Integrated Modulo Scheduling and Cluster Assignment for TI TMS320C64x+ Architecture [*]

Nikolai Kim Andreas Krall
Institute of Computer Languages,
Vienna University of Technology
{kim,andi}@complang.tuwien.ac.at

ABSTRACT

For the exploitation of the available parallelism clustered Very Long Instruction Word (VLIW) processors rely on highly optimizing compilers. Aiming this parallelism, many advanced compiler concepts have been developed and proposed in the past. Many of them concentrate on loops only as most of the execution time is usually spent executing repeating patterns of code. Software pipelining techniques, such as modulo scheduling, try to speed up the execution of loops by simultaneous initiation of multiple iterations, thus additionally exploiting parallelism across loop iteration boundaries. This increases processor utilization at the cost of higher complexity which is especially true for architectures featuring multiple clusters and distributed register files. Additional scheduling constraints need to be considered in order to produce valid schedules. Targeting TI's TMS320C64x+ clustered VLIW architecture, we describe a code generation approach that adapts an iterative modulo scheduling scheme, and also propose two heuristics for cluster assignment, all together implemented within the popular LLVM compiler framework. We cover implementation of developed algorithms, present evaluation results for a selection of benchmarks popular for embedded system development and discuss gained insights on the topics of integrated modulo scheduling and cluster assignment in this paper.

Keywords

VLIW, instruction level parallelism, modulo scheduling, cluster assignment, phase ordering, integer linear programming, LLVM

1. INTRODUCTION

Software pipelining is an effective loop scheduling technique for exploiting instruction level parallelism. The main idea is to restructure loops for the overlapped execution of multiple iterations and to issue them at a computed, constant rate. This issue rate, called *initiation interval* (*II*), depends on the loop properties and is subject to additional restrictions. *Resource constraints* restrict the initiation interval regarding the number and availability of functional units on the target machine, *recurrence constraints* require *II* that no data dependences, that span multiple loop iterations, are violated. Thus, for the highest performance in terms of pipeline throughput, the *II* is the smallest value that satisfies all of these constraints [19] [1].

Conditional control flow within a loop complicates scheduling in general. To prevent applicability to extremely simple loops only, software pipelining techniques need therefore to be capable of dealing with it. One solution is hierarchical reduction [13] which introduces artificial pseudo operations for conditional constructs and allows scheduling them in the usual manner. After scheduling, conditional code corresponding to these constructs is regenerated. While not requiring special hardware features, this leads to a code size increase because of code duplication during the regeneration step. Another way to deal with conditionals within a loop is to remove them entirely by a forward *if-conversion* [9]. Given hardware support for predication, conditional branches are removed by predicating and merging instructions within branch destination blocks. Iterative application of if-conversion helps decreasing the amount of control dependences, at the expense of higher register pressure however, since the number of live registers in the resulting block is increased.

Independent of the conditional control flow, restructuring the body of a loop into overlapping stages creates additional constraints for register reuse. In order to avoid conflicts between pipeline stages covered by a particular data dependency, allocation of additional registers is required in many cases. In absence of special hardware support, such as *rotating register files* [10], this issue is traditionally resolved by *modulo variable expansion* (*MVE*). First, the amount of kernel unrolling is computed by identifying register lifetimes exceeding *II* in length. After unrolling the kernel, conflicting registers are renamed for each stage within the conflict range. Lam in [13] describes modulo variable expansion in more detail, Dinechin [4] additionally addresses complications with respect to the general pipeline construction.

Along with established heuristic approaches, various optimal solutions to instruction scheduling have been proposed in the past. Ertl and Krall [8] produce optimal schedules for the Motorola 88100 RISC processor using constraint logic

[*]This work is supported by the Austrian Science Fund (FWF) under contract P21842, *Optimal Code Generation for Explicitly Parallel Processors.*

programming. Wilken et al. [25] follow an integer linear programming approach for scheduling single basic blocks and propose Dependence Graph (DG) transformations, such as partitioning, linearization and edge elimination, profitable for scalability and effective for solver time improvement. Shobaki et al. extend the scheduling scope to superblocks [21] and traces [22], and propose a solution that finds optimal region schedules through enumeration. Eriksson [7] addresses optimal modulo scheduling, describes an integer linear programming formulation specific to the TMS320C64x+ architecture, and further discusses on topics of instruction selection- and cluster assignment-integration. Altman provides an enumeration scheme for finding optimal modulo schedules and contrasts obtained timing results to the results based on integer linear programming [2].

While optimal, most of those approaches are not tractable for production compilers due to a high time complexity and exponential scaling, and are therefore limited to small problem instances. On the other hand, many heuristic solutions perform nearly optimal, at the same time featuring a comparably low processing time overhead. Huff [12] describes a solution that mainly aims at lowering register pressure for produced modulo schedules. This is achieved through the computation of a *slack* which represents a range of possible placement slots for a particular instruction. After computation, this range is scanned and the corresponding instruction heuristically placed either as early or as late as possible, depending on the already scheduled nodes and for the preference of short register lifetimes. Another approach, known as *swing scheduling*, is presented by Llosa et al. as part of a production compiler [15]. The proposed solution adapts an iterative scheme and, similar to slack scheduling, also uses precomputed parameters, such as valid placement slots and mobility values, to assist in scheduling. In contrast to Huff's algorithm, no backtracking is used for the profit of lower processing times. Eichenberger et al. describe *stage scheduling* [5], a backtracking variant that attempts to lower register requirements by moving instructions across pipeline stages.

