

# A high-level implementation of software-pipelining for LLVM

**Citation for published version (APA):**

Jordans, R., & Moloney, D. (2015). A high-level implementation of software-pipelining for LLVM. In *EuroLLVM 2015 - European LLVM Conference, 13-14 April 2015, London, UK*

**Document status and date:**

Published: 01/01/2015

**Document Version:**

Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

**Please check the document version of this publication:**

- A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
- The final author version and the galley proof are versions of the publication after peer review.
- The final published version features the final layout of the paper including the volume, issue and page numbers.

[Link to publication](#)

**General rights**

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

[www.tue.nl/taverne](http://www.tue.nl/taverne)

**Take down policy**

If you believe that this document breaches copyright please contact us at:

[openaccess@tue.nl](mailto:openaccess@tue.nl)

providing details and we will investigate your claim.

# A high-level implementation of software pipelining in LLVM

Roel Jordans <sup>1</sup>, David Moloney <sup>2</sup>

<sup>1</sup> Eindhoven University of Technology, The Netherlands  
r.jordans@tue.nl

<sup>2</sup> Movidius Ltd., Ireland

2015 European LLVM conference  
Tuesday April 14th

# Overview

Rationale

Implementation

Results

Conclusion

# Overview

Rationale

Implementation

Results

Conclusion

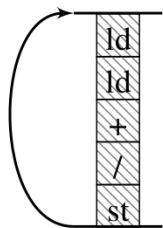
# Rationale

## Software pipelining (often Modulo Scheduling)

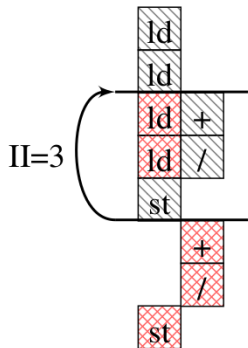
- ▶ Interleave operations from multiple loop iterations
- ▶ Improved loop ILP
- ▶ Currently missing from LLVM
- ▶ Loop scheduling technique
  - ▶ Requires both loop dependency and resource availability information
  - ▶ Usually done at a target specific level as part of scheduling
- ▶ But it would be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

## Example: resource constrained

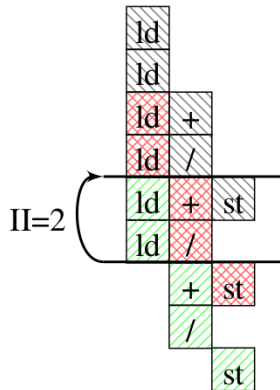
```
for(int i = 0; i < N; i++) {  
    B[i] = (A[2*i] + A[2*i+1]) / 2;  
}
```



(a) Original



(b) Single memory



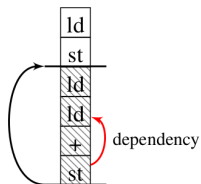
(c) Two memories

## Example: data dependencies

---

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

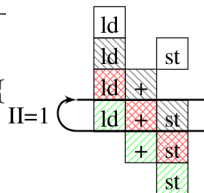
---



---

```
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

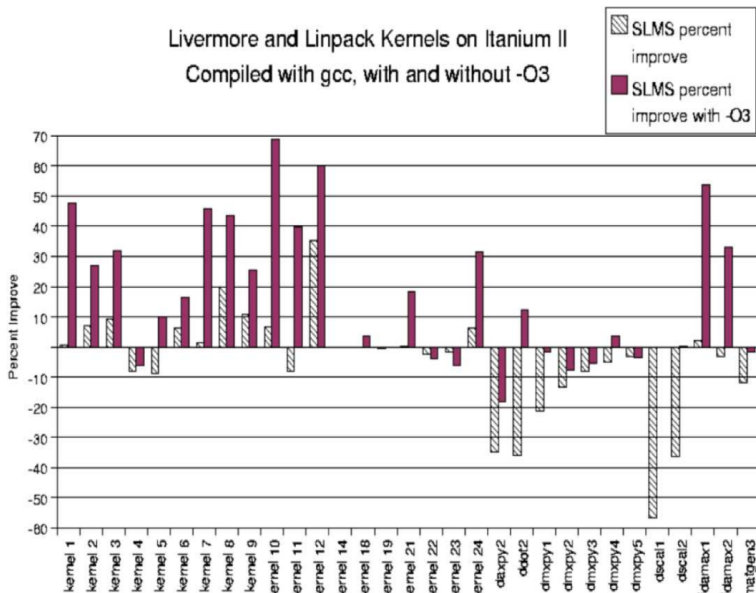
		$S1_0 : t = A[0] * B[0];$
$for(i = 0; i < n; i++)$		$for(i = 0; i < n - 1; i++)$
{		{
$S1_i : t = A[i] * B[i];$	$\longrightarrow$	$S2_i : s = s + t;$
$S2_i : s = s + t;$		$S1_{i+1} : t = A[i + 1] * B[i + 1];$
}		}
		$S2_{n-1} : s = s + t;$

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



# SLMS results

Livermore and Linpack Kernels on Itanium II  
Compiled with gcc, with and without -O3



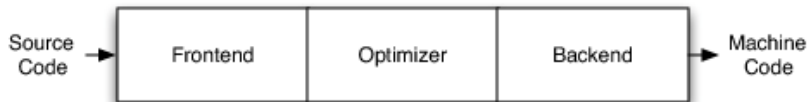
## SLMS features and limitations

- ▶ 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

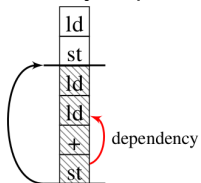
**Implementation**

Results

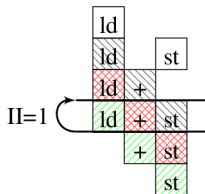
Conclusion

# 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)



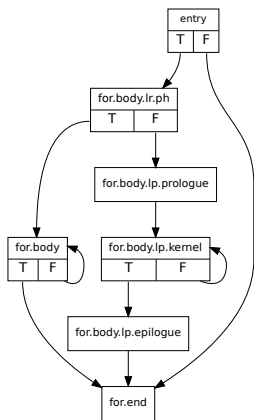
# The scheduling algorithm

- ▶ 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

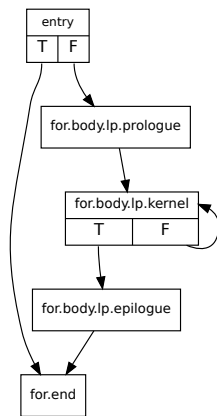
*Swing Modulo Scheduling: A Lifetime-Sensitive Approach*

– 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

Rationale

Implementation

Results

Conclusion

## 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

Rationale

Implementation

Results

Conclusion

# 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

TU/e

HIPEAC

Movidius 