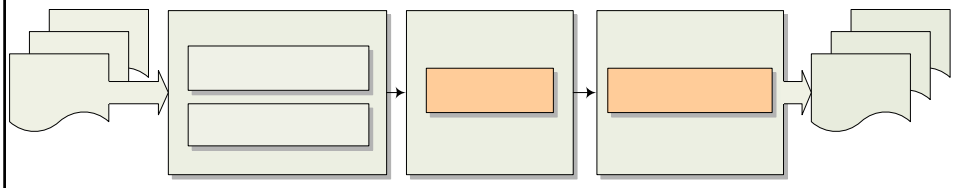
Optimization software for apeNEXT

Max Lukyanov, 12.07.05

- apeNEXT : a VLIW architecture
- Optimization basics
- Software optimizer for apeNEXT
- Current work

Generic Compiler Architecture

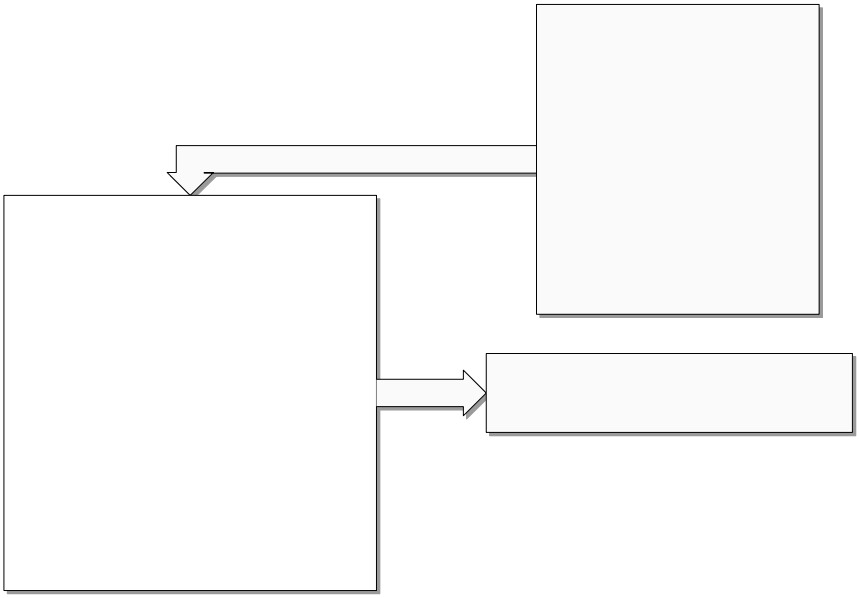


- Front-end: Source Code → Intermediate Representation (IR)
- Optimizer: IR → IR
- Back-end : IR → Target Code (executable)

Importance of Code Optimization

- Code resulting from the direct translation is not efficient
- Code tuning is required to
  - Reduce the complexity of executed instructions
  - Eliminate redundancy
  - Expose instruction level parallelism
  - Fully utilize underlying hardware
- Optimized code can be several times faster than the original!
- Allows to employ more intuitive programming constructs improving clarity of high-level programs

Optimized matrix transposition

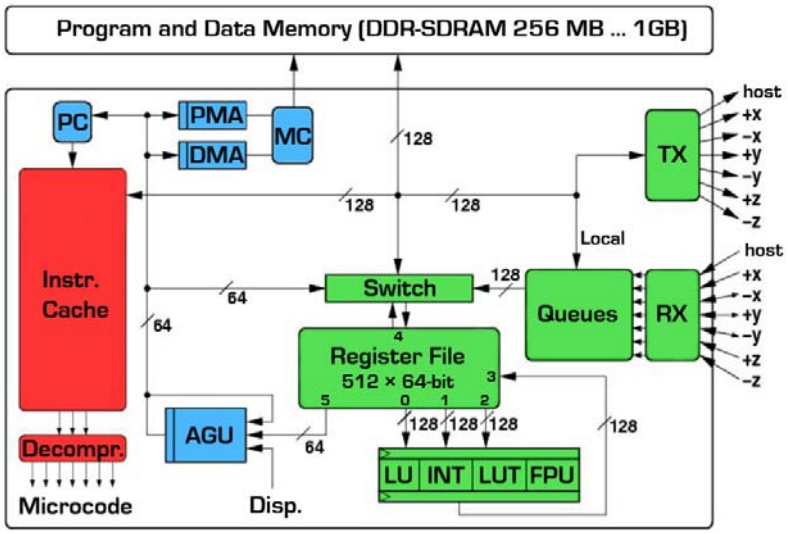


apeNEXT/VLIW

Very Long Instruction Word (VLIW) Architecture

- General characteristics
  - Multiple functional units that operate concurrently
  - Independent operations are packed into a single VLIW instruction
  - A VLIW instruction is issued every clock cycle
- Additionally
  - Relatively simple hardware and RISC-like instruction set
  - Each operation can be pipelined
  - Wide program and data buses
  - Software compression/Hardware decompression of instructions
  - Static instruction scheduling
  - Static execution time evaluation