Clustered VLIW architectures present a number of advantages. Clustering simplifies on-chip wiring, increases density and scalability due to a better chip area utilization, lowers production costs and reduces power consumption. For compilers clustering means an additional code generation step that is responsible for distributing the workload over all available machine resources, and is therefore critical to overall system performance. Early work of Ellis [6] gives a good introduction into the problem and proposes a greedy bottom-up *BUG* solution that tries to minimize the number of intercluster transfers by assigning DG nodes in the latency-weighted, depth-first manner. List scheduling is then applied that is adapted to respect cluster assignments. Since both steps are performed more or less independent of each other in different compiler phases, scheduling is eventually restricted more than necessary by prior clustering decisions. Addressing this issue, Özer et al. [17] describe *UAS*, an algorithm that unifies the process of cluster assignment with instruction scheduling. The main idea is to extend the resource model by a priority list of available clusters and intercluster busses, and to check this list during greedy list scheduling. Exposing cluster assignment choices to the scheduler is shown to be profitable, since generally more efficient schedules are generated by UAS when compared to

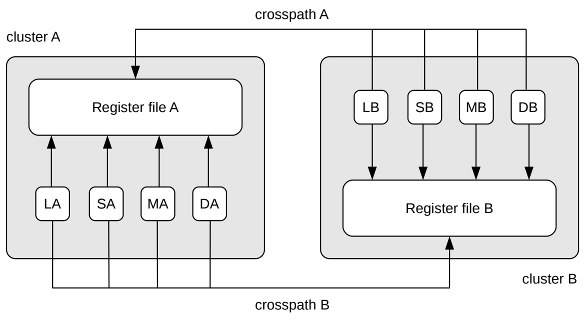

Figure 1: TI C64x clustered register file and functional units

BUG. Stotzer et al. [23] provide basic insights into cluster assignment in the context of modulo scheduling and present an adaptation of slack scheduling to the TMS320C6x VLIW target. However, no clustering algorithm is given, the scheduler is set up to operate on already cluster-assigned nodes, which in turn is done by hand as preprocessing. Sanchez et al. [20] employ UAS within a swing scheduler implementation in order to perform modulo scheduling and cluster assignment in one step. Additionally, loop unrolling is selectively applied for the reason of lowering the pressure on intercluster paths. Fernandes et al. [11] describe *distributed modulo scheduling*, an alternative integrated approach that sequentially uses three strategies for cluster assignment. The first strategy tries to assign a node to a cluster without involving explicit intercluster transfers. If such an assignment is not possible, the second strategy is attempted, which is to resolve cluster requirements through insertion of copies. If that in turn fails, the node is arbitrarily assigned to any of the available clusters and backtracking is applied in order to unschedule some of the node's predecessors. Nystrom et al. [16] also concentrate on clustering and propose a cluster assignment algorithm that produces graph annotations suitable to serve as input to any traditional modulo scheduling algorithm. So, for example, a swing modulo scheduler is used by the authors for evaluation.

2. CONTRIBUTION

In this paper we propose an integrated solution for generating modulo schedules targeting TI's TMS320C6000 signal processor family. The optimization combines an adaptation and extension of the swing scheduling scheme [15] with fast heuristics for cluster assignment, and is entirely implemented as part of an experimental backend within the LLVM [26] compiler framework. We cover major implementational details of proposed algorithms, discuss obtained results, and relate them for comparison to available alternative solutions.

3. TARGET ARCHITECTURE

For our implementation we targeted TI's TMS320C6000 DSP family [27] and concentrated especially on C64x and C64x+ VLIW CPU's. These CPU's provide predication support for almost every non-compact, non-call instruction, and feature two clusters (*A* and *B*). Each cluster contains four functional units: *L*, *S*, *M*, and *D*. Units *L* and *S* are the integer arithmetic/logic units (ALUs), with *S* units being additionally responsible for control flow changes (branches).

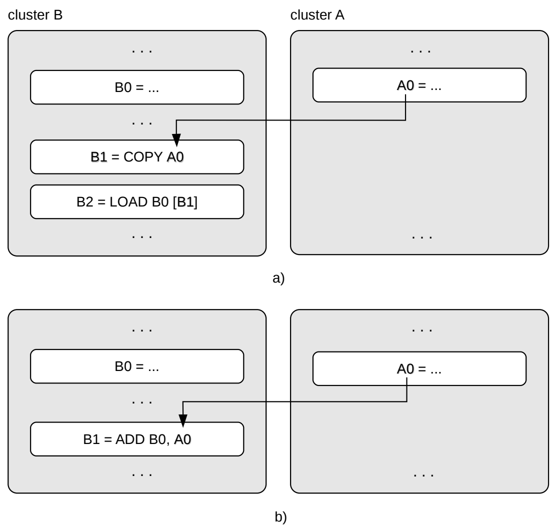

Figure 2: Source operand access: a) because LOAD is not allowed to use crosspaths for address operands, register A0 is explicitly copied to cluster B; b) register A0 is not copied but is referenced by ADD directly via crosspath

Loads, stores, as well as complex address calculations are performed on *D* units. Unit *M* is exclusively used for different multiplication variants together with specialized DSP instructions. Given 2 clusters with 4 functional units in each of them, up to 8 operations can therefore be executed in parallel. Each cluster is connected to its own register file, each file contains 32 general purpose registers, 3 of them (*A0-A2*, resp. *B0-B2*) can additionally be used as predicate registers. Figure 1 depicts the general organization of our target.

