A high-level implementation of software-pipelining for LLVM

A high-level implementation of software pipelining in LLVM

Roel Jordans [1], David Moloney [2]

[1] Eindhoven University of Technology, The Netherlands
r.jordans@tue.nl

[2] Movidius Ltd., Ireland

2015 European LLVM conference
Tuesday April 14th

Overview

Overview

Rationale

Implementation

Results

Conclusion

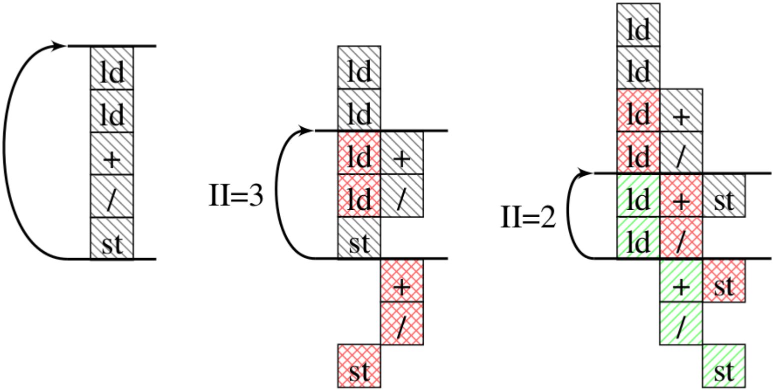
Rationale

Software pipelining (often Modulo Scheduling)

- Interleave operations from multiple loop iterations
- Improved loop ILP
- Currently missing from LLVM
- Loop scheduling technique
  - Requires both loop dependency and resource availability information
  - Usually done at a target specific level as part of scheduling
- But it would be very good if we could re-use this implementation for different targets

Example: resource constrained

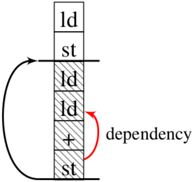
```
for(int i = 0; i < N; i++) {
    B[i] = (A[2*i] + A[2*i+1]) / 2;
}
```



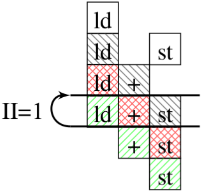
(a) Original (b) Single memory (c) Two memories

Example: data dependencies

```
B[0] = A[0];
for(int i = 1; i < N; i++) {
    B[i] = B[i-1] + A[i];
}
```




```
register int r = A[0];
B[0] = r;
for(int i = 1; i < N; i++) {
    r = r + A[i];
    B[i] = r;
}
```



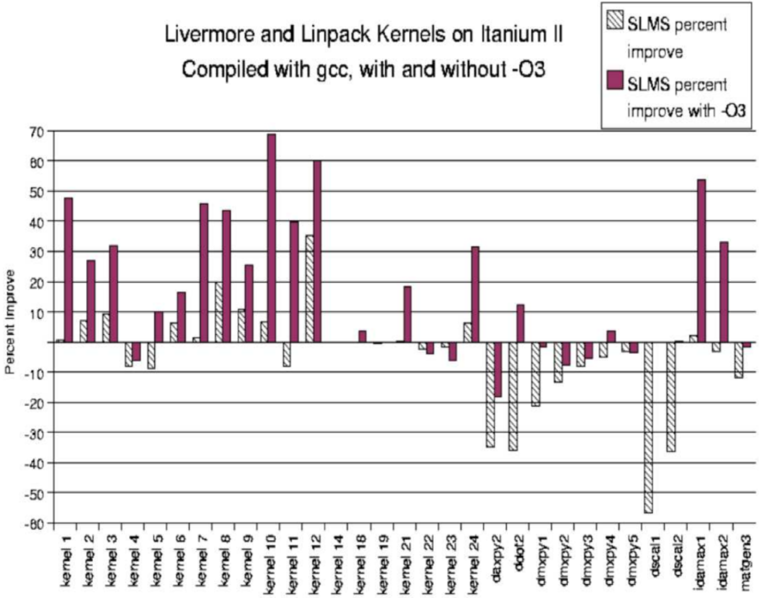
Source Level Modulo Scheduling (SLMS)

SLMS: Source-to-source translation at statement level

$$
\begin{array}{l}
for(i=0; i<n; i++) \\
\{ \\
S1_i : \quad t = \; A[i] * B[i]; \\
S2_i : \quad s = \; s + t; \\
\}
\end{array}
\quad \longrightarrow \quad
\begin{array}{l}
S1_0 : \quad t = A[0] * B[0]; \\
for(i=0; i<n-1; i++) \\
\{ \\
S2_i \quad : \quad s = \; s + t; \\
S1_{i+1} : \quad t = \; A[i+1] * B[i+1]; \\
\} \\
S2_{n-1} : \quad s = s + t;
\end{array}
$$

Towards a Source Level Compiler: Source Level Modulo Scheduling
– Ben-Asher & Meisler (2007)

SLMS results



SLMS features and limitations

- Improves performance in many cases
- No resource constraints considered
- Works with complete statements
- When no valid II is found statements may be split (decomposed)

This work

What would happen if we do this at LLVM's IR level

- More fine grained statements (close to operations)
- Coarse resource constraints through target hooks
- Schedule loop pipelining pass late in the optimization sequence (just before final cleanup)