The apeNEXT processor (J&T)



apeNEXT microcode example

ADDR	DISP	C5	MCC	STKC	FLW	IOC	AGU	ASEL	BS5	P5	BS4	C4	P4	MPC	BS3	C3	P3	P2	P1	P0
00FFFCE5:	00020000	0	M2Q	-	-	-	DA	-	0	18	0	0	00	-	0	0	00	00	00	00
00FFFCE6:	00000000	1	-	-	-	-	-	-	0	00	0	0	00	-	0	0	00	00	00	00
00FFFCE7:	00000010	1	-	-	-	-	RXE	LAL	0	00	0	0	00	-	3	0	17	00	00	00
00FFFCE8:	00100000	0	-	-	-	-	-	-	0	00	0	0	00	IADD	0	0	00	00	01	17
00FFFCE9:	00020000	0	-	-	-	-	DA	-	0	00	0	0	00	-	0	0	00	00	00	00
00FFFCEA:	00000000	0	-	-	-	-	-	-	0	00	0	0	00	-	0	0	00	00	00	00
00FFFCEB:	00000000	0	-	-	-	-	-	-	0	00	0	0	00	-	0	0	00	00	00	00
00FFFCEC:	00000000	0	-	-	-	-	-	-	0	00	0	0	00	-	3	0	17	00	00	00
00FFFCED:	00000000	0	-	-	-	-	-	-	0	00	0	0	00	IADD	0	0	00	00	01	17
00FFFCEE:	00000000	0	-	-	-	-	-	-	0	00	0	0	00	-	0	0	00	00	00	00
00FFFCEF:	00000000	0	-	-	-	-	-	-	0	00	0	0	00	-	0	0	00	00	00	00
00FFFCF0:	00000000	0	-	-	-	-	-	-	0	00	0	0	00	-	0	0	00	00	00	00
00FFFCF1:	00000000	0	-	-	-	-	-	-	0	00	0	0	00	-	3	0	17	00	00	00
00FFFCF2:	00000000	0	-	-	-	-	-	-	0	00	0	0	00	IADD	0	0	00	00	01	17
00FFFCF3:	00000000	0	-	-	-	-	-	-	0	00	0	0	00	-	0	0	00	00	00	00
00FFFCF4:	00000000	0	-	-	-	Q2R	-	-	0	00	0	0	00	-	0	0	00	00	00	00
00FFFCF5:	00000000	0	M2Q	-	-	Q2R	-	-	0	18	3	0	54	-	0	0	00	00	00	00
00FFFCF6:	00000000	1	-	-	-	Q2R	-	-	0	00	3	0	55	CN04	3	0	17	00	54	54
00FFFCF7:	**00000010**	**1**	**-**	**-**	**-**	**Q2R**	**RXE**	**LAL**	**0**	**00**	**3**	**0**	**54**	**CN04**	**0**	**0**	**00**	**00**	**55**	**55**
00FFFCF8:	00100000	0	-	-	-	Q2R	-	-	0	00	3	0	55	CN04	0	0	00	44	54	54
00FFFCF9:	00020000	0	-	-	-	Q2R	DA	-	0	00	3	0	54	CN04	0	0	00	45	55	55
00FFFCFA:	00000000	0	-	-	-	Q2R	-	-	0	00	3	0	55	CN04	0	0	00	46	54	54
00FFFCFB:	00000000	0	-	-	-	Q2R	-	-	0	00	3	0	54	CN04	0	0	00	47	55	55
00FFFCFC:	00000000	0	-	-	-	Q2R	-	-	0	00	3	0	55	CN04	0	0	00	48	54	54
00FFFCFD:	00000000	0	-	-	-	Q2R	-	-	0	00	3	0	54	CN04	0	0	00	49	55	55
00FFFCFE:	00000000	0	-	-	-	Q2R	-	-	0	00	3	0	55	CN04	0	0	00	4a	54	54
00FFFCFF:	00000000	0	-	-	-	Q2R	-	-	0	00	3	0	54	CN04	0	0	00	4b	55	55
00FFFD00:	00000000	0	-	-	-	Q2R	-	-	0	00	3	0	55	CN04	3	0	56	4c	54	54
00FFFD01:	00000000	0	-	-	-	Q2R	-	-	0	00	3	0	54	CN04	3	0	57	4d	55	55
00FFFD02:	00000000	0	-	-	-	Q2R	-	-	0	00	3	0	55	CN04	3	0	58	4e	54	54