For intercluster transfers two directed *crosspaths* exist. Within an execution packet, all operations that are allowed to use a crosspath and are assigned to a particular cluster (f.e. *A*) are restricted to access at most one register in the opposite register file (i.e. *B*). Thus, at most two different crosspath register accesses are permitted per cycle. For instructions that are not allowed to use crosspaths, required data must be copied explicitly to the desired cluster. Figure 2 shows both intercluster transfer variants. One specialty of the target is the presence of *crosspath stalls.* In case a crosspath is used to access a register that has been written in the previous cycle, a hardware stall occurs. An example is given in Figure 2 b): if register *A0* is set in cycle *n* and is read via crosspath in cycle *n + 1*, the execution is delayed by 1 cycle.

C64x+ CPU's additionally feature a hardware buffer that is dedicated to software pipelining and offers space for up to 14 instruction packets. The buffer is controlled by special instructions and requires *II* as well as the number of iterations to be specified explicitly. Additionally, the end of the kernel needs to be marked in order to drain the pipeline properly. Since implemented in hardware, this buffer entirely eliminates the latency of the trailing branch and does not call for any special actions about pipeline construction. Unfortunately, the size limit of 14 packets prevents a general application and allows this buffer to be used for small loops only.

4. IMPLEMENTATION

This section is structured as follows. First, a brief description of if-conversion will be given, which we implemented for preprocessing prior to scheduling. We then describe modulo scheduling extensions, and depict our clustering solutions, addressing a simple naive implementation first, and presenting extended integrated implementation at the end of this section.

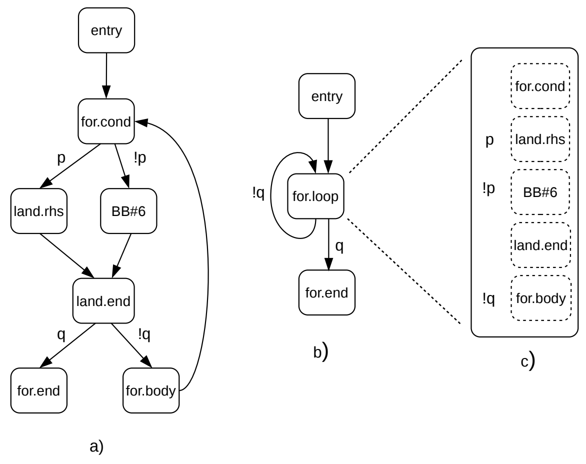

Figure 3: If conversion: a) original CFG; b) loop structure after two conversion steps; c) predicated loop content

4.1 If-conversion

If-conversion is a compilation technique that eliminates conditional branches and increases the size of basic blocks by using predication. Extensive description and discussion on predication can be found in [18] and [9]. Here we only mention the basic idea, which is to encode a special flag into a machine instruction that determines at runtime whether this instruction is to be executed or ignored.

For our experiments we have implemented a simple if-conversion variant, that iteratively collapses patterns of basic blocks based on profitability heuristics. While being simple, such an iterative design is effective for producing single block loops as shown in Figure 3. Given a CFG together with predicates as branching conditions (p and q) in a), two conversion steps are performed to produce a self-looping basic block b). The first step collapses a diamond consisting of *for.cond*, *land.rhs*, *BB#6* and *land.end*. The result is then merged with *for.body* to produce *for.loop*. The predicated content of the transformed loop is depicted in c). As an addition to the if-converter, *tail duplication* has been implemented, which further increases conversion scope, and allows more aggressive transformations.

Alternative if-conversion implementations are found in [14] and [3]. Warter et al. discuss benefits of if-conversion in the context of software pipelining in detail in [24].

4.2 Modulo scheduler

For the task of modulo scheduling we adapted and extended a heuristic *swing* algorithm originally proposed by Llosa [15]. The algorithm computes $ALAP$ (As Late As Possible), the latest possible issue slot respecting successors in the DG, and $ASAP$ (As Soon As Possible), the earliest issue slot in dependence on predecessors, parameters which assist in scheduling decisions and aim at minimizing total register requirements. Additionally, a mobility function MOB is provided which represents a range of valid issue slots for a node, and is used in case no rational priority decision is possible (f.e. when two nodes have the same $ALAP$ resp. $ASAP$ values). However, it is possible for two nodes to have the same mobility value as well. For that we extend the original heuristic by a balancing factor that assigns priorities based on the number of supporting functional units. For example, a *move* instruction can be executed on any of the L, S, D units, a sign extension *ext* in turn is only executable on S units. Thus, *move* will be given a lower priority leading to the sign extension being processed first.

