High-level software-pipelining in LLVM

Roel Jordans
Eindhoven University of Technology
Eindhoven, The Netherlands
r.jordans@tue.nl

Henk Corporaal
Eindhoven University of Technology
Eindhoven, The Netherlands
h.corporaal@tue.nl

ABSTRACT

Software-pipelining is an important technique for increasing the instruction level parallelism of loops during compilation. Currently, the LLVM compiler infrastructure does not offer this optimization although some target specific implementations do exist. We have implemented a high-level method for software-pipelining within the LLVM framework. By implementing this within LLVM's optimization layer we have taken the first steps towards a target independent software-pipelining method.

Categories and Subject Descriptors

C.0 [**Computer Systems Organization**]: General—*modeling of computer architecture*; D.3.4 [**Programming Languages**]: Processors—*compilers, optimization*

General Terms

Performance

Keywords

LLVM, software-pipelining

1. INTRODUCTION

Embedded systems are nowadays often used in situations where both high-performance processing and low energy consumption are critical. This has led to the design of highly specialized processor hardware which often obtains a large part of its efficiency from high instruction-level parallelism. Very-long instruction-word (VLIW) processors are a prime example of such processors and many DSP-like processors [1] incorporate VLIW characteristics internally. It is the task of the compiler to find and utilize this ILP as part of the application compilation process. One key optimization in the compilation process is software-pipelining [6]. Software-pipelining is a scheduling technique which schedules loop code in such a way that independent operations from different loop iterations can be scheduled in parallel.

SCOPES '15, June 1–3 2015, Sankt Goar, Germany
 ISBN 978-1-4503-3593-5/15/06 ...$15.00
DOI: http://dx.doi.org/10.1145/2764967.2771935.

Figure 1: Compilation in three stages

The design of such an optimizing compiler is commonly separated into three layers of increasing architectural detail as is shown in figure 1. The *frontend* translates the input language into an intermediate representation (IR). This IR is then optimized by the target independent *optimizer*, and finally translated into actual assembly or binary code by the *backend*. Traditionally, implementing a software-pipelining algorithm into a compiler is within the target specific backend code [3, 4, 5]. All details of the program's instructions, together with a detailed view of the processor architecture are known at this stage. This allows for highly detailed scheduling decisions and produces good optimized code in general. However, such a backend implementation is usually highly target specific which makes it difficult to re-use the software-pipelining implementation across different architectures.

An alternative implementation was proposed by Ben-Asher and Meisler [2]. They demonstrated the effects of implementing source-level modulo-scheduling. Doing so places the software-pipelining algorithm just before the frontend stage of the compiler. No information about the hardware resources of the processor is available yet at this level which frequently makes the obtained schedules inefficient, as could be observed in their experimental results. Their source-level approach was able to find several significant improvements but also presented equally large regressions in several cases.

In this paper we consider finding the middle ground between both options. By placing the software-pipelining algorithm at the end of the optimizer stage we are able to use the generic IR instructions, and combine them with some basic information on the available resources from the target backend, in order to achieve a more accurate but still target independent solution.

The remainder of this paper is organized as follows, first we introduce the basic concepts of software-pipelining in section 2. Then we present our implementation considerations in section 3 and finalize with a discussion on our initial experiences with this approach in section 4.

2. SOFTWARE-PIPELINING

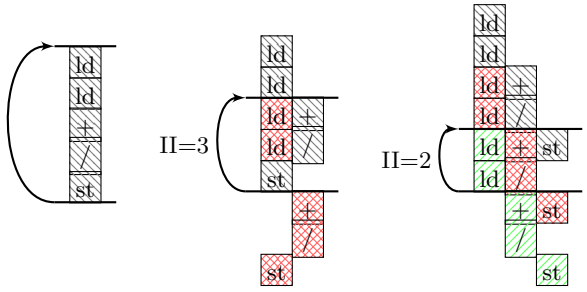
Software-pipelining is a loop scheduling technique aimed at increasing the instruction level parallelism by scheduling operations from different loop iterations in an overlapping fashion. Most commonly, software-pipelining is implemented through a technique called modulo scheduling [2, 4, 6, 9].

In a modulo-scheduled loop kernel, the operations of the original loop body are overlapped such that there is a fixed *initiation interval* between the start of consecutive loop iterations which is smaller than the total length of the original loop body. This initiation interval (II) is constrained by two factors; the available resources in the processor, and loop carried dependencies in the code.