← VLIW

apeNEXT: specific features

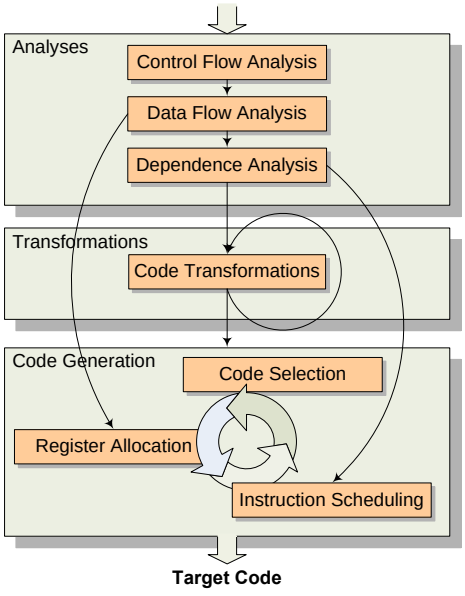
- Predicated execution
- Large instruction set
- Instruction cache
  - Completely software controlled
  - Divided on static, dynamic and FIFO sections
- Register file and memory banks
  - Hold real and imaginary parts of complex numbers
- Address generation unit (AGU)
  - Integer arithmetics, constant generation

apeNEXT: challenges

- apeNEXT is a VLIW
  - Completely relies on compilers to generate efficient code!
- Irregular architecture
  - All specific features must be addressed
- Special applications
  - Few, but relevant kernels (huge code size)
  - High-level tuning (data prefetching, loop unrolling) on the user-side
  - Remove slackness and expose instruction level parallelism
- Optimizer is a production tool!
  - Reliability & performance

Optimization

Optimizing Compiler Architecture

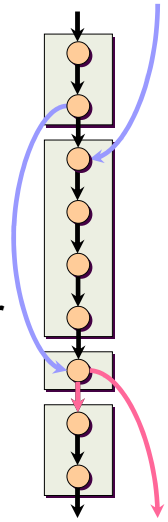


Analysis phases

- Control-flow analysis
  - Determines hierarchical flow of control within the program
  - Detecting loops, *unreachable code elimination*
- Data-flow analysis
  - Determines global information about data manipulation
  - *Live variable analysis* etc.
- Dependence analysis
  - Determines the ordering relationship between instructions
  - Provides information about feasibility of performing certain transformation without changing program semantics

Control-flow analysis basics

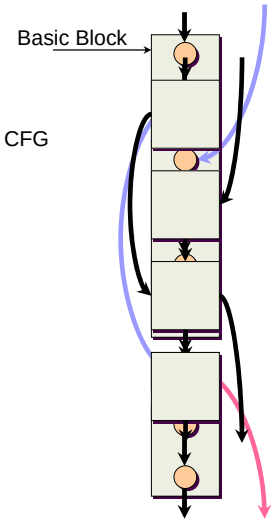
- Execution patterns
  - Linear sequence — execute instruction after instruction
  - Unconditional jumps — execute instructions from a different location
  - Conditional jumps — execute instructions from a different location or continue with the next instruction
- Forms a very large graph with a lot of straight-line connections
- Simplify the graph by grouping some instructions into basic blocks



Control-flow analysis basics

- A basic block is a maximal sequence of instructions such that:
  - the flow of control enters at the beginning and leaves at the end
  - there is no halt or branching possibility except at the end

- Control-Flow Graph (CFG) is a directed graph $G = (N, E)$
  - Nodes (N): basic blocks
  - Edges (E): (u, v) $\in$ E if v can immediately follow u in some



Control-flow analysis (example)

