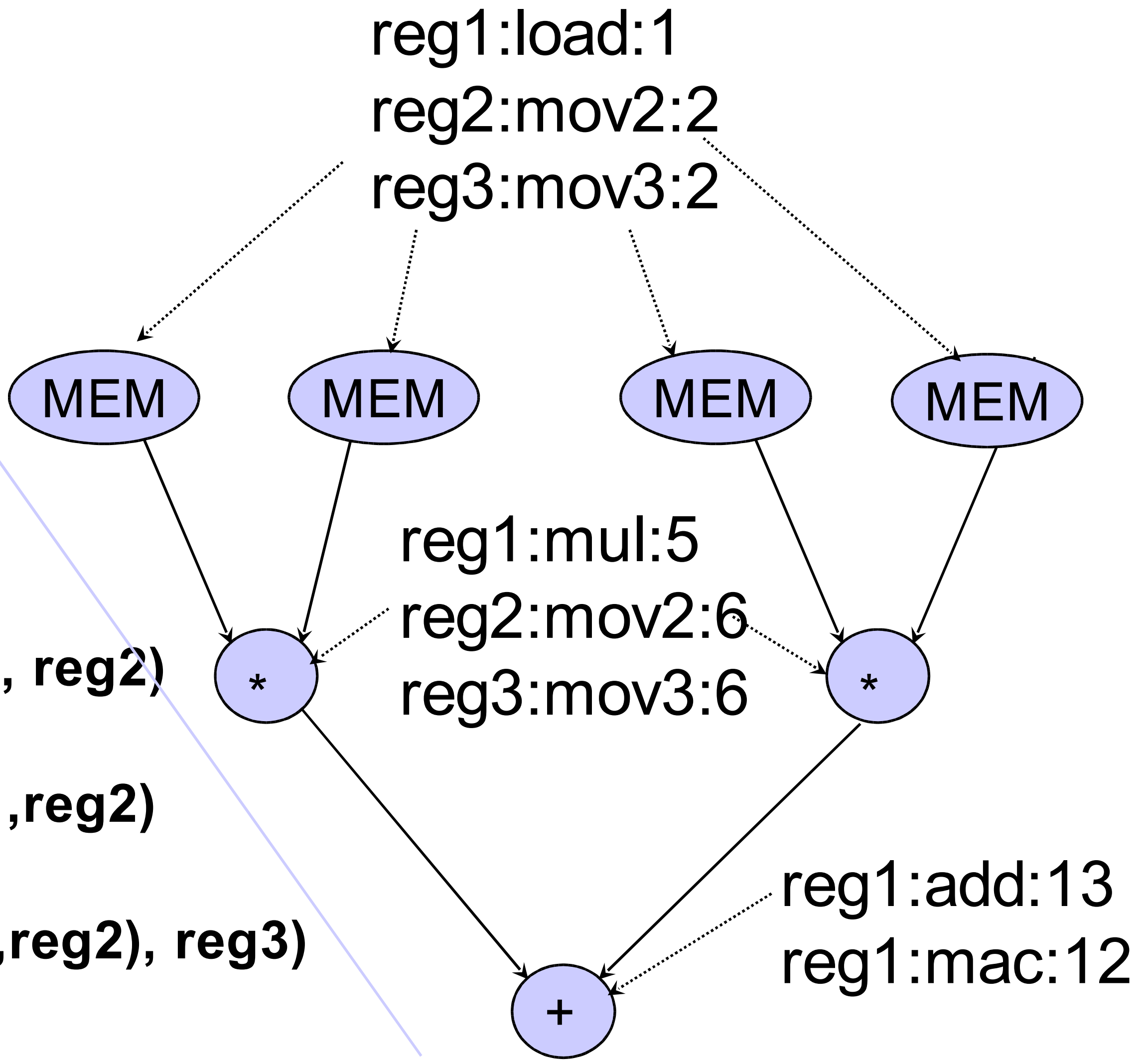
```
 “add”  (cost=2): reg1 ->  + (reg1, reg2)
 “mul”  (cost=2): reg1 -> * ( reg1,reg2)
 “mac”  (cost=3): reg1 -> + (*(reg1,reg2), reg3)
 “load”  (cost=1): reg1 -> MEM
 “mov2”(cost=1): reg2 -> reg1
 “mov3”(cost=1): reg3 -> reg1
```

Code selection by tree parsing (2)

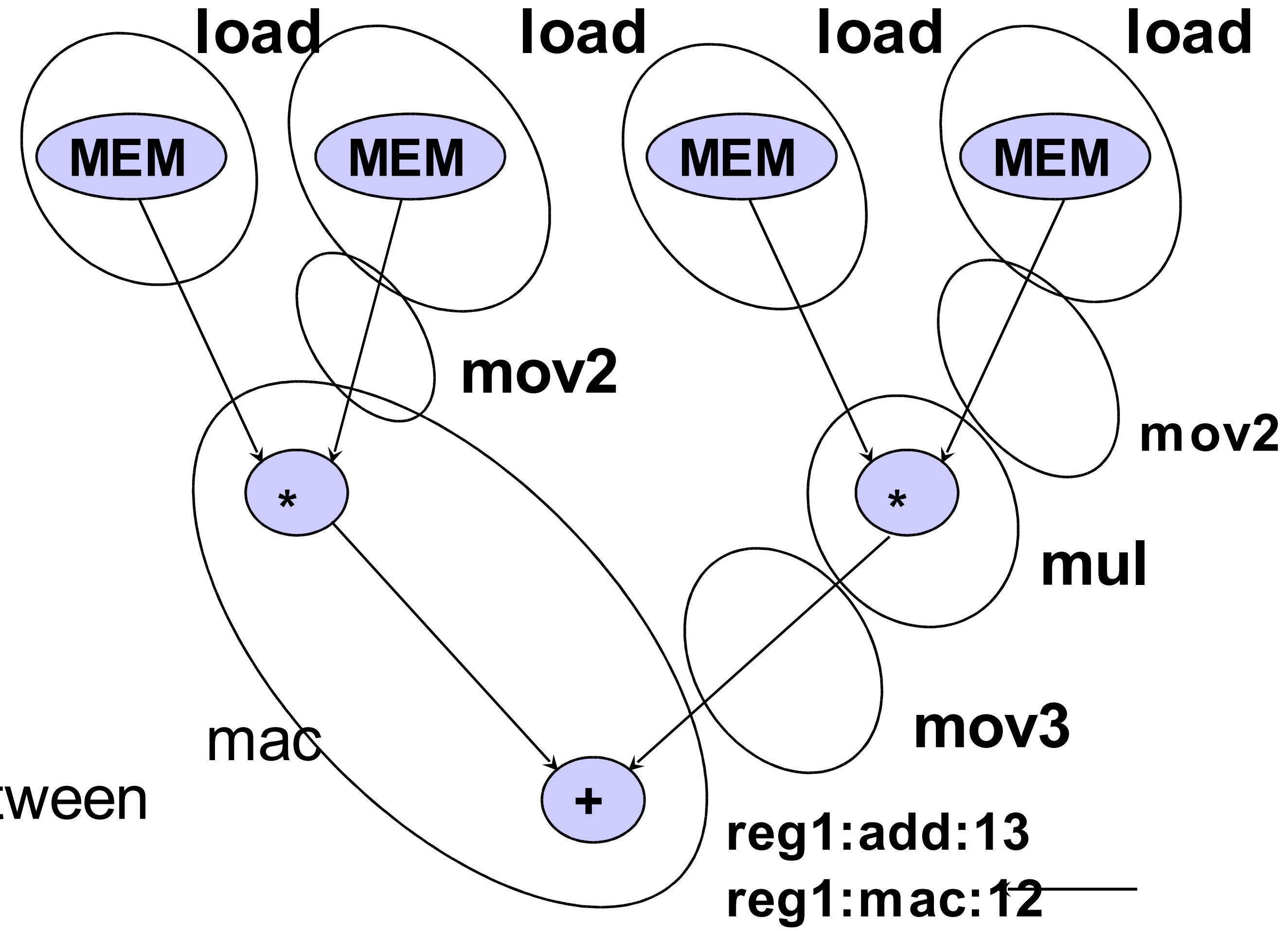
- nodes annotated with (register/pattern/cost)-triples -