Listing 1 and figure 2 illustrate the effect of a resource constraint on the II, in this case the number of parallel load-store operations that can be executed. Operations from different loop iterations are distinguished by their background color and texture. Only the kernel operations are shown in these example schedules, address calculation and control-flow operations are hidden for brevity and it is assumed that arrays `A` and `B` do not overlap. Figure 2b shows that the minimal II is three cycles if only one load-store operation is allowed in parallel, while figure 2c demonstrates an II of two cycles if two load-store operations are allowed in parallel.

Listing 1: Example loop nest showing an initiation interval constrained by the number of available load-store unit(s).

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



(a) Original (b) Single load-store (c) Dual load-store

Figure 2: Simplified schedules of the loop shown in listing 1 showing the original sequential schedule and two software-pipelined versions demonstrating the influence of a resource constraint.

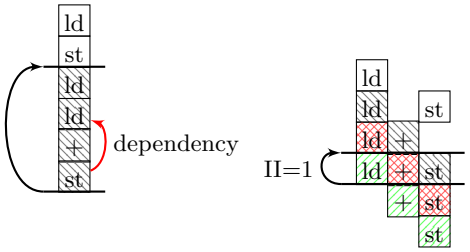
The second constraint to the II comes from the semantics of the application in the form of recurring values. Such recurring values can occur in two forms; memory carried dependencies and register carried dependencies. Listings 2 and 3, together with figure 3, illustrate their differences. The original schedule shows the inter-iteration dependency which constrains software-pipelining as the store needs to have completed before the value can be loaded back. The transformed schedule avoids this by storing a copy of the value in a register, avoiding the II constraining load. As a result, the transformed version could be pipelined with a single-cycle II whereas this was impossible in the original code.

Listing 2: Example loop nest showing an initiation interval constrained by a loop caried dependency.

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

Listing 3: Restructured version of the code shown in listing 2, breaking the loop caried dependency by storing the intermediate result into a register.

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



(a) Original (b) Transformed

Figure 3: Simplified schedules for the original (listing 2) and transformed (listing 3) version of a loop showing an inter-iteration dependency.

After determining the minimal II, a modulo scheduling algorithm will usually attempt to schedule the loop kernel with that II as an input. If it fails it will increment the II and try again until either a schedule is found or the II grows beyond the schedule length of the original non-pipelined loop and no pipelined schedule exists.

3. IMPLEMENTATION

Although several implementations of software-pipelining have been published before, none of these is available in the most recent version of the LLVM framework. Either they have been lost in previous restructuring of the schedulers [7], or they are very target specific implementations [3, 5, 9].

In our implementation we have chosen to use the swing modulo scheduling algorithm [8]. This algorithm is very efficient at finding good software-pipelined schedules, and was previously also used in e.g. LLVM [7] and GCC [4].

The swing modulo scheduling algorithm operates in five steps; *a*) find cyclic (loop carried) dependencies and their length; *b*) find resource pressure; *c*) compute minimal initiation interval (II); *d*) order nodes according to critical-

ity ; and *e*) schedule nodes in this order, either as-soon-as-possible or as-late-as-possible based on the status of their dependencies. For full details on the swing modulo scheduling algorithm please refer to [8] which has an excellent set of examples to illustrate the approach.

The LLVM IR is a low-level abstract representation of the program. It uses a basic set of operations that often translate directly into processor operations, however, some more complex operations also exist. One example of such a more complex operation is the `getelementpointer` operation, or GEP in short. These operations can perform complex address calculations although they also frequently reduce to only a single operation or even a constant value. LLVM offers a `TargetTransformInfo` interface which provides information about the cost of specific operations on the processor architecture, as well as, information about the available features of the processor architecture. Using this cost information through LLVM s `CostModelAnalysis` allows us to estimate scheduling information such as the length of loop carried dependencies.

As shown in figure 3, loop carried dependencies can exist in two forms. Memory carried dependencies can be recognized in IR code by using LLVM s `DependencyAnalysis` to check if a pair of a store and load operation may address the same memory location across subsequent loop iterations. If this is the case, then a memory carried dependency exists. Currently our implementation will not accept such loops and these need to be transformed to the second form before applying software-pipelining. We assume that such dependencies are translated into register carried dependencies by an earlier optimization pass. Luckily, most of LLVM s transformations, including loop vectorization, already produce loops in the second form. The second form uses a register to explicitly represent loop carried dependencies.

