VU2 185.324

Compilation Techniques for VLIW Architectures

Dietmar Ebner, Florian Brandner
{ebner|brandner}@complang.tuwien.ac.at

http://complang.tuwien.ac.a.t/cd/vliw

In the Last Lecture(s)

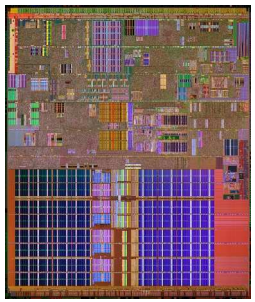
- VLIW instruction set design
  - Expose architectural details
- Instruction Encoding
  - Operation vs. Instruction
  - Template based encodings
  - Dispatching techniques
- Clustered Architectures

In the Last Lecture(s)

- Code Compression
- History
- Example Architectures
  - Intel Itanium (IA64)
  - Multiflow TRACE
  - OnDemand CHILI
  - HP / ST Micro ST2xx

In Today's Lecture

- Floating Point Number Formats
- Embedded C
- Profiling
- The Role of the Compiler
- Loop Unrolling



Number Formats

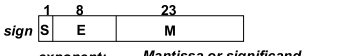
- Integer
  - Two's complement prevalent on all platforms
- Floating Point Support
  - Hardware FPUs are not very common for embedded systems
  - IEEE 754
  - “Softfloat” implementations
  - Fixed-point arithmethic

IEEE 754 Standard

- Single and double precision
- Addresses several issues
  - Representation
  - Arithmetic operations
  - Range and Precision
  - Rounding
  - Exceptions (e.g., divide by zero, overflow, ...)

IEEE 754 Single Precision

- C: `float` datatype
- Normalized Mantissa
- Biased Exponent



exponent: excess 127 binary integer

Mantissa or significand sign + magnitude, normalized binary significand w/ hidden one bit: 1.M

actual exponent is e = E - 127

$X = (-1)^S \, 2^{E-127} \, (1.M)$

01000101110010000000000000000000

S=0 $E=10001011_2 = 139_{10}$ $M=1+2^{-1}+2^{-4} = 1.5625$

$X = (-1)^0 \times 2^{139-127} \times 1.5625 = \mathbf{6400}$

IEEE754

- Denormalized Numbers
  - Fill the gap between $[0 - 2^{-bias}]$
  - Exponent of 0
  - $X = (-1)^S \, 2^{-bias+1} \, (0.M)$
- Special Values for Infinity, NaN
- Four rounding modes
  - round to nearest (default)
  - round towards plus infinity
  - round towards minus infinity
  - round towards 0

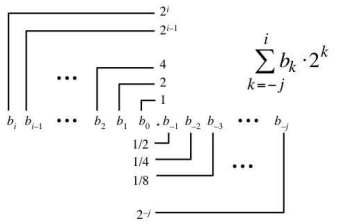
Softfloat Libraries

- Compiler replaces floating point comparisons and arithmetic operations with support library calls
- **Very** slow due to exception handling, denormalized numbers, complex rounding, etc.
- **`addsf3`** from libgcc (ARM): about 230 LOC
- Mostly highly optimized target dependent assembler libraries

Fixed-Point Arithmetic

- Simple way to deal with fractional data
- Easy to implement
- w = i + j

```
i=j=8
0 .. 255.99609375
step: 0.00390625
```



Embedded C

- ISO/IEC TR18037 (approved 2004)
- Mainly identical with industry standard DSP-C
- Addresses some major deficiencies:
  - Support for fixed-point arithmetic
  - Support for saturated logic
  - Named address spaces and named-register storage classes
- No high level support for circular buffers (c.f. `circ` as in DSP-C)

Fixed-Point in Embedded C

```
[signed | unsined] [short | long] _Fract
[signed | unsined] [short | long] _Accum
```

- Overflow and Rounding
  - explicit: `_Sat [_Fract | _Accum]`
  - implicit: #pragma STDC FX_FRACT_OVERFLOW [SAT | DEFAULT]
- Exceptions in arithmetic conversions

```
fract f = 0.25; int i=3;
f*=i /*0.75*/ != f*= (fract) i;
```