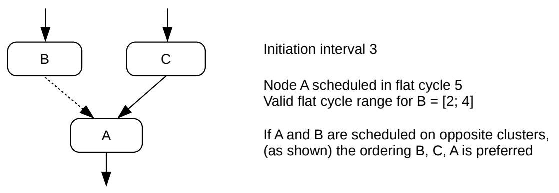
B
C
A
Initiation interval 3
Node A scheduled in flat cycle 5
Valid flat cycle range for B = [2; 4]
If A and B are scheduled on opposite clusters,
(as shown) the ordering B, C, A is preferred

Figure 4: Example of cycle reordering for crosspath stall reduction

As mentioned in Section 3, one source for a potential performance degradation that is specific to our target, is the presence of crosspath stalls. Thus, for a high performance minimization of such stalls is desirable. To deal with this issue, another extension is provided to the scheduler which becomes active during the final scheduler run. Given a valid cycle range for a particular instruction to be scheduled, a *cycle reordering* is performed. Each cycle within the obtained range is inspected and a cycle distance to each DG successor scheduled on opposite cluster is calculated. A distance of one cycle signals a crosspath stall detection, in this case the cycle currently inspected is given a lower priority. Given a DG fragment consisting of nodes A, B and C as shown in Figure 4, assume (for II=3) the node A to be scheduled in cycle 5 which translates to the modulo resource table row 2 (cycle modulo II). Now, when scheduling node B, a cycle range [2; 4] is obtained which is processed in a reverse order (to keep register lifetimes short). Cycle 4 translates to row 1, the distance between A and B equals 1, thus, assigning B to this slot would introduce a stall. Cycles 3 and 2 do not expose any crosspath conflicts, therefore a new cycle priority order is established which is 3, 2, 4. Note, that only cycles need to be considered which result in distances less than *II*. Dependences longer than *II* are ignored, since subject to modulo variable expansion later.

For the purpose of resource tracking we utilize a modulo resource table which is parameterizable to support multiple executions of the scheduler. During the first run, no crosspath constraints are considered, instructions are greedily placed in the first available cycle/functional unit slot found. After the step of cluster assignment, the scheduling is eventually repeated, in this case a modulo resource table is used which is now set up to respect the mapping of instructions to clusters and to track the use of crosspaths.

Figure 5 shows in a) a flat schedule for a simple loop used

0:
1: B6 = A13
1: A6 = mvk 31
2: B10 = mvk 0
2: **A5 = shl B6, 1**
2: **B7 = cmpgt B6, A6**
3: A9 = ext A5, 2
3: **A8 = B7**
4: store A0, A9, B10
4: B12 = mvk 1
4: B11 = addm B14, B6
4: **A3 = add B6, 1**
5: A13 = A3
5: store B11, 1, B12
6: brcond, A8

a)

	cycle 0	cycle 1	cycle 2
LA	**A8 = B7**	**A3 = add B6, 1**	A13 = A3
SA	A9 = ext A5, 2	A6 = mvk 31	**A5 = shl B6, 1**
MA			
DA		store A0, A9, B10	
LB		**B6 = A13**	**B7 = cmpgt B6, A6**
SB	brcond, A8	B12 = mvk 1	B10 = mvk 0
MB			
DB		B11 = addm B14, B6	store B11, 1, B12
xA		x	x
xB	x	x	x

b)

Figure 5: a) flat schedule example; b) corresponding modulo resource table

to zero-initialize an array[1] and in b) a corresponding modulo resource table for II=3. LA-DB represent functional units L-D within corresponding clusters A-B, and xA, xB capture occupation of intercluster busses in the current instruction packet. Instructions that use a crosspath are boldfaced. As can be seen, a left-shift instruction *shl* in cycle 2 references register *B6* that is set in cycle 1. This extends the runtime of the depicted schedule by 1 cycle due to the mentioned architectural crosspath restriction. The compare instruction *cmpgt* also involves an implicit intercluster operand transfer (register $A6$). Here however, the crosspath stalls do not accumulate, since *shl* and *cmpgt* are scheduled to be issued together in the same cycle.

4.3 Cluster assignment

4.3.1 Simple heuristic

For cluster assignment to be used with the swing scheduler two models are provided. The first model implements a simple top-down heuristic that processes the entire DG at once and is therefore only loosely coupled with the scheduling routine.

Since operating on a preliminary schedule which is correct in terms of the data-flow, loop-carried dependences are eliminated and DG nodes put in ascending order (while various criterias are possible here, we used node's depth for ordering). Given this priority order, nodes are then processed with assignment preference of a least number of register copies. This is achieved by inspecting predecessors of a given node that already have been assigned. Given their cluster distribution, and taking crosspath access possibility for each predecessor into account, copy counts are estimated, which then translate by comparison into the final decision.

4.3.2 Extended integrated implementation

The second heuristic employs a similar idea, but instead of computing copy costs naively on-the-fly, it derives cluster assignment decisions from annotated DG edges. A simple annotation example is given in Figure 7. In case a data dependency exists between a node and its predecessor, a *copy cost* is generated that equals 0 if the result of the predecessor can be supplied via crosspaths, and 1 otherwise. Therefore,

[1]The code shown is not in the most compact form. Since register allocation has not been performed yet, register copies can be seen which are eventually eliminated by coalescing. Also, loop-invariant definitions are visible and only basic addressing modes are used for this example.