C Code Example

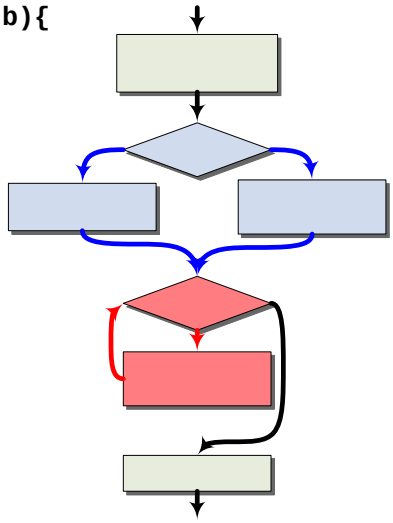
```
int do_something(int a, int b){
   int c, d;
   c = a + b;
   d = c * a;

   if (c > d) c -= d;
   else       a = d;

   while (a < c){
       a *= b;
   }

   return a;
}
```

CFG



Control-flow analysis (apeNEXT)

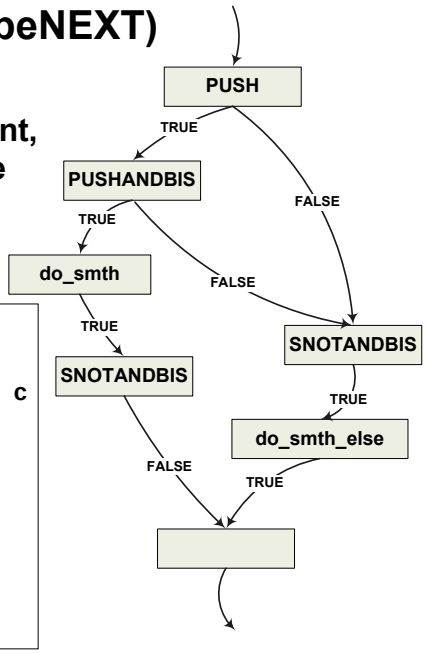
- All the previous stands for apeNEXT, but is ***not*** sufficient, because instructions can be ***predicated***

APE C

```
where(a > b){
   where(b == c){
     do_smth
   }
}
elsewhere{
   do_smth_else
}
```

ASM

```
...
PUSH_GT a b
PUSH_ANDBIS_EQ b c
!! do_smth
NOTANDBIS
!! do_smth_else
...
```



Data-flow analysis basics

- Provides global information about data manipulation
- Common data-flow problems:
  - Reaching definitions (forward problem)
    - Determine what statement(s) could be the last definition of x along some path to the beginning of block B
  - Available expressions (forward problem)
    - What expressions is it possible to make use of in block B that was computed in some other blocks?
  - Live variables (backward problem)
    - More on this later

Data-flow analysis basics

- In general for a data-flow problem we need to create and solve a set of data-flow equations
- Variables: $IN(B)$ and $OUT(B)$
- Transfer equations relate $OUT(B)$ to $IN(B)$

$$OUT(B) = f_B(IN(B))$$

- Confluence rules tell what to do when several paths are converging into a node

$$IN(B) = \bigwedge_{P \in pred(B)} OUT(P)$$

  - $\bigwedge$ is associative and commutative confluence operator
- Iteratively solve the equations for all nodes in the graph until fixed point

Live variables

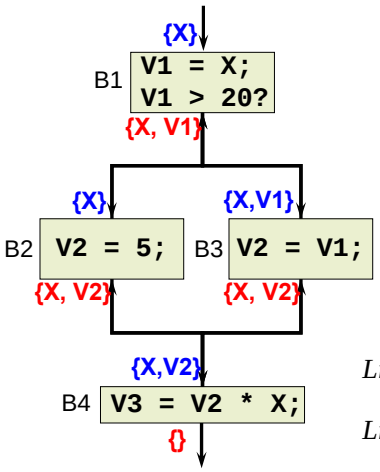
- A variable v is live at a point p in the program if there exists a path from p along which v may be used without redefinition
- Compute for each basic block sets of variables that are live on the entrance and the exit ($LiveIN(B)$, $LiveOUT(B)$)
- Backward data-flow problem (data-flow graph is reversed CFG)
- Dataflow equations:

$$LiveIN(B) = GEN(B) \cup (LiveOUT(B) - KILL(B))$$

$$LiveOUT(B) = \bigcup_{i=Pred(B)} LiveIN(i)$$

  - $KILL(B)$ is a set of variables that are defined in B prior to any use in B
  - $GEN(B)$ is a set of variables used in B before being redefined in B

Live variables (example)



Vars = {X, V1, V2, V3)