- Result type is the one with highest rank

Named Address Spaces

- Global **generic** address space
- Additional implementation defined intrinsic address spaces as type qualifiers:
  `_A struct {int a; fract b;} *_B q;`
- No address space qualification for objects with automatic storage duration
- Pointers point to a specific address space, but can be cast explicitly

The Role of the Compiler

- "Probably the largest engineering effort of a VLIW system"
  - Hundreds of men-years
  - More than 1 Million LOC
  - Typical Lifetime: 10-15 years
- Rigorous software engineering requirements
- Many opposing engineering goals
- Extensive quality assurance / validation

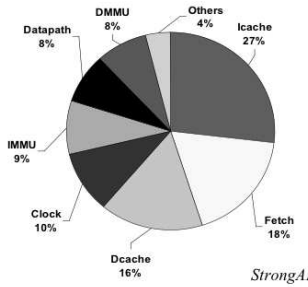
Engineering Goals

- Effort for building a *robust* **and** *aggressive* ILP compiler ist often underestimated
- Correctness!
- Optimization goals
  - ***Execution Time***
  - *Code Size*
    static code size v.s. dynamic footprint (icache)
  - *Energy*
- Objectives are often diametrically opposed

Energy Optimizations

- Important for battery powered and thermally constrained systems
- Traditional optimizations often benefit speed, energy, and size (CSE, PRE, dead code elim.)
- Not the case for many ILP oriented techniques:
  - loop unrolling
  - tail duplication
  - function inlining
  - predication & speculation

Where Power goes in a CPU



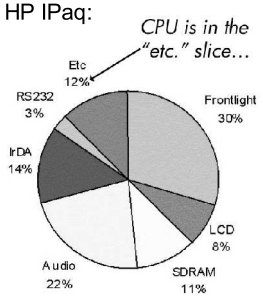
StrongARM, JSSC'96

- Integer Unit: 8%
- Issue Logic: 18%
- Memories: 50–55%

Where Power goes in a System

- Sometimes, the power consumption of the CPU is neglectable
- Power consumption to a large extend controlled by Software
- Effectiveness of compiler oriented optimizations is disputed

HP IPaq:



Terminology

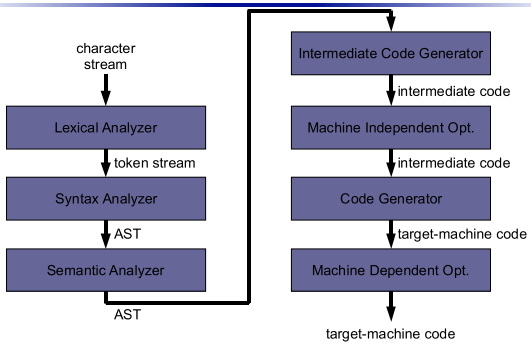
CPI

- *cycles per instruction*
- CPI of 1 has traditionally been the "holy grail" for sequential machines

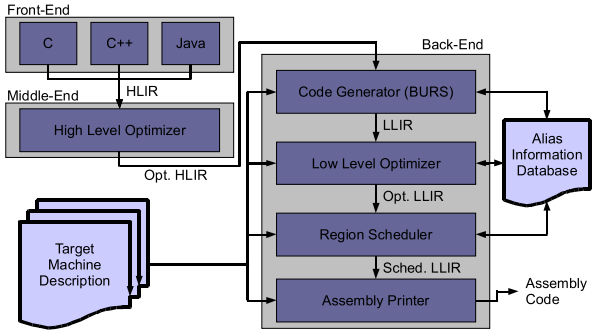
IPC

- *instructions per cycle*
- *more common for parallel architectures*
- *IPC's above 1 are common nowadays*
- *rough measure of the ILP attained during execution*

Typical "Dragon Book" Structure



Phases of an ILP Oriented Compiler



Front-End

- Scanner / Parser / Type System / etc.
- Major improvements during the 70s:
  LALR parser, lexical analysis, type checking, ...
- Language specific analyses and optimizations
  (alias information, virtual function call elim.)
- Tree / DAG oriented representation
- Tools: `lex, yacc/bison, ox, m4, ...`
- Fairly well understood