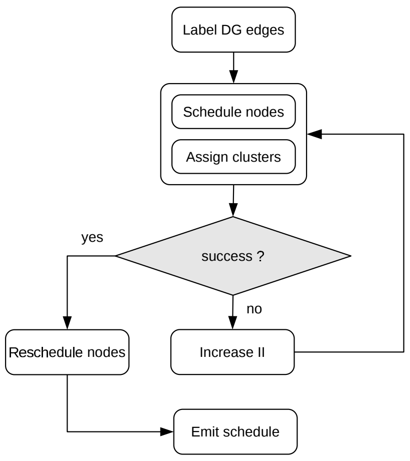

Figure 6: General organization of modulo scheduling and cluster assignment

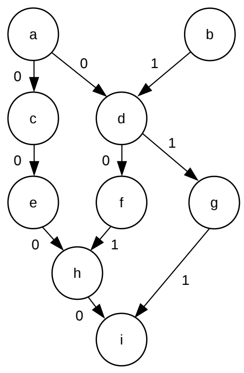

Figure 7: DG *copy cost* annotation example

a 0-edge means that adjacent nodes can be assigned to any cluster without involving copy overhead, while 1-edges enforce copy insertion.

While qualifying the data dependences between adjacent nodes in terms of register copies, the result of the annotation itself is cluster-independent, i.e. there is no assignment information available yet. This concrete information is generated during the scheduling step and - additionally to register transfers - takes cluster utilization into account.

For each node to be scheduled, data successors are considered that already have been scheduled. Given their number and weights of incident edges, a concrete assignment is generated for the node. So for example, if there is only one successor that is reachable via a 1-edge and scheduled on cluster A, a decision is made to assign the node to cluster A as well. In turn, in case no decision is possible based on the successor count, the heuristic maintains two utilization counters (*utilA*, *utilB*), one for each cluster, which are used for the favor of a balanced clustering and indeed account for a more even distribution, as demonstrated in Section 5.

After the cluster assignment information has been generated for a given node, a functional unit assignment takes place. If done successfully, the global clustering information map is updated, cluster utilization counters increased properly, and the heuristic continues with the next node in the ordering queue, as described in Section 4.2.

Algorithm 1 Labeling of DG edges

```
$edgeList \leftarrow$ empty list for annotated DG edges

**function** LABELDGEDGES
    $nodes \leftarrow$ list of DG nodes
    **for** each node $N$ contained in $nodes$ **do**
        $predList \leftarrow$ list of predecessors of $N$

        **for** each node $P$ contained in $predList$ **do**
            $copyCost \leftarrow 0$
            **if** no crosspath between $N$ and $P$ **then**
                $copyCost \leftarrow 1$
            **end if**

            $edgeList \leftarrow edgeList + (P, N, copyCost)$
        **end for**
    **end for**
**end function**
```

In integration with modulo scheduling, the presented transformation therefore heuristically approaches optimization of following performance influencing factors:

1. *initiation interval* for the shortest possible schedule through iterative modulo scheduling scheme;

2. reduction of *crosspath stalls* through explicit code rescheduling after preliminary schedule has been cluster assigned, and intercluster transfers committed;

3. number of *intercluster copies* through labeling of DG edges, hereby taking target properties, such as availability of crosspath support and functional unit assignment for a given instruction, into consideration;

4. even *cluster balance* through simple counters capturing utilization during cluster assignment;

A general collaboration of cluster assignment and modulo scheduling steps is shown in Figure 6. Algorithm 1 depicts the annotation procedure, Algorithm 2 additionally presents the cluster assignment routine in pseudo-code.

5. EXPERIMENTAL RESULTS

Our performance evaluation is based on a collection of 35 loops as found in benchmark suites, such as *DSPStone (DSP)*, *MiBench (MiB)*, *mediabench (MB)*, and *Benchmark-Game (BG)*, popular for general DSP compiler development and suitable for testing loop related optimizations. Additionally, 5 calculation intensive single unit tests (*SU*) have been added to the collection in order to extend the scope and stress both optimizers. All tests have been run on TI's software simulator configured for cycle accuracy.

We first intended to compare the efficiency of our implementations to the TI compiler shipped as part of the *TI Code Composer Studio*. Unfortunately, due to the experimental state of our entire backend, no fair and undistorted comparison is possible that allows rational conclusions. One of the reasons is the instruction selection phase, which lacks capability of complex pattern matching and currently only exploits basic addressing modes. Additionally, no advanced loop analysis and optimization passes for the machine code