"load"(cost=1):
 reg1 -> MEM
"mov2"(cost=1):
 reg2 -> reg1
"mov3"(cost=1):
 reg3 -> reg1

"add" (cost=2):
 reg1 -> +(reg1, reg2)
"mul" (cost=2):
 reg1 -> *(reg1,reg2)
"mac" (cost=3):
 reg1->+(*(reg1,reg2), reg3)



Code selection by tree parsing (3)
- final selection of cheapest set of instructions -

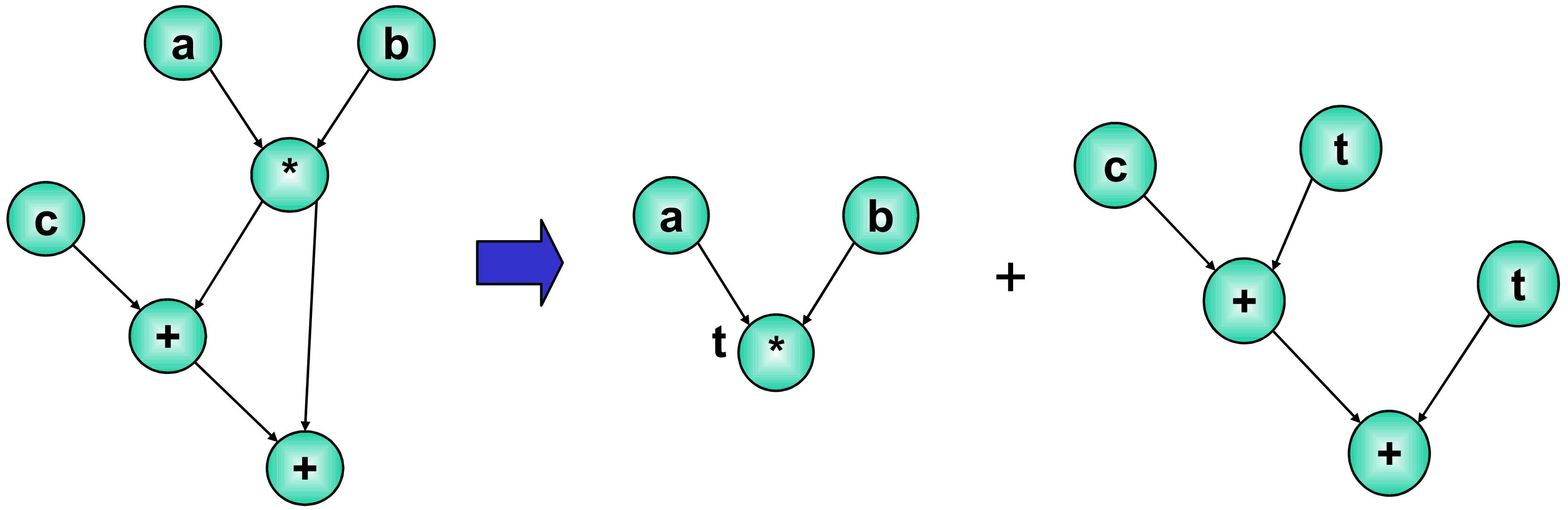


Includes routing of values between various registers!

From tree covering to graph covering

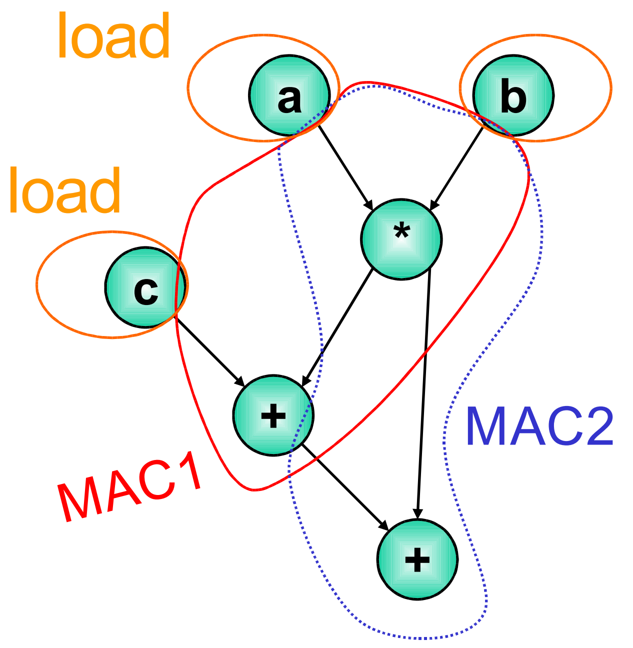
For programs with common subexpressions, the tree covering approach requires data flow graphs to be split at common subexpressions.

Example: c+(a*b)+(a*b)