Middle-End

- Traditional “scalar” and “loop” optimizations
  - common subexpressions, dead code, constant/copy propagation, PRE, ...
- Inlining, alias analysis, interproc. analyses, ...
- How often and when to apply a particular pass is still an unsolved research problem
- To some extend **architecture independent**

Back-End

- Gradually “lowers” HLIR into machine specific assembly code
- Hosts most machine specific optimizations
- “Retargetability” using Target Machine Descriptions (ADL)
- **Major performance contributor** for VLIWs
- Relies on high level transformations and analyses

Side Note: Compiler Optimizations

- Many compiler transformations are – despite common usage - not optimization problems in the classical OR sense
  - What transformations “optimize” a program?
  - What is a “maximum” / “minimum”?
- Optimizations often refer to transformations that are *likely to improve* overall performance
- It's all about trade offs

VLIW Compiler Complexity

“VLIWs require heroic compilers to do what superscalars do in the hardware.”

- The effort required is largely the same for both architectural styles
- Complexity is mainly a function of the aspired ILP
- VLIWs usually just offer more ILP

Key ILP Transformations

- Instruction Scheduling
  - reorganizes operations
  - finds and arranges operations that can be executed in parallel
  - Regions Scheduling
- How to handle Loops?
  - Unrolling and apply region scheduling techniques to loop body
  - Software pipelining

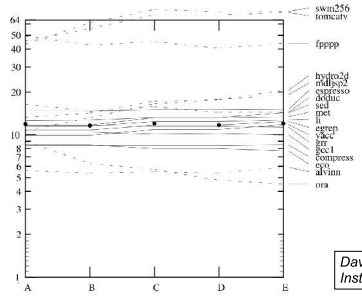
Limits of ILP

- What is the upper bound for ILP given *infinite resources* (early 70s)?
- J.Fisher: “*These studies show that the most ILP we'll ever get goes up by a factor of 2 every 3 years.*”
- What could you really do with infinite resources?
- Basic flaw: studies must ignore future DAG flattening via compiler optimizations!

D. Wall: “Limits of Instruction Level Parallelism” Nov. 1993

- Max. ILP seen around 500 (numerical programs, unlimited parallelism, omniscient scheduler)
- More realistic models
  - Around 50 for peak performance and 10 for the mean
  - In practice, ILP of up to 6 is more common

Parallelism in the “Superb” Model



David W. Wall: “Limits of Instruction Level Parallelism”

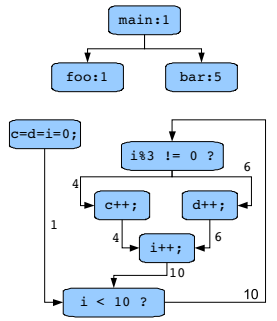
Profiling

- Many crucial ILP transformations (scheduling, clustering, code layout, register allocation) rely heavily on accurate *profiling* information
- Granularity
  - Call graph (`gprof`)
  - Control flow graph (CFG)
  - Paths
- How to obtain representative input data?
- How to keep the information accurate?

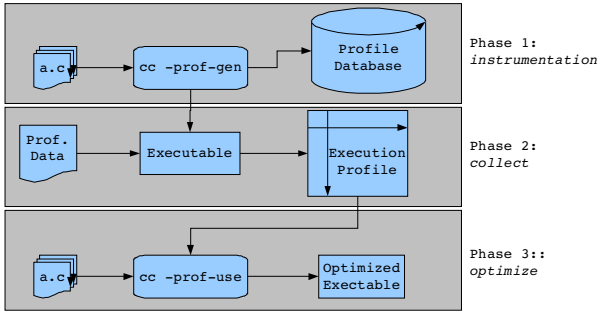
Node vs. Edge Profiles

```
int main(){
  int i;
  foo;
  for(i=0; i<5; i++)
    bar();
}

int bar() {
  int i,c,d;
  c=d=0;
  for(i=0; i<10; i++)
    if(i%3 == 0)
      c++;
    else
      d++;
  return c;
}
```