Algorithm 2 Cluster assignment heuristic

```
Require: DG node $N$
Require: $edgeList$: list of labeled DG edges
Require: $utilA$, $utilB$: current cluster utilization
Require: $assignments$: list of already assigned nodes

  function GETCLUSTERASSIGNMENT($N$, $utilA$, $utilB$)
      $succList \leftarrow$ list of successors of $N$
      $numSuccsA \leftarrow 0$
      $numSuccsB \leftarrow 0$

      for each node $S$ contained in $succList$ do
          if $S$ is in $assignments$ then
              $succCluster \leftarrow$ cluster assigned to $S$
              $edgeWeight \leftarrow$ weight of edge ($N$, $S$)

              if $edgeWeight$ is greater than 0 then
                  if $succCluster$ equals $CLUSTER\_A$ then
                      $numSuccsA \leftarrow numSuccsA + 1$
                  else
                      $numSuccsB \leftarrow numSuccsB + 1$
                  end if
              end if
          end if
      end for

      $cluster \leftarrow CLUSTER\_A$
      if $numSuccsB$ greater than $numSuccsA$ then
          $cluster \leftarrow CLUSTER\_B$
      else
          if $numSuccsA$ equals $numSuccsB$ then
              if $utilA$ greater than $utilB$ then
                  $cluster \leftarrow CLUSTER\_B$
              end if
          end if
      end if

      $assignments \leftarrow assignments + (N, cluster)$
      if $cluster$ equals $CLUSTER\_A$ then
          $utilA \leftarrow utilA + 1$
      else
          $utilB \leftarrow utilB + 1$
      end if
  end function
```

are available within LLVM, resp. currently implemented within our backend. This leads to the fact that our compiler, compared to TI's tool, produces a different code, with differences (in terms of size and content) being significant enough to prevent any serious comparison. Instead, we first compare our solutions against each other taking a UAS implementation as reference. This baseline implementation is cluster-aware, it heuristically performs scheduling and clustering, but does so in a non-modulo fashion. Since comparing linear vs. modulo schedules is not entirely fair, we additionally present a comparison to a fully integrated ILP solution that is developed in our group, optimizes for the same parameters as listed at the end of the Section 4, and additionally incorporates register requirements into the model. Described heuristics as well as the ILP reference employ modulo variable expansion and therefore basically utilize loop unrolling by replicating kernel blocks. The amount of unrolling is not controlled directly but determined implicitly during processing as a result of register pressure estimation.

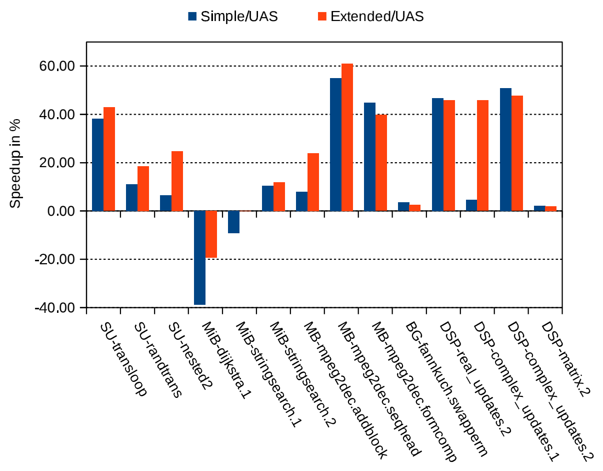

Figure 8: Cycle speedup (%) comparison to UAS for a selection of tested loops

Figure 8 compares described heuristics to UAS and Figure 9 lists detailed statistics. While presented results match the expectation of both solutions being more potent than UAS in terms of parallelization and hence runtime, some cases demand further explanation.

benchmark		extended			simple		
name	size	cycles	II	copies	cycles	II	copies
MiB-dijkstra.1	16	1274	5	0	1484	8	4
MiB-stringsearch.1	13	5439	4	0	5943	6	2
MiB-stringsearch.2	17	3989	6	1	4052	6	3
SU-transloop	29	1665	10	0	1800	11	4
SU-randtrans	25	14428	9	3	15740	11	5
SU-nested2	23	12442	7	1	15450	10	4
MB-mpeg2dec.addblock	24	181	7	2	219	11	3
MB-mpeg2dec.seqhead	23	835	9	2	965	11	7
MB-mpeg2dec.formcomp	24	823	9	2	753	8	4
BG-fannkuch.1	16	3946	7	4	3913	5	2
DSP-real_updates.2	15	380	5	1	373	6	1
DSP-complex_updates.1	28	504	8	1	612	12	10
DSP-complex_updates.2	47	1179	24	6	1108	25	6
DSP-matrix.2	16	89386	6	2	89286	6	2

Figure 9: Detailed statistic for a selection of tested loops

For *MiB-dijkstra.1* a significant runtime degradation is observed for both implementations (19.29% resp. 38.95%). Here, the extended candidate achieves a schedule length that is, compared to UAS, shorter by 33% (9 vs. 6 instructions). Nevertheless, without unrolling being performed, the length benefit of 3 cycles is not sufficient to equalize the latency of one additional branch in the latch block (6 cycles). For *MiB-stringsearch.1* a low trip count (8 iterations) together with the loop size (13 instructions) is recorded. This makes a pipeline representation not profitable, the runtime overhead induced by prologues and epilogues together with the latch block terminating the kernel outweigh the advantage of increased parallelization. Further, for machines with high branch latencies such small loops are likely to be compressed entirely into the branch delay slots, again minimizing performance gain, or even causing a performance drop. While the extended heuristic still manages to improve marginally (0.09%), the simple clustering candidate suffers a degradation by 9.17%. For loops bigger in size, such a behavior is however not observed, most of the depicted loops clearly benefit from proposed optimizations. The simple heuristic solution scores 16.71% on average, the improved heuristic achieves a speedup of 24.82% and is therefore clearly more profitable.

