Instruction Scheduling for Clustered VLIW DSPs

Rainer Leupers*

University of Dortmund
Department of Computer Science 12
Embedded Systems Group
44221 Dortmund, Germany
email: leupers@LS12.cs.uni-dortmund.de

Abstract

Recent digital signal processors (DSPs) show a homogeneous VLIW-like data path architecture, which allows C compilers to generate efficient code. However, still some special restrictions have to be obeyed in code generation for VLIW DSPs. In order to reduce the number of register file ports needed to provide data for multiple functional units working in parallel, the DSP data path may be clustered into several sub-paths, with very limited capabilities of exchanging values between the different clusters. An example is the well-known Texas Instruments C6201 DSP. For such an architecture, the tasks of scheduling and partitioning instructions between the clusters are highly interdependent. This paper presents a new instruction scheduling approach, which in contrast to earlier work, integrates partitioning and scheduling into a single technique, so as to achieve a high code quality. We show experimentally that the proposed technique is capable of generating more efficient code than a commercial code generator for the TI C6201[1].

1. Introduction

Software development for embedded DSP systems is frequently a bottleneck in the design process, due to the lack of powerful development tools. On the other hand, efficient software is becoming more and more important in embedded system design. For example, Siemens recently announced [1] that the stand-by power consumption of their C25 mobile phone has been reduced by 60 % through a pure software modification. In particular, programming support for DSPs by C compilers is known to be very poor in terms of code quality [2, 3, 4]. The reason is that traditional fixed-point DSPs show an irregular, domain-specific architecture, to which C programs can hardly be mapped efficiently.

However, this situation has changed as new families of high-performance DSPs, such as the Texas Instruments C6201 [5], have become available. These recent DSPs tend to show a very long instruction word (VLIW) architecture. The TI C6201, for instance, has 8 parallel functional units (FUs), working independently of each other. Each FU is controlled via a separate 32-bit field in the VLIW instruction word. Additionally, all registers are general-purpose like in a RISC processor. Another example for VLIW DSPs is the Philips Trimedia architecture [6].

A main motivation for developing VLIW DSPs was the fact, that their rather regular architectures facilitate the construction of compilers capable of generating efficient code. This is extremely important for embedded systems which have to be area-efficient and have to meet real-time constraints. Several standard compiler techniques can be used, such as register allocation graph coloring [7], and software pipelining [8]. The most important code generation phase for VLIW DSPs is *instruction scheduling*, which performs the FU and control step binding of instructions, so as to achieve an optimum exploitation of potential parallelism.

Unfortunately, available VLIW DSPs with many parallel instruction slots are still different from the "ideal" VLIW model with full orthogonality[2] between registers and FUs. The limiting factor is the need to move up to two arguments from the register file (RF) to each FU in each instruction cycle, and to move back one result per FU, so that the RF would need to be equipped with a large number of read/write ports, which are expensive in terms of silicon area. The number of required RF ports can be reduced by *clustering* the data path. For instance, the TI C6201 data path is divided into two identical halves, called A and B. Both A and B have their local RF, and there is full orthog-

*This work has been supported by Agilent Technologies, USA.

[1] Publication: PACT 2000, Philadelphia, Oct 2000, ©IEEE

[2] Orthogonality means that each FU has random access to each register.

onality between the local RF and the FUs in the respective cluster. A more detailed description of this architecture will be given in section 3.

Instruction scheduling for processors with a clustered data path is more difficult than for an orthogonal VLIW architecture. In case of the TI C6201, for instance, the limited communication capabilities between clusters A and B have to be taken into account. Computations executed on A and B in general need to exchange values, but the transport of values between A and B must take place via a *restrictive interconnection network*, which allows the transfer of only a single value from A to B (and vice versa) within each instruction cycle. As we will discuss in section 2, existing VLIW scheduling techniques are not capable of directly incorporating the partitioning of instructions between the different clusters for such an architecture. As a result, there is a potential loss in code quality when using such scheduling techniques.

The contribution of this paper is an *integrated instruction scheduling and partitioning technique* specifically designed for clustered VLIW data paths. As a demonstrator we will use the TI C6201 VLIW DSP. The structure of the paper is as follows. After a discussion of related work in section 2, we describe the TI C6201 architecture in more detail in section 3. Section 4 defines the instruction scheduling problem, while sections 5 and 6 present our scheduling technique, which consists of two interleaved phases. In section 7, we experimentally show that this approach achieves higher code quality in terms of performance than the scheduling technique in the native TI C6201 code generator. Finally, conclusions are given.

2. Related work

A number of effective local and global scheduling algorithms are known for orthogonal VLIW machines. These include list scheduling, critical path scheduling [9], trace scheduling [10], and percolation scheduling [11]. However, these algorithms have not been designed for clustered data paths, so that they at least require a partitioning phase prior to scheduling in order to be applicable.