Two-Step Compilation



Profile Collection

- Instrumentation
  - Insert extra code to count the frequency of a particular event
  - Performed by the compiler or a post-compilation tool
  - Do we really have to measure every edge?
- Statistical sampling
- Hardware / simulator support
- Synthetic profiles (heuristics)

Loop Unrolling

- Important transformation to increase inter-iteration ILP
- Removes redundancies (compares, branches)
- Duplicate a loop body several times
  - preconditioning / postconditioning to handle trip counts mod n
  - preconditioning is not possible for data dependent loop exits
- Small loops are often completely unrolled

Preconditioning

Original Loop:

```
L: if -- goto EXIT
    <<body>>
    goto L
EXIT:
```

Unrolled by 4:

```
L: if -- goto EXIT
    <<body>>
   if -- goto EXIT
    <<body>>
   if -- goto EXIT
    <<body>>
   if -- goto EXIT
    <<body>>
    goto L
EXIT:
```

Preconditioned by 4:

```
   if -- goto EXIT
    <<body>>
   if -- goto EXIT
    <<body>>
   if -- goto EXIT
    <<body>>
L: if -- goto EXIT
    <<body>>
    <<body>>
    <<body>>
    <<body>>
    goto L
EXIT:
```

Postconditioning

Original Loop:

```
L: if -- goto EXIT
    <<body>>
    goto L
EXIT:
```

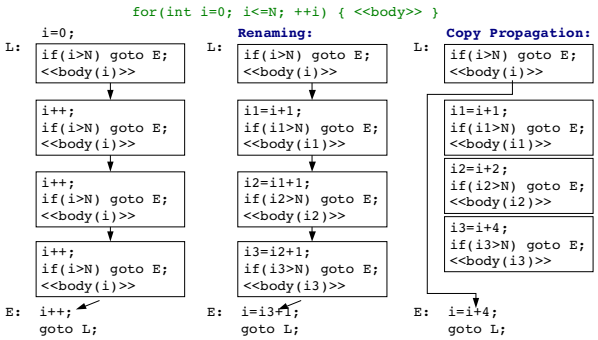
Unrolled by 4:

```
L: if -- goto EXIT
    <<body>>
   if -- goto EXIT
    <<body>>
   if -- goto EXIT
    <<body>>
   if -- goto EXIT
    <<body>>
    goto L
EXIT:
```

Postconditioned by 4:

```
L: if -- goto X
    <<body>>
    <<body>>
    <<body>>
    <<body>>
    goto L
X: if -- goto EXIT
    <<body>>
   if -- goto EXIT
    <<body>>
   if -- goto EXIT
    <<body>>
EXIT:
```


Loop Unrolling: Increasing Parallelism



Reductions

- Classic example for "false" loop dependencies
- **`a = a op fn(i)`** for a commutative *op*

```
    x=0.0;
    i=0;
L:  if(i > N) goto E;
    x=x+y[i]*z[i];
    goto L;
E:
```

```
    x=0.0;
    t1=t2=t3=t4=0.0;
    i=0;
L:  if(i > N) goto E;
    t1=t1+y[i]*z[i];
    i++;
    if(i > N) goto E;
    t2=t2+y[i]*z[i];
    i++;
    if(i > N) goto E;
    t3=t3+y[i]*z[i];
    i++;
    if(i > N) goto E;
    t4=t4+y[i]*z[i];
    i++;
    goto L;
E:
    x = t1+t2+t3+t4;
```


Loop Unrolling: Tradeoffs

- ✓ Unrolling effectively increases ILP and decreases loop overead (compares, branches)
- ✗ Increases code size (icache pressure)
- ✗ Increases register pressure
- ✗ Increases compile time
- Choosing the right amount of unrolling is hard (heuristics, canned recipes)

Next Time

- VEX toolchain tutorial
  - Compiler
  - Simulator
  - Benchmarks
  - Profiling
  - Machine Descriptions
- Solutions for assignment 1
- Assignment 2