Listing 4: Memory carried 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.0 = phi i32 [%inc, %for.body], [1, %entry]
  %sub = add i32 %i.0, -1
  %idx = getelementptr inbounds i8* %in, i32 %sub
  %0 = load i8* %idx, align 1, !tbaa !0
  %idx1 = getelementptr inbounds i8* %in, i32 %i.0
  %1 = load i8* %idx1, align 1, !tbaa !0
  %add = add i8 %1, %0
  store i8 %add, i8* %idx1, align 1, !tbaa !0
  %inc = add i32 %i.0, 1
  %exitcond = icmp eq i32 %inc, %width
  br i1 %exitcond, label %for.end, label %for.body

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

The LLVM IR is a static single assignment (SSA) representation of the program. Each operation in the IR creates a new value in a unique virtual register which may only be assigned once. Phi-nodes are used in order to cope with merge-points in the control-flow graph (such as introduced by if-statements and the back-edges of loops). Listing 4 shows an example loop with a memory carried dependency. In this example we can observe that the address computation of both GEP operations addresses subsequent locations which are then used by a store-load pair to create a memory dependency. To analyze this, we need to consider the loop induction variable `%i.0`, its increment direction, and the address computation of both `%idx` and `%idx1`, which is all achieved using the existing `DependenceAnalysis`.

Once we have excluded loops with memory carried dependencies we are only left with those loops that have either no dependencies or only register carried ones. For these loops we compute the length of the cyclic dependency using the cost model. Listing 5 shows the same loop implemented using a register carried dependency. Here we observe that there is one less load operation in the loop, which has been replaced by a new phi operation. The minimal recurrence II is now computed by finding the cycle `%i.0` `%add` `%i.0`, and computing its weight. All cycles in the operation graph are enumerated and the longest cycle represents the minimal recurrence II.

Listing 5: After promoting the memory dependency to a register carried dependency