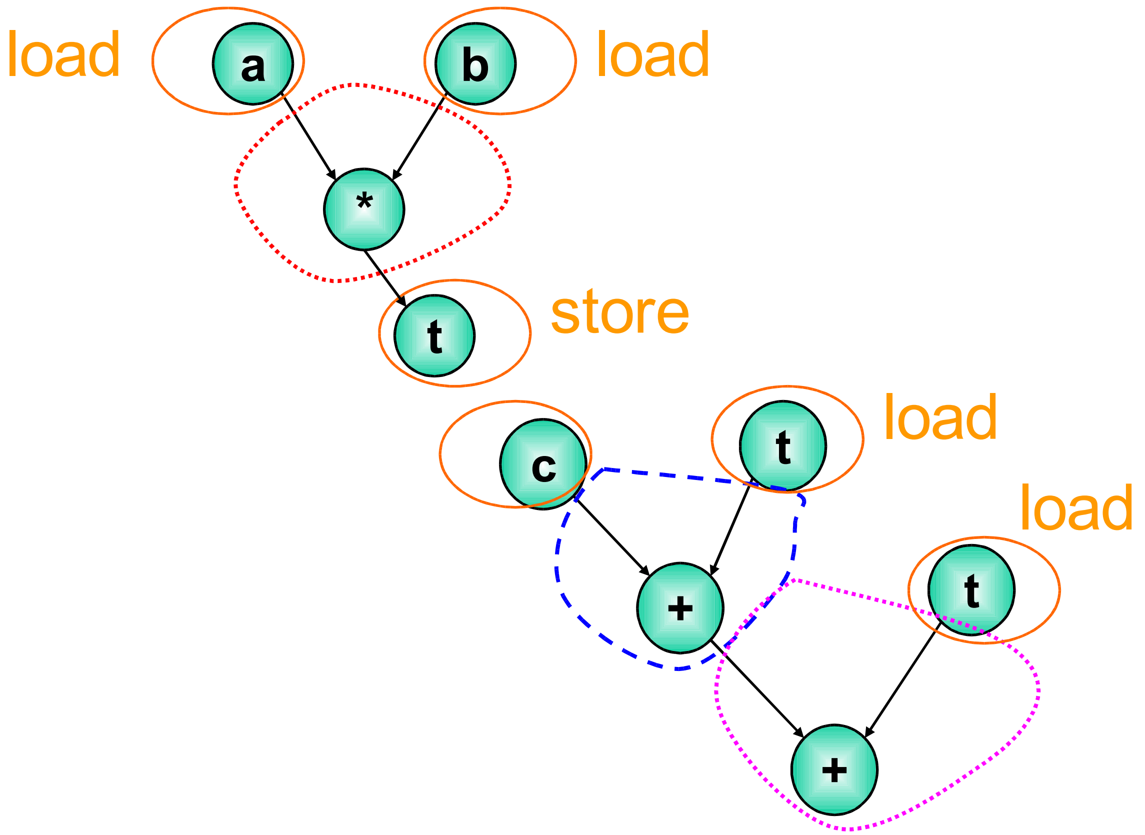


Covers for this example

With graph covering and multiply/accumulate (MAC) instructions



With tree covering



Problems with tree covering for embedded processors

1. Missing exploitation of complex instructions such as MAC
2. Missing homogeneous register file: intermediate value may have to be stored in background memory.

☞ Graph covering should be used.

☹ Graph covering is NP-complete (run-times of all known algorithms increase exponentially as a function of the size of the graph)

Approaches for graph covering

1. Optimal graph covering for small graphs.
2. Heuristic for larger graphs
3. Exploiting special situations, in which graph covering is not NP-complete.

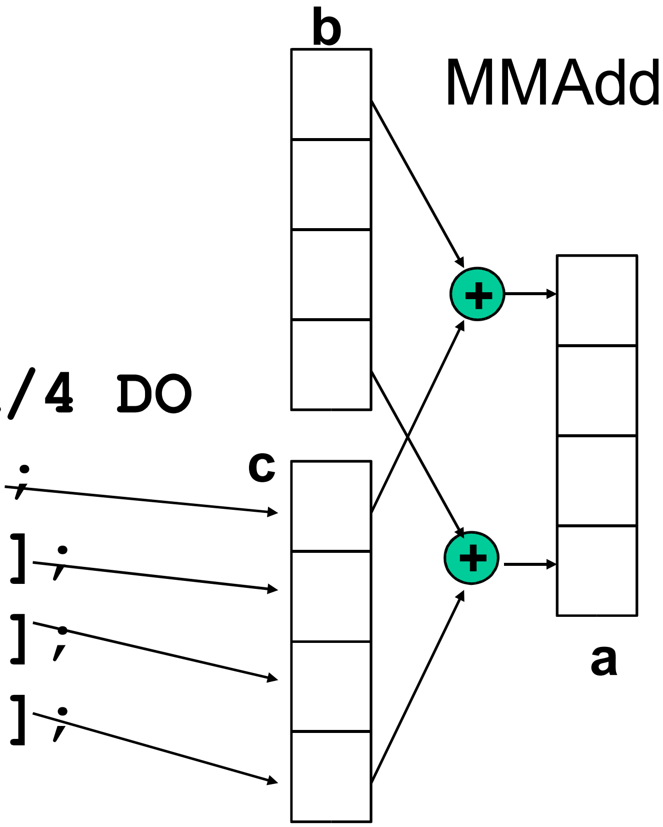
Empirical result for approach 1 combined with 2:

Cost reductions of 20 ... 50% [Bashford]

Exploitation of Multimedia Instructions

```
FOR i:=0 TO n DO
  a[i] = b[i] + c[i]
```

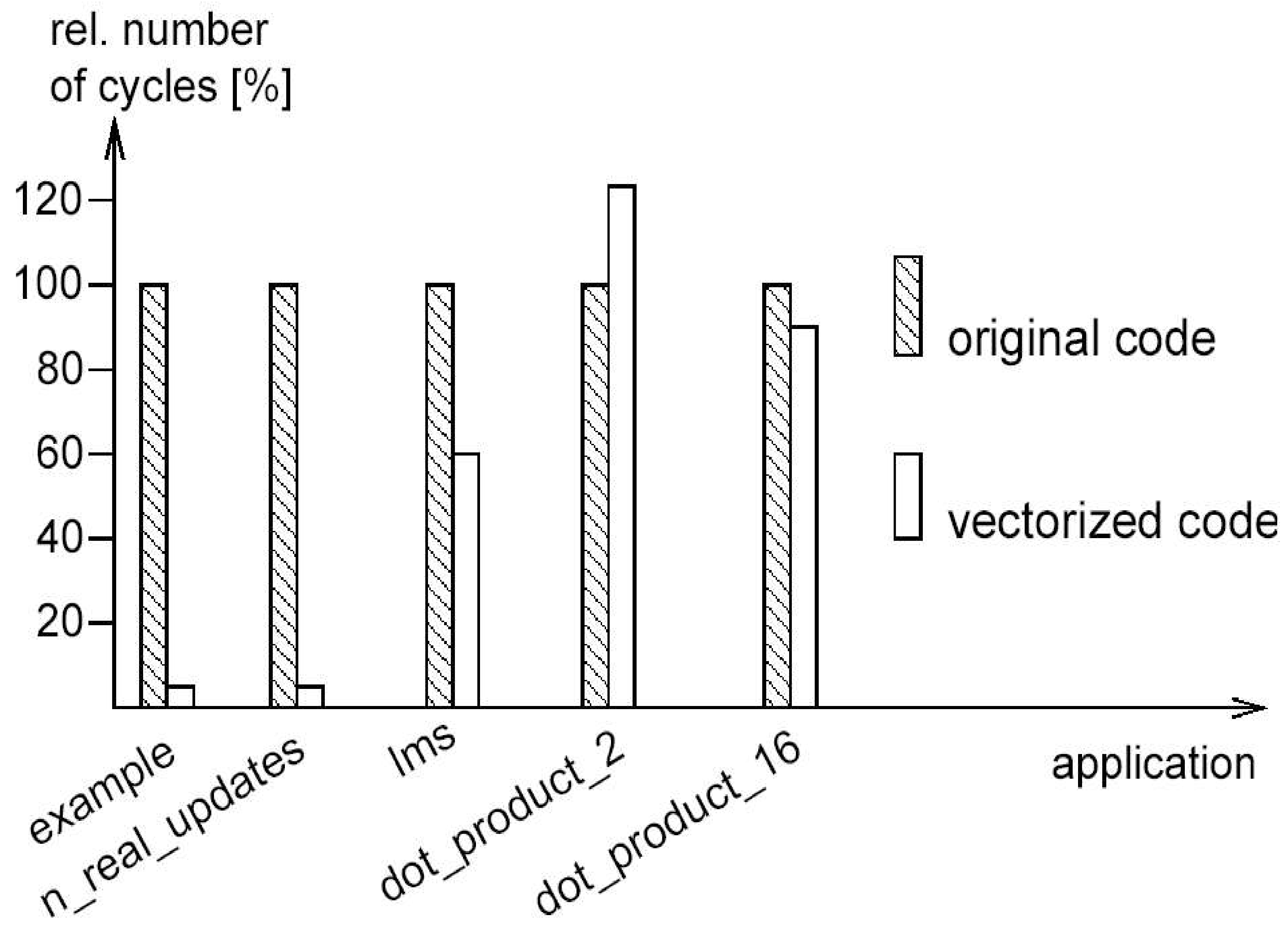
```
FOR i:=0 STEP 4 TO n/4 DO
  a[i  ]=b[i  ]+c[i ];
  a[i+1]=b[i+1]+c[i+1];
  a[i+2]=b[i+2]+c[i+2];
  a[i+3]=b[i+3]+c[i+3];
```