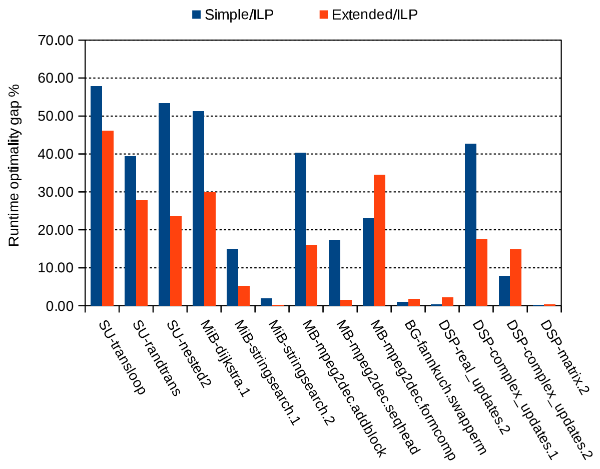

Figure 10: Cycle optimality gap (%), compared to the integrated ILP solution

A comparison to an integrated ILP model for scheduling and clustering is found in Figure 10. The used ILP reference creates optimal solutions for multiple performance factors (initiation interval, copies, crosspath stalls, register pressure), thus, the presented data can be seen as overall optimality gap with respect to those factors. The extended heuristic performs significantly better, resulting in the average value of 15.81%.

Figure 11 shows a comparison of achieved initiation intervals for presented kernels. Here too, the naive implementation results in longer schedules, leading to the extended model to be consistently rendered as a winner, resulting in an average of 8.29 (vs. 9.71).

Figure 12 finally addresses balancing capabilities of proposed heuristics. The displayed data corresponds to the percentage of instructions assigned to cluster A. For example, taking the naive implementation, for *MB-mpeg2dec.addblock* 70% of all instructions are scheduled on A and 30% on B. Generally, the extended clustering heuristic distributes more evenly, which is not entirely surprising due to the distribution counters incorporated into the clustering mechanism.

6. CONCLUSION

In this paper we have addressed generation of cluster assigned modulo schedules for the popular TMS320C64X DSP family by Texas Instruments. First, adaptations and extensions of the swing modulo scheduling scheme are described in order to address major specialties of the TMS320C64X

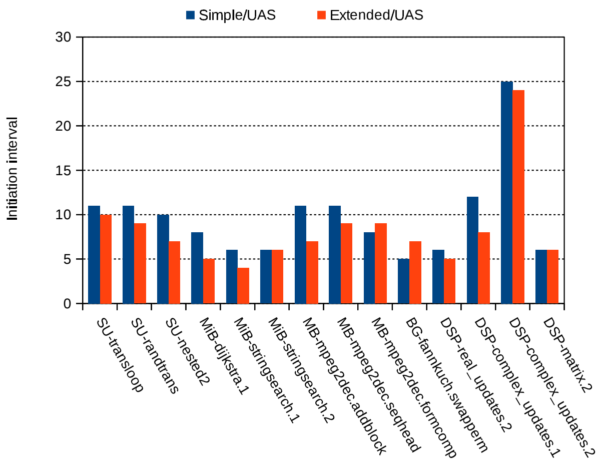

Figure 11: Comparison of absolute initiation interval values

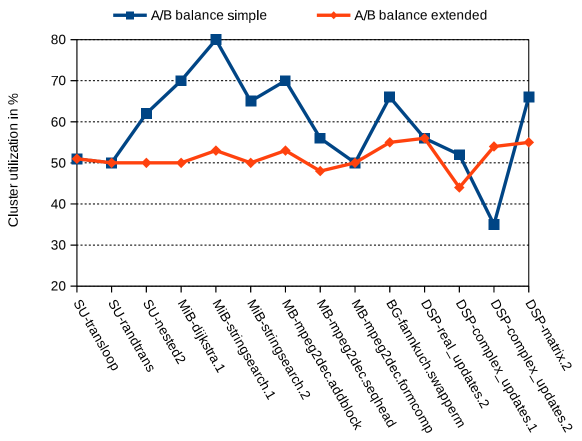

Figure 12: Cluster (A to B) balance (%) graph

architecture. Then, a simple greedy clustering heuristic is presented that is combined with the swing scheduler and is especially suitable for a fast schedule generation. As alternative, an extended clustering is provided which achieves significantly better results, with respect to both, dynamic cycle performance, as well as static parameters, such as copies and initiation intervals.

Compared to UAS and independent of the cluster assignment heuristic in use, a substantial performance gain is achieved for generated schedules for most of the tested kernels. However, when considering numbers obtained from the optimal integrated ILP solution, there is still enough space for further improvements which we intend to do in future.

7. FUTURE WORK

One of the major issues we want to address in near future is a tighter collaboration of the modulo scheduler with the register allocation process. Both presented implementations realize a register pressure estimation scheme which conservatively rejects scheduling attempts for a given *II* in case register demands are estimated to be too high. While functional and robust, a proper integration of spill code may be more profitable in some particular cases.