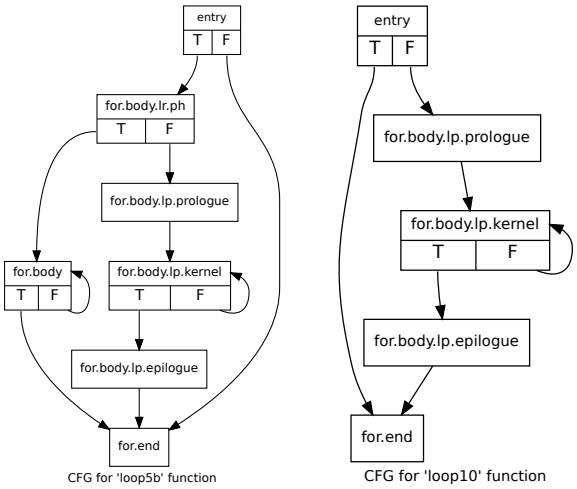
```
define void @foo(i8* nocapture %in, i32 %width) #0
{
entry:
  %idx = getelementptr inbounds i8* %in, i32 0
  %pre = load i8* %idx, align 1, !tbaa !0
  %cmp = icmp ugt i32 %width, 1
  br i1 %cmp, label %for.body, label %for.end

for.body:    ; preds = %entry, %for.body
  %i.0 = phi i32 [%inc, %for.body], [1, %entry]
  %0 = phi i32 [%add, %for.body], [%pre, %entry]
  %idx1 = getelementptr inbounds i8* %in, i32 %i.0
  %1 = load i8* %idx1, align 1, !tbaa !0
  %add = add i8 %1, %0
  store i8 %add, i8* %idx1, align 1, !tbaa !0
  %inc = add i32 %i.0, 1
  %exitcond = icmp eq i32 %inc, %width
  br i1 %exitcond, label %for.end, label %for.body

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

From this point, we compute the minimal resource II using two new hooks in the `TargetTransformInfo`. These hooks represent the number of available execution resources for scalar and vector operations respectively. At this point we assume that the processor architecture is capable of executing either scalar or vector operations on each issue-slot, as this was the case for our initial target architecture. This model may be extended in the future when support for other architectures is considered.

With both the minimal recurrence and resource based II values, we can now start the actual node ordering and scheduling steps. These steps again use the cost model and the new `TargetTransformInfo` hooks to determine a software-pipelined schedule.

From this schedule, we then generate a loop prologue, kernel, and epilogue, in IR form and connect them to the original code together with a conditional block that checks if there are sufficient loop iterations to satisfy the requirements for the prologue. This results in the loop structure shown in figure 4a. However, in many cases the loop bypass from the (top) entry block checks the same, or a very similar,

condition as the newly inserted guard block does. In such cases, running LLVM s constant propagation, common sub-expression elimination, and control-flow graph simplification can help clean-up the new structure. Figure 4b illustrates the effect on one of our example loops.



(a) Generated structure (b) After cleanup

Figure 4: The generated loop structure before (a) and after (b) constant propagation and control-flow graph simplification when sufficient iterations are guaranteed

In order to make sure that we can optimally make use of LLVM s existing optimizations, while still making reasonable cost estimates, we have scheduled our high-level software-pipelining pass at the end of the optimization pipeline. This puts it after the loop vectorization and leaves only the above mentioned cleanup passes in between the pipelining algorithm and the target backend.

4. CONCLUSIONS

In some of our initial experiments we have observed that our high-level software-pipelining approach can indeed bring good improvements(up to 1.5x speedup) to some of the TSVC benchmarks when running on the Movidius SHAVE architecture [1]. However, we also found several large regressions and there are still some bugs in our implementation which currently prevent real experimentation.

Most of these regressions were found to be caused by the software-pipelining algorithm breaking up operation sequences that would otherwise be recognized as complex DSP operations. One way to avoid this is by incorporating more of these operations as intrinsics in the IR before attempting software-pipelining. That would force the scheduler to consider these operation patterns as single operations and would help mitigate these regressions.

The remaining bugs in our code mainly have to do with proper book-keeping of live values when operations from three or more loop iterations are scheduled in parallel in the loop kernel. This currently prevents the compiler from correctly generating code for loops that are most likely to benefit from software pipelining. We therefore expect that the final performance improvement after fixing these bugs will significantly exceed the currently observed benefits.

Further improvements of the approach may be found in tuning or extending the set of `TargetTransformInfo` hooks and possibly by adding heuristics that estimate register file pressure. As such, these will be considered as part of our future work.

5. ACKNOWLEDGMENTS

The authors would like to acknowledge the HiPEAC network for supporting this research through a collaboration grant, as well as, Martin O Riordan and David Moloney of Movidius Ltd. for providing the opportunity for, and their support while working on, this project.

6. REFERENCES

[1] B. Barry, C. Brick, F. Connor, D. Donohoe, D. Moloney, R. Richmond, M. O Riordan, and V. Toma. Always-on vision processing unit for mobile applications. *IEEE Micro*, 35(2):56 66, 2015.

[2] Y. Ben-Asher and D. Meisler. Towards a source level compiler: Source level modulo scheduling. In *Program analysis and compilation, theory and practice*, pages 328 360. Springer, 2007.

[3] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson. Legup: An open-source high-level synthesis tool for fpga-based processor/accelerator systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(2):24, 2013.

[4] M. Hagog and A. Zaks. Swing modulo scheduling for gcc. In *Proceedings of the 2004 GCC Developers Summit*, pages 55 64, 2004.

[5] N. Kim and A. Krall. Integrated modulo scheduling and cluster assignment for ti tms320c64x+ architecture. In *Proceedings of the 11th Workshop on Optimizations for DSP and Embedded Systems*, pages 25 32. ACM, 2014.

[6] M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. *ACM SIGPLAN Notices*, 23(7):318 328, 1988.

[7] T. M. Lattner. An implementation of swing modulo scheduling with extensions for superblocks. Master s thesis, University of Illinois at Urbana-Champaign, 2005.

[8] J. Llosa, A. González, E. Ayguadé, and M. Valero. Swing module scheduling: a lifetime-sensitive approach. In *Parallel Architectures and Compilation Techniques, 1996., Proceedings of the 1996 Conference on*, pages 80 86. IEEE, 1996.

[9] Z. Zhang and B. Liu. Sdc-based modulo scheduling for pipeline synthesis. In *Proceedings of the International Conference on Computer-Aided Design*, pages 211 218. IEEE Press, 2013.