MMAdd (4 x 8/16 bit)

Generation of cover difficult: non-connected regions of the DFG are covered by 1 instruction

Improvements for M3 DSP due to vectorization



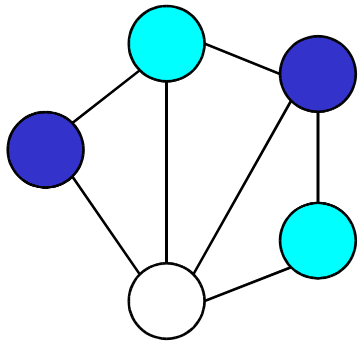
Register allocation (RA)

- Code selection typically ignores limited size of the register set and allocates to “virtual registers” of a register class.
- Register allocation maps virtual registers to physical registers and generates “spill code” (copies to the memory) in case there are not enough registers.

Register allocation using interference graphs

- Nodes of interference graph = virtual registers
- Edge (*u*,*v*) iff if *u* and *v* cannot be allocated to same real register
- Each real register corresponds to a “color”.
  Goal: allocate colors to nodes such that no two nodes connected by an edge have the same color;
  minimize the number of colors (coloring problem).

Interference graph



- The coloring is known to be NP-complete in general.
- In practice, heuristics are used to solve the problem.
- Linear complexity within basic blocks (“left edge algorithm”).

Phase coupling problem for embedded processors

Traditional compiler:

- Code selection (CS) assigns virtual register, only a single class of registers exists; hence CS cannot select the “wrong” class.
- Register allocation (RA) assigns real register.
- Instruction scheduling of little importance.

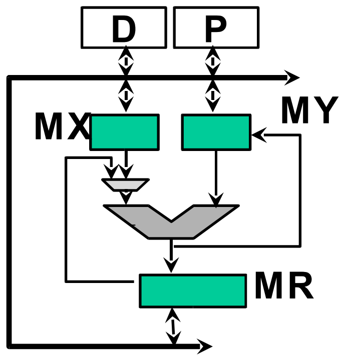
Embedded system compiler:

- CS assigns virtual register, several classes exist; the class to be selected is only known during RA.
- RA cannot precede CS, since the registers that are needed, are only known after CS.
- IS also has mutual dependencies with CS and RA

☞ **cyclic dependency, difficult to handle.**

Potential solution: representation of sets of registers

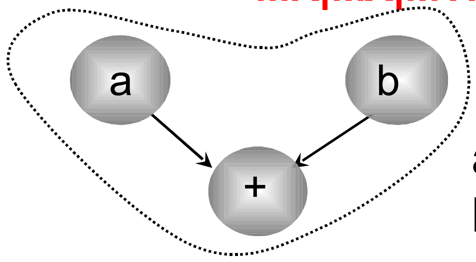
Easy with constraint logic programming



MR:=MR+MY
MR:=MX+MY
MY:=MR+MY
MY:=MX+MY

d4
MR|MX|MY:= D|P

d6
MR|MX|MY:= D|P

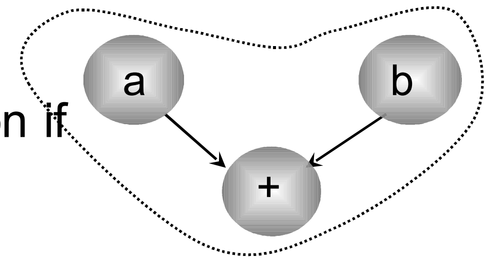


MR|MY := MR|MX|MY+ MR|MX|MY
d1 d2 d3

automatic reduction if b bound to MY

d4
MR|MX:= D|P

MY:= D|P



MR|MY := MR|MX + MY
d2

d2 $\neq$ d3; d4=d2; d6=d3;
d2=MR $\rightarrow$d3 $\neq$MX; ....

$\Leftarrow$ constraints $\Rightarrow$ d4=d2

Efficient representation of constraints in *constraint logic programming*
Delayed binding of resources (delay decisions as long as possible)
Larger decision space for following phases

Instruction Scheduling (IS)

Instruction scheduling is the task of generating an order of executing operations/instructions.

Traditional concerns:

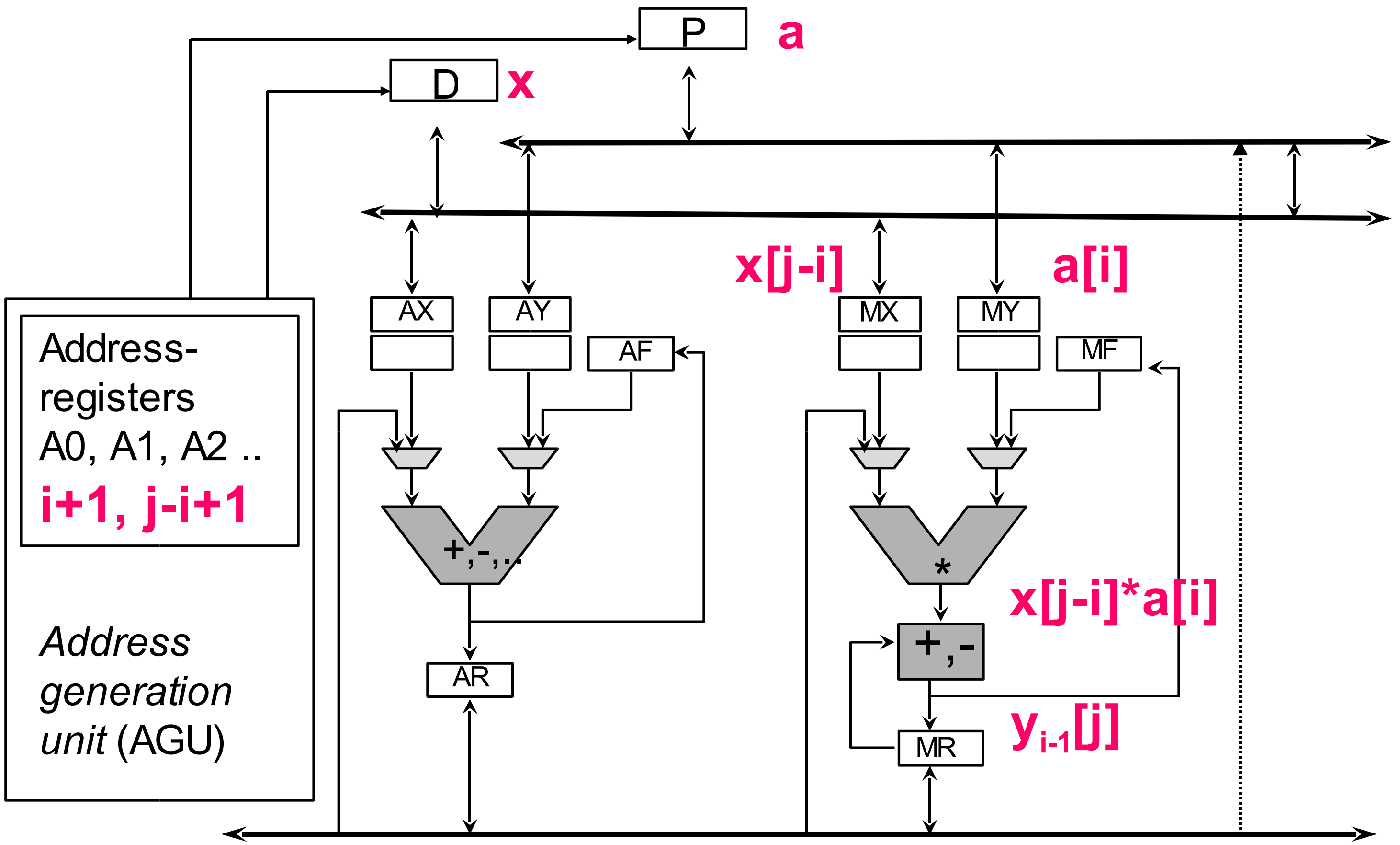
- Chose an order minimizing register requirements
- Fill delay slots of processors with delayed branches
- Avoid pipeline stalls after load instructions
- General attempt to hide memory latency
- Schedule operations on slow functional units (e.g. on floating point units)