Overview

Rationale

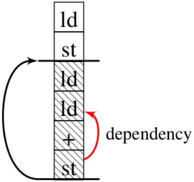
Implementation

Results

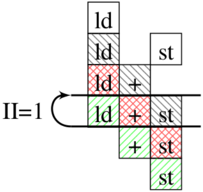
Conclusion

IR data dependencies

- Memory dependencies



- Phi nodes



Revisiting our example: memory dependencies

```
define void @foo(i8* nocapture %in, i32 %width) #0 {
entry:
  %cmp = icmp ugt i32 %width, 1
  br i1 %cmp, label %for.body, label %for.end

for.body:   ; preds = %entry, %for.body
  %i.012 = phi i32 [ %inc, %for.body ], [ 1, %entry ]
  %sub = add i32 %i.012, -1
  %arrayidx = getelementptr inbounds i8* %in, i32 %sub
  %0 = load i8* %arrayidx, align 1, !tbaa !0
  %arrayidx1 = getelementptr inbounds i8* %in, i32 %i.012
  %1 = load i8* %arrayidx1, align 1, !tbaa !0
  %add = add i8 %1, %0
  store i8 %add, i8* %arrayidx1, align 1, !tbaa !0
  %inc = add i32 %i.012, 1
  %exitcond = icmp eq i32 %inc, %width
  br i1 %exitcond, label %for.end, label %for.body

for.end:   ; preds = %for.body, %entry
  ret void
}
```

Revisiting our example: using a phi-node

```
define void @foo(i8* nocapture %in, i32 %width) #0 {
entry:
  %arrayidx = getelementptr inbounds i8* %in, i32 0
  %prefetch = load i8* %arrayidx, align 1, !tbaa !0
  %cmp = icmp ugt i32 %width, 1
  br i1 %cmp, label %for.body, label %for.end

for.body:    ; preds = %entry, %for.body
  %i.012 = phi i32 [ %inc, %for.body ], [ 1, %entry ]
  %0 = phi i32 [ %add, %for.body ], [ %prefetch, %entry ]
  %arrayidx1 = getelementptr inbounds i8* %in, i32 %i.012
  %1 = load i8* %arrayidx1, align 1, !tbaa !0
  %add = add i8 %1, %0
  store i8 %add, i8* %arrayidx1, align 1, !tbaa !0
  %inc = add i32 %i.012, 1
  %exitcond = icmp eq i32 %inc, %width
  br i1 %exitcond, label %for.end, label %for.body

for.end:    ; preds = %for.body, %entry
  ret void
}
```

Target hooks

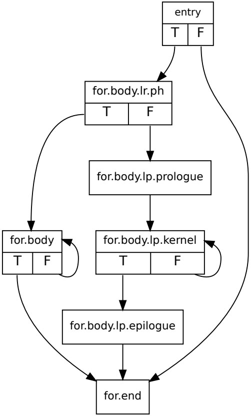
- Communicate available resources from target specific layer
- Candidate resource constraints
  - Number of scalar function units
  - Number of vector function units
  - ...
- IR instruction cost
  - Obtained from CostModelAnalysis
  - Currently only a debug pass and re-implemented by each user (e.g. vectorization)

The scheduling algorithm

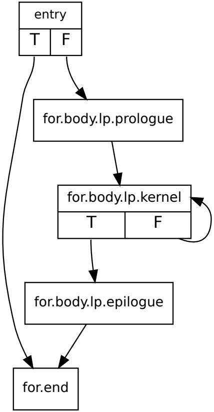
- Swing Modulo Scheduling
  - Fast heuristic algorithm
  - Also used by GCC (and in the past LLVM)
- Scheduling in five steps
  - Find cyclic (loop carried) dependencies and their length
  - Find resource pressure
  - Compute minimal initiation interval (II)
  - Order nodes according to 'criticality'
  - Schedule nodes in order

Swing Modulo Scheduling: A Lifetime-Sensitive Approach
– Llosa et al. (1996)

Code generation



CFG for 'loop5b' function



CFG for 'loop10' function

- Construct new loop structure (prologue, kernel, epilogue)
- Branch into new loop when sufficient iterations are available
- Clean-up through constant propagation, CSE, and CFG simplification

Overview

Target platform

- Initial implementation for Movidius' SHAVE architecture
- 8 issue VLIW processor
- With DSP and SIMD extensions
- More on this architecture later today! (LG02 @ 14:40)
- But implemented in the IR layer so mostly target independent

Results

- Good points:
  - It works
  - Up to 1.5x speedup observed in TSVC tests
  - Even higher ILP improvements
- Weak spots
  - Still many big regressions (up to 4x slowdown)
  - Some serious problems still need to be fixed
    - Instruction patterns are split over multiple loop iterations
    - My bookkeeping of live variables needs improvement
    - Currently blocking some of the more viable candidate loops

Possible improvements

- User control
  - Selective application to loops (e.g. through `#pragma`)
- Predictability
  - Modeling of instruction patterns in IR
  - Improved resource model
  - Better profitability analysis
  - Superblock instruction selection to find complex operations crossing BB bounds?

Overview

Conclusion

- It works, somewhat. . .
- IR instruction patterns are difficult to keep intact
- Still lots of room for improvement
  - Upgrade from LLVM 3.5 to trunk
  - Fix bugs (bookkeeping of live values, . . .)
  - Re-check performance!
  - Fix regressions
  - Test with other targets!

Thank you