GEN(B1) = {X}
GEN(B2) = {}
GEN(B3) = {V1}
GEN(B4) = {V2 X}

KILL(B1) = {V1}
KILL(B2) = {V2}
KILL(B3) = {V2}
KILL(B4) = {V3}

$$LiveIN(B) = GEN(B) \cup (LiveOUT(B) - KILL(B))$$

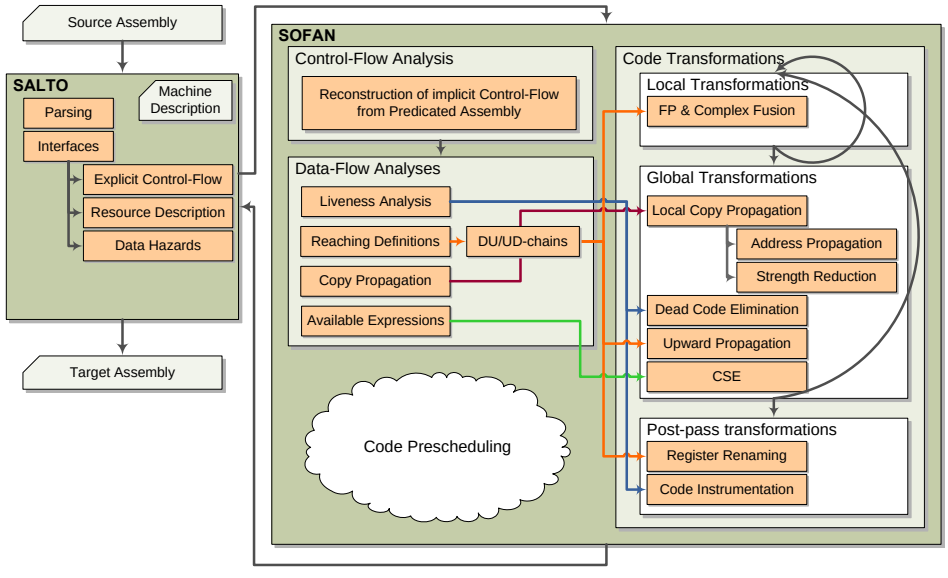
$$LiveOUT(B) = \bigcup_{i=Pred(B)} LiveIN(i)$$

Status and results

Software Optimizer for ApeNEXT (SOFAN)

- **Fusion of floating point and complex multiply-add instructions**
  - Compilers produce add and multiply that have to merged
- **Copy Propagation** (downwards and upwards)
  - Propagating original names to eliminate redundant copies
- **Dead code removal**
  - Eliminates statements that assign values that are never used
- **Optimized address generation**
- **Unreachable code elimination**
  - branch of a conditional is never taken, loop does not perform any iterations
- **Common subexpression elimination**
  - storing the value of a subexpression instead of re-computing it
- **Register renaming**
  - removing dependencies between instructions

Software Optimizer for apeNEXT (SOFAN)



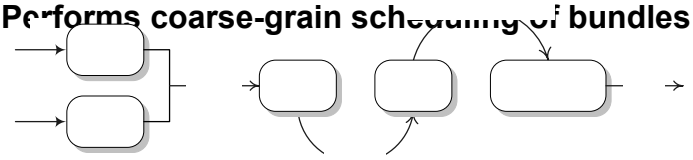
Benchmarks

operation	*max*	*asm*	*C*	*C+SOFAN*	*TAO+SOFAN*
zdotc	50%	41%	28%	40%	37%
vnorm	50%	37%	31%	34%	26%
zaxpy	33%	29%	27%	28%	28%

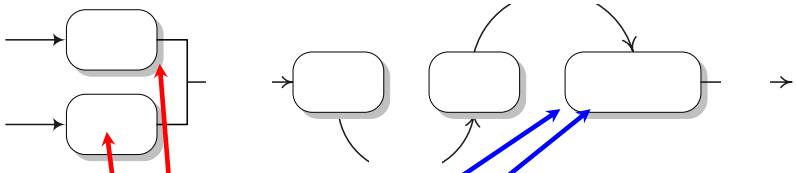
Current work

Prescheduling

- Instruction scheduling is an optimization which attempts to exploit the parallelism of underlying architecture by reordering instructions
  - Shaker performs placement of micro-operations to benefit from the VLIW width and deep pipelining
  - Fine-grain microcode scheduling is intrinsically limited
- Prescheduling
  - Groups instruction sequences (memory accesses, address computations) into bundles
  - Performs coarse-grain scheduling of bundles



Phase-coupled code generation



- Phases of code generation
  - Code Selection
  - Instruction scheduling
  - Register allocation

} Poor performance if no communication

- Better understand the code generation phase interactions
  - On-the-fly code re-selection in prescheduler
  - Register usage awareness

- The end