Fully integrated integer linear programming approaches are not attractive for the daily routine due to temporal requirements that usually restrict effective application to tiny loops only. However, hybrid approaches that model selected parts of a global problem as ILP can be a promising compromise, pairing good quality code with low compile time demands. Thus, additionally to purely heuristic integrated solutions, we also intend to develop hybrid solutions that approach optimality (in terms of code) within a compile time span that qualifies them to be suitable for an industrial use.

8. REFERENCES

[1] V. H. Allan, R. B. Jones, R. M. Lee, and S. J. Allan. Software pipelining. *ACM Computing Surveys*, 27(3), Sept. 1995.

[2] E. R. Altman and G. R. Gao. Optimal modulo scheduling through enumeration. *Int. J. Parallel Program.*, 26(3):313–344, June 1998.

[3] C. Bruel. If-conversion ssa framework for partially predicated vliw architectures. In *ODES Workshop on Optimizations for DSP and Embedded Systems*, pages 5–13, 2006.

[4] B. Dupont de Dinechin. A unified software pipeline construction scheme for modulo scheduled loops. *Lecture Notes in Computer Science*, 1277:189–200, 1997.

[5] A. E. Eichenberger and E. S. Davidson. Stage scheduling: a technique to reduce the register requirements of a modulo schedule. In *Proceedings of the 28th annual international symposium on Microarchitecture*, MICRO 28, pages 338–349, Los Alamitos, CA, USA, 1995. IEEE Computer Society Press.

[6] R. Ellis. *Bulldog: A compiler for VLIW architectures.* The MIT Press, 1986.

[7] M. Eriksson and C. Kessler. Integrated code generation for loops. *ACM Trans. Embed. Comput. Syst.*, 11S(1):19:1–19:24, June 2012.

[8] A. Ertl and A. Krall. Optimal instruction scheduling using constraint logic programming. *Programming Language Implementation and Logic Programming (PLILP)*, 1991.

[9] J. Fang. Compiler algorithms on if-conversion, speculative predicates assignment and predicated code optimizations. In *LCPC '96 Proceedings of the 9th International Workshop on Languages and Compilers for Parallel Computing*, 1996.

[10] K. I. Farkas, N. P. Jouppi, and P. Chow. Register file design considerations in dynamically scheduled processors. In *In Proceedings of the Second IEEE Symposium on High-Performance Computer Architecture*, pages 40–51, 1995.

[11] M. M. Fernandes, J. Llosa, and N. Topham. Distributed modulo scheduling. pages 130–134, 1999.

[12] R. A. Huff. Lifetime-sensitive modulo scheduling. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, PLDI '93, pages 258–267, New York, NY, USA, 1993. ACM.

[13] M. S. Lam. Software pipelining: an effective scheduling technique for vliw machines. *SIGPLAN Not.*, 39(4):244–256, Apr. 2004.

[14] R. Leupers. Code generation for embedded processors. In *Proceedings of the 13th international symposium on System synthesis*, ISSS '00, pages 173–178, Washington, DC, USA, 2000. IEEE Computer Society.

[15] J. Llosa, E. Ayguadeá, A. González, M. Valero, and J. Eckhardt. Lifetime-sensitive modulo scheduling in a production environment. *IEEE Transactions on Computers*, 50:234–249, 2001.

[16] E. Nystrom and A. E. Eichenberger. Effective cluster assignment for modulo scheduling. In *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, MICRO 31, pages 103–114, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[17] E. Özer, S. Banerjia, and T. M. Conte. Unified assign and schedule: A new approach to scheduling for clustered register file microarchitectures. In *In International Symposium on Microarchitecture*, pages 308–315, 1998.

[18] S. Park. On predicated execution. Technical report, Tech. report, HP laboratories, 1991.

[19] B. R. Rau. Iterative modulo scheduling: an algorithm for software pipelining loops. In *Proceedings of the 27th annual international symposium on Microarchitecture*, MICRO 27, pages 63–74, New York, NY, USA, 1994. ACM.

[20] J. Sánchez and A. González. Instruction scheduling for clustered vliw architectures. In *Proceedings of the 13th international symposium on System synthesis*, ISSS '00, pages 41–46, Washington, DC, USA, 2000. IEEE Computer Society.

[21] G. Shobaki and K. Wilken. Optimal superblock scheduling using enumeration. In *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, pages 283–293, Washington, DC, USA, 2004. IEEE Computer Society.

[22] G. Shobaki, K. Wilken, and M. Heffernan. Optimal trace scheduling using enumeration. *ACM Trans. Archit. Code Optim.*, 5(4):19:1–19:32, Mar. 2009.

[23] E. Stotzer and E. Leiss. Modulo scheduling for the tms320c6x vliw dsp architecture. In *Proceedings of the ACM SIGPLAN 1999 workshop on Languages, compilers, and tools for embedded systems*, LCTES '99, pages 28–34, New York, NY, USA, 1999. ACM.

[24] N. J. Warter, G. E. Haab, K. Subramanian, and J. W. Bockhaus. Enhanced modulo scheduling for loops with conditional branches. In *Proceedings of the 25th annual international symposium on Microarchitecture*, MICRO 25, pages 170–179, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[25] K. Wilken, J. Liu, and M. Heffernan. Optimal instruction scheduling using integer programming. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, pages 121–133, New York, NY, USA, 2000. ACM.

[26] www.llvm.org.

[27] www.ti.com.