Additional requirements for recent embedded processors

☞

Scheduling for parallel instructions

Typical DSP processor:
Several transfers in the same cycle:



Scheduling can be expressed as "compaction" of register-transfers

1: MR := MR+(MX*MY);
2: MX:=D[A1];
3: MY:=P[A2];
4: A1- -;
5: A2++;
6: D[0]:= MR;
.....

$\Rightarrow$

1´: MR := MR+(MX*MY), MX:=D[A1], MY:=P[A2], A1- -, A2++;
2´: D[0]:= MR;

- Modelling of possible parallelism using n-ary relation.
- Generation of integer programming (IP)- model (max. 50 statements/model); e.g.:
  $x_{j,i}$=1 if transfer *j* mapped to instruction *i*, 0 otherwise and
  $\forall i$: $x_{1,i} + x_{6,i} \leq 1$ (no instruction can contain RT 1 and RT 6)
- Using standard-IP-solver to solve equations

Example for ti processor

$u(n) = u(n - 1) + K0 \times e(n) + K1 \times e(n - 1);$
$e(n - 1) = e(n)$

```
ACCU   := u(n - 1)
TR     := e(n - 1)
PR     := TR × K1
TR     := e(n)
e(n - 1) := e(n)
ACCU   := ACCU + PR
PR     := TR × K0
ACCU   := ACCU + PR
u(n)   := ACCU
```

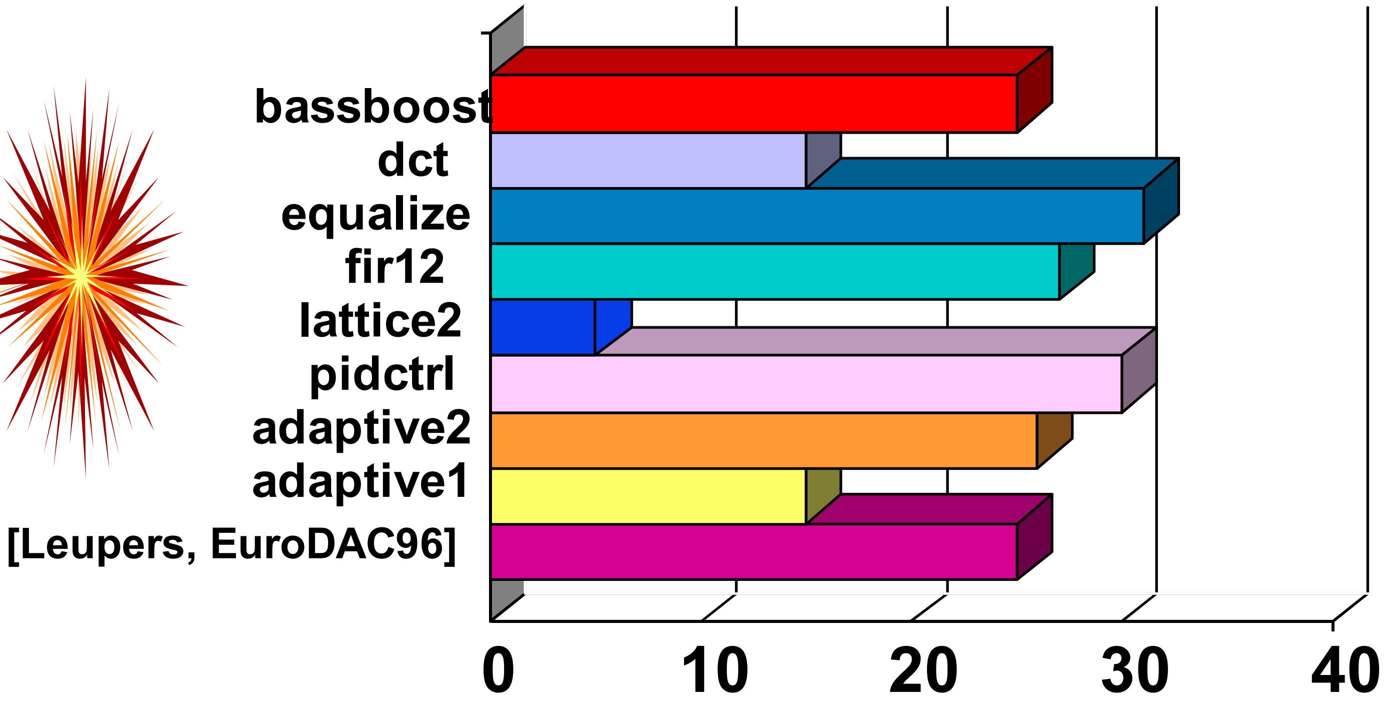
```
ACCU:= u(n - 1)
TR    := e(n - 1)
PR    := TR × K1
e(n - 1):= e(n) || TR:= e(n)  ||
                ACCU:= ACCU + PR
PR    := TR × K0
ACCU:= ACCU + PR
u(n)  := ACCU
```

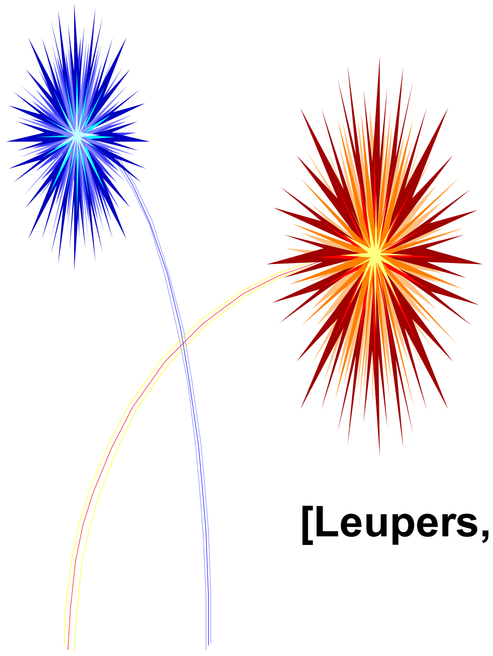
- From 9 to 7 cycles through compaction -

Results

Results obtained through integer programming:

Code size reduction [%]



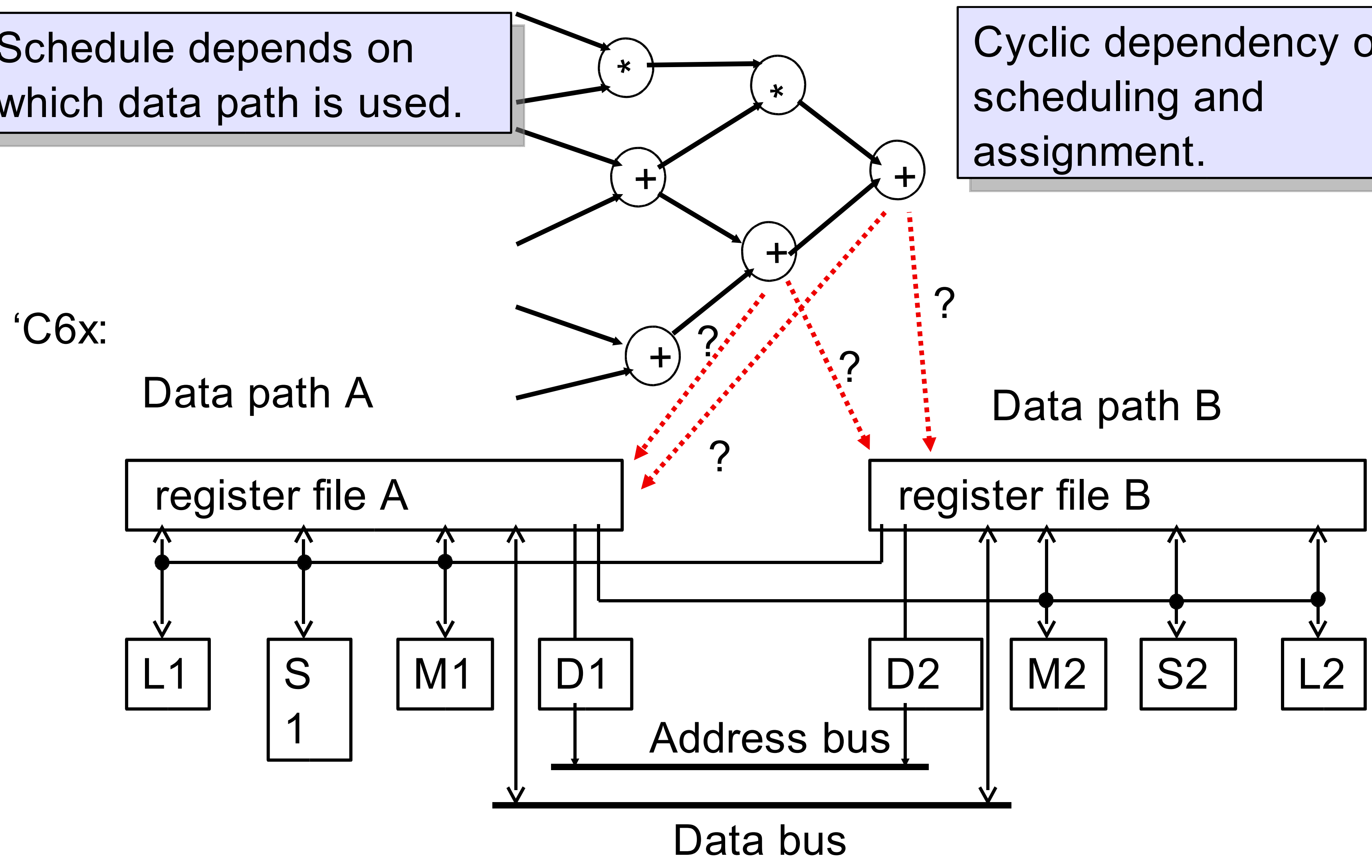


Compaction times: 2 .. 35 sec

Scheduling for partitioned data paths of VLIW processors

Schedule depends on which data path is used.

Cyclic dependency of scheduling and assignment.

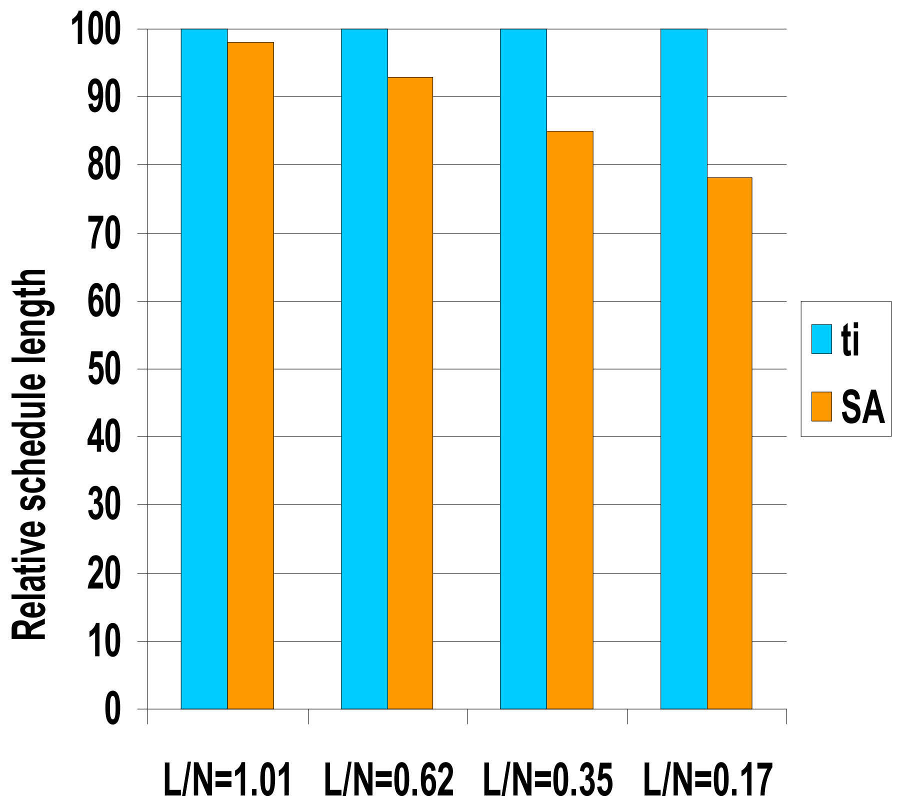


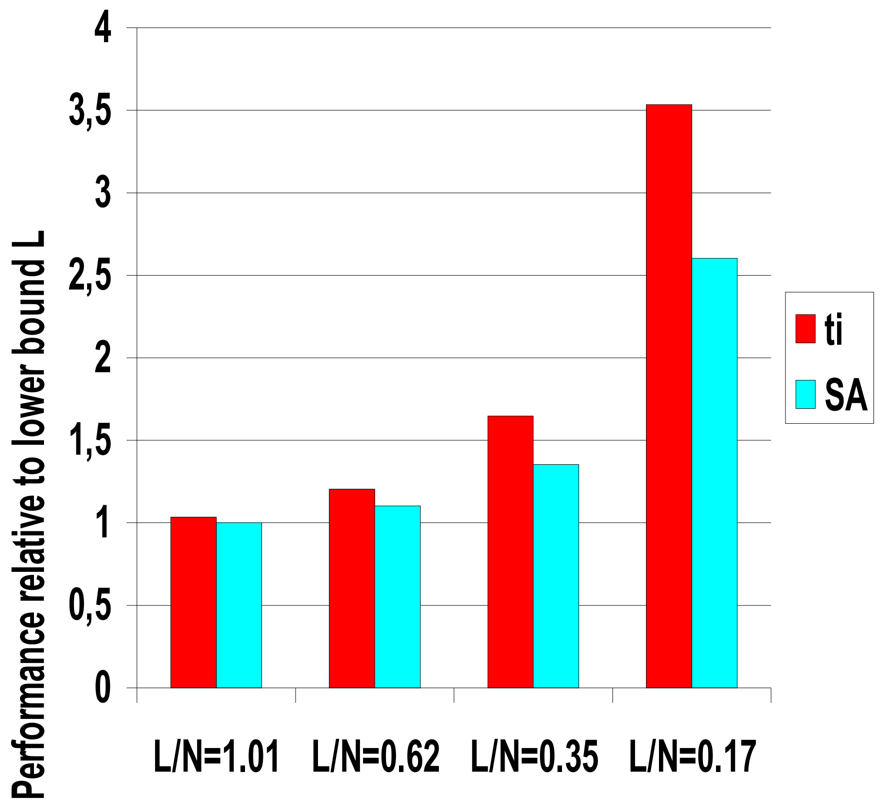
Integrated scheduling and assignment using Simulated Annealing (SA)

```
algorithm Partition
input DFG G with nodes;
output: DP: array [1..N] of 0,1 ;
var int i, r, cost, mincost;
 float T;
 begin
   T=10;
   DP:=Randompartitioning;
   mincost :=
    LISTSCHEDULING(G,D,P);
   WHILE_LOOP;
   return DP;
 end.
```

```
WHILE_LOOP:
while T>0.01 do
 for i=1 to 50 do
  r:= RANDOM(1,n);
  DP[r] := 1-DP[r];
  cost:=LISTSCHEDULING(G,D,P);
  delta:=cost-mincost;
  if delta <0 or
    RANDOM(0,1)<exp(-delta/T)
    then mincost:=cost
    else DP[r]:=1-DP[r]
   end if;
  end for;
  T:= 0.9 * T;
end while;
```

Results: relative schedule length as a function of the "width" of the data flow graph

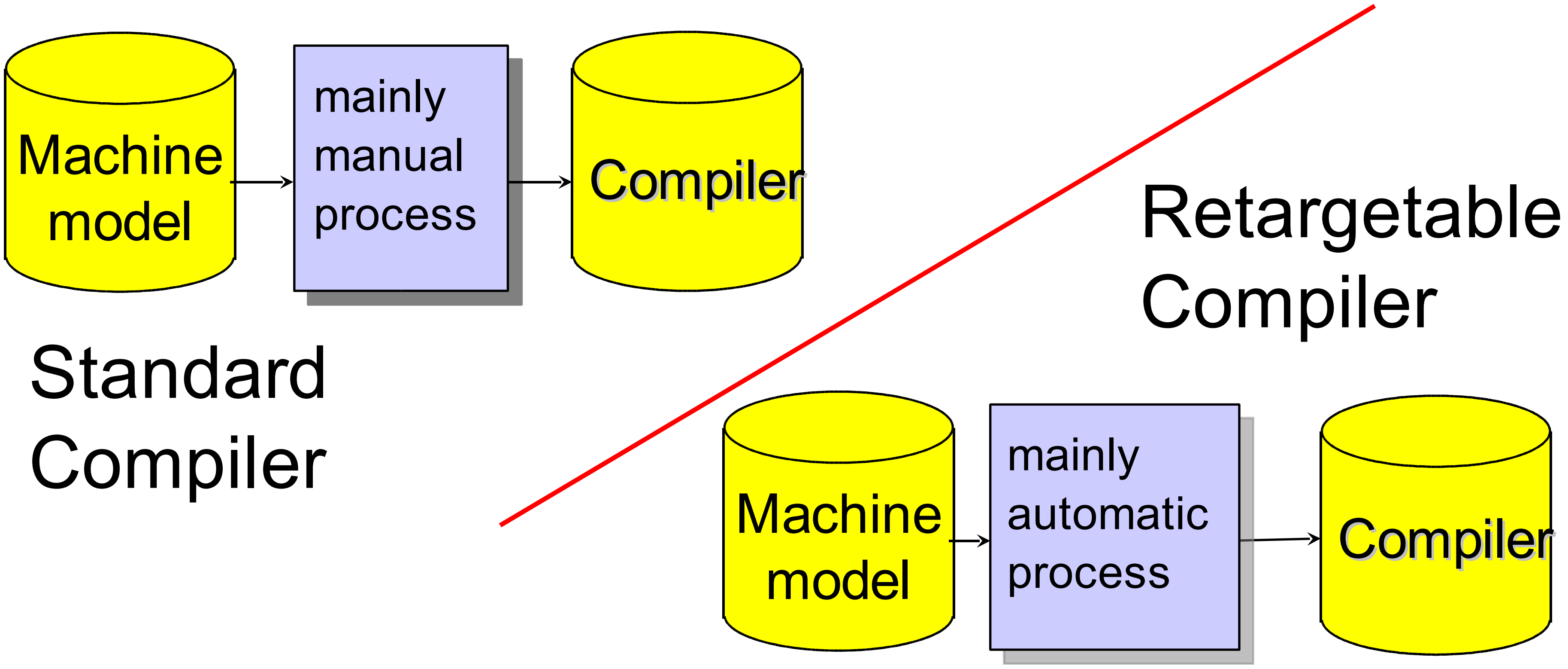




SA approach outperforms the ti approach for "wide" DFGs (containing a lot of parallelism)

For wide DFGs, SA algorithm is able of "staying closer" critical path length.

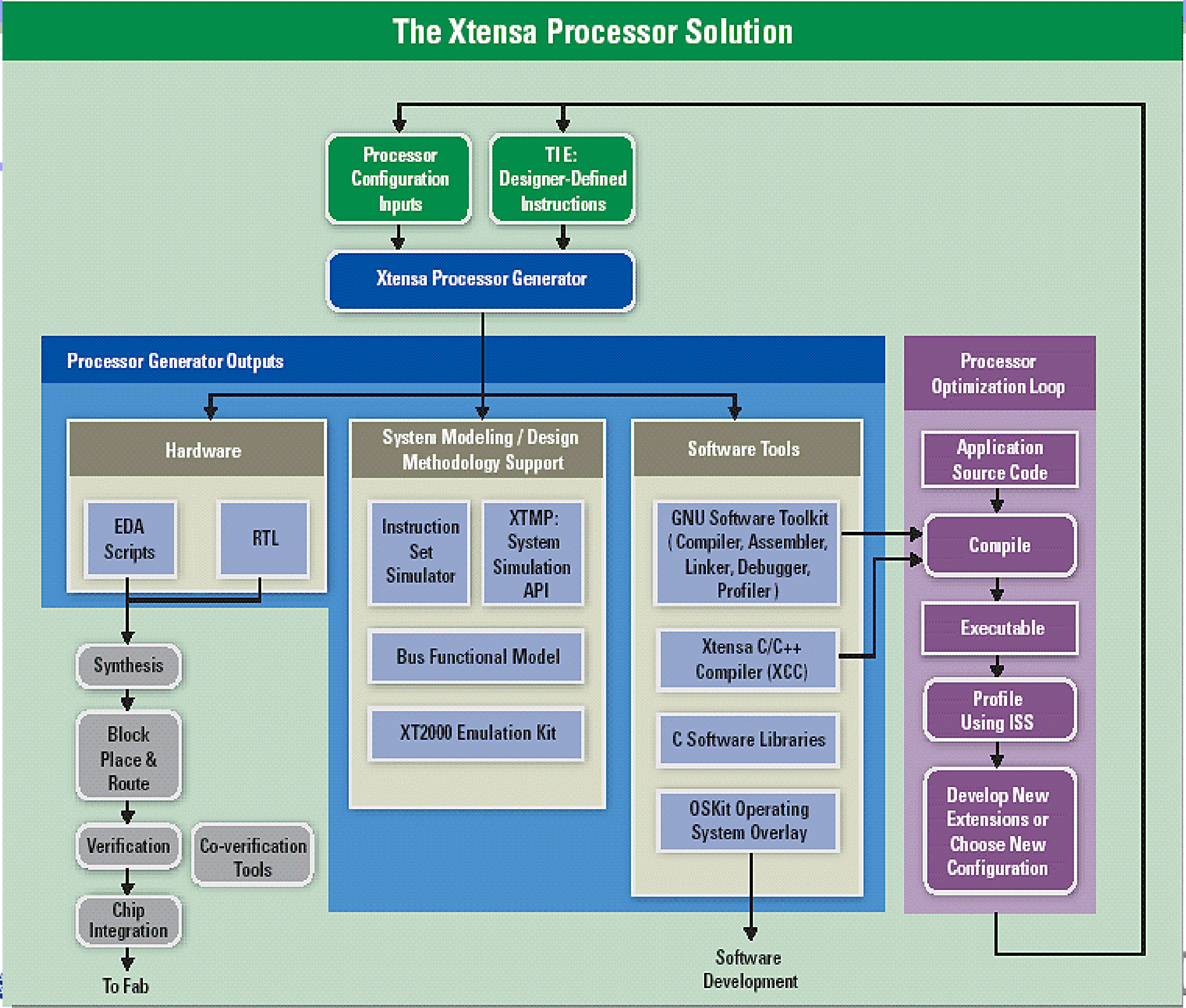
Retargetable Compilers vs. Standard Compilers



Developer retargetability: compiler specialists responsible for retargeting compilers.

User retargetability: users responsible for retargeting compiler.

Commercial goals for retargetable compilation (1)

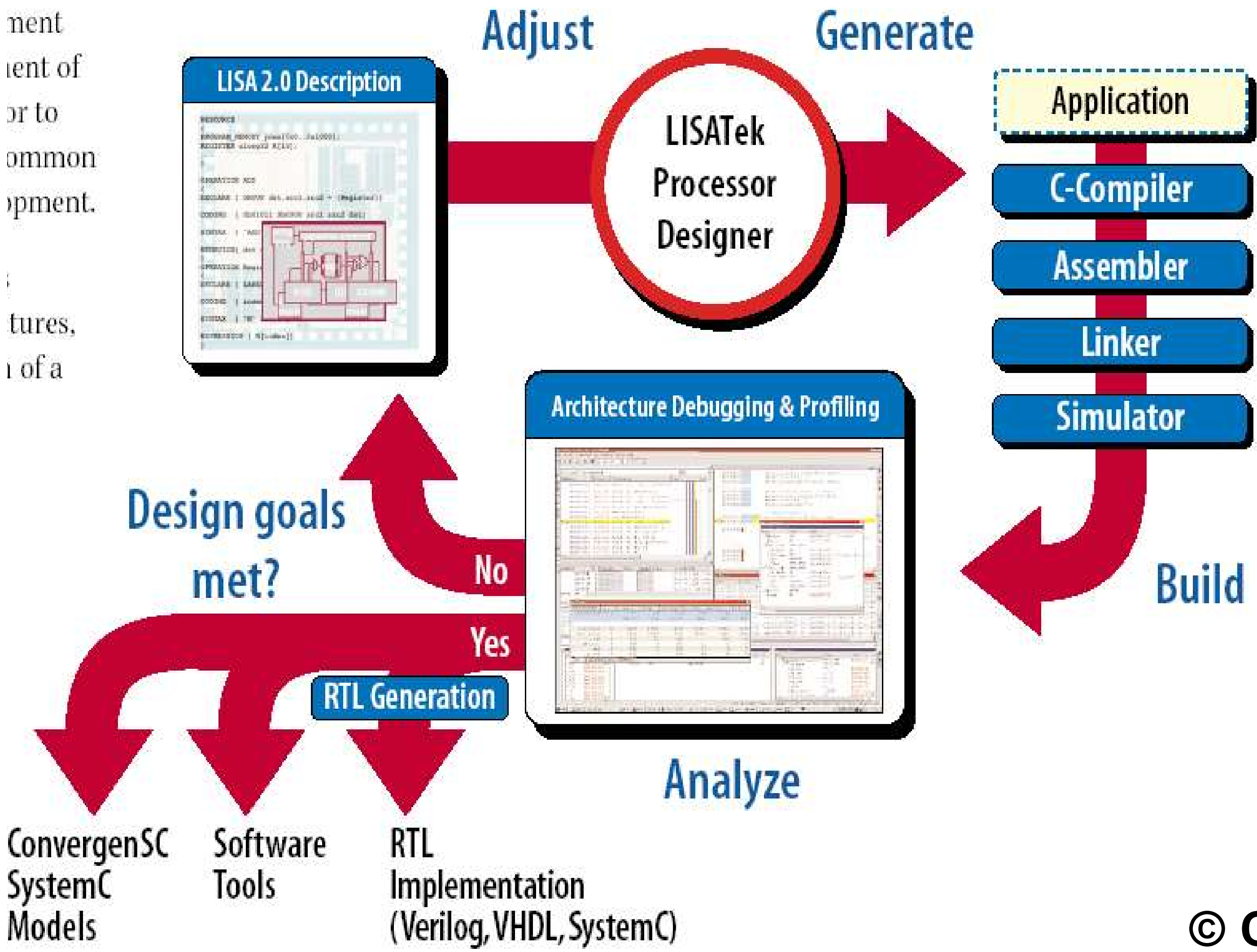

The Xtensa Processor Solution
Processor Configuration Inputs
TIE: Designer-Defined Instructions
Xtensa Processor Generator
Processor Generator Outputs
Hardware
EDA Scripts
RTL
System Modeling / Design Methodology Support
Instruction Set Simulator
XTMP: System Simulation API
Bus Functional Model
XT2000 Emulation Kit
Software Tools
GNU Software Toolkit (Compiler, Assembler, Linker, Debugger, Profiler)
Xtensa C/C++ Compiler (XCC)
C Software Libraries
OSKit Operating System Overlay
Processor Optimization Loop
Application Source Code
Compile
Executable
Profile Using ISS
Develop New Extensions or Choose New Configuration
Synthesis
Block Place & Route
Verification
Co-verification Tools
Chip Integration
To Fab
Software Development

© Tensilica

Xtensa (2)

- The Xtensa approach is based on configuring the gcc compiler suite.
- Exploits the fact that a set of core instructions is fixed and only some additional instructions have to be taken into account.

LisaTek (1)



© Coware Inc

LisaTek (2)

The LisaTek approach is more challenging:

- The Lisa language should be general enough to describe almost all processors.
- Ideally, compilers should be generated for all processors which can be described in the Lisa language.
- Generating highly efficient code for all these processors requires specialized optimizations, which can hardly be made retargetable.

Summary

- Existing compiler frameworks
- Lexical analysis, parsing, abstract syntax trees
- code selection (CS)
  - Tree parsing
  - Graph matching
- Register allocation (RA)
  - Coloring
  - Phase coupling using constraint logic programming (CLP)
- Instruction scheduling (IS)
  - Exploitation of parallel instructions
  - Scheduling for VLIW processors
- Retargetable compilation with Xtensa and LisaTek