Nicolau et al. [12] have considered the problem of register assignment for VLIW architectures with multiple RFs. Their approach is based on a hypergraph coloring technique, where the goal is to find a register assignment that meets the constraints imposed by the number of physically available RF read/write ports. Partial instruction rescheduling is performed in case that constraints are violated. However, the underlying processor model is a fully orthogonal VLIW architecture (except for the RF port constraints), so that the potential communication bottleneck in clustered data paths is not considered.

Rau et al. [13] explicitly consider code generation for VLIW processors with clustered data paths (which they call EPIC architectures). They use a very fine-grained organization of code generation phases to obtain an efficient mapping of the source code to the given EPIC architecture. However, the partitioning of instructions is performed heuristically *before* instruction scheduling takes place, where the goal of the partitioning phase is to balance the FU load in the clusters. The partitioning phase inserts a number of *copy operations* into the code, that move values from one register file to another, so as to make values accessible for the FUs in a different cluster.

This approach results in a *phase ordering problem*. Partitioning is performed in advance without actually knowing the consequences on the resource conflicts and thus on the final schedule length. An unfavorable partitioning may impose unnecessary constraints on the subsequent scheduling phase, and may insert superfluous copy operations, eventually resulting in a suboptimal schedule. In fact, one can easily observe that the optimum partitioning can only be determined *at the time of scheduling*: Instructions might be well balanced between the clusters, but the need to copy values may induce additional instruction cycles. In turn, free instruction slots in these additional cycles could be used to execute useful instructions. Thus, the insertion of copy operations generally has a global impact on the schedule.

Also min-cut partitioning algorithms (e.g., [14]) are not useful in this situation, since there is no need to *minimize* the communication between the clusters. Instead, available communication resources and free instruction slots for copy operations should be fully exploited.

In [15], a theoretical analysis has been provided which allows to derive lower bounds on the minimum schedule length for a fixed binding of DFG nodes to clusters. This technique could be used in a branch-and-bound algorithm for simultaneous partitioning and scheduling. However, due to some simplifying assumptions in the underlying processor model it is not clear whether this is also possible for real-life processors with special architectural restrictions, such as the TI C6201.

The unified assign and schedule algorithm from [16] is conceptually close to the one presented in this paper, since it also relies on a list scheduler for performing integrated partitioning and scheduling. One main difference is that our approach includes a feedback path from the scheduler to the partitioner in order to revise unfavorable partitioning decisions. In addition, we evaluate our technique for a real-life VLIW processor instead of a hypothetical architecture. In [17], the work from [16] has been embedded into a software pipelining technique for clustered VLIWs. Likewise, the technique from [18] emphasizes software pipelining and is dedicated to a very special class of VLIWs. In contrast, this paper is focused on acyclic code segments.

Fisher et el. [19] proposed a heuristic algorithm called

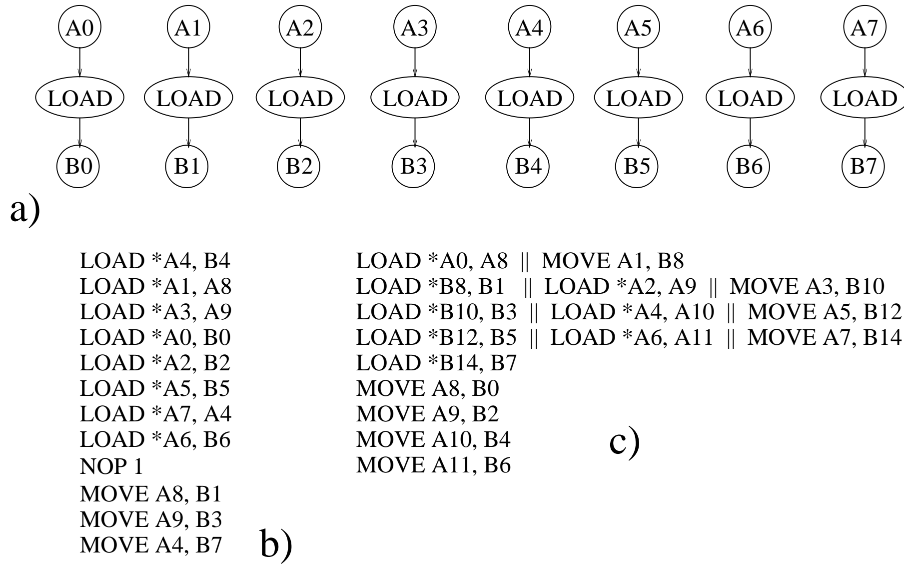

Figure 1. Schedule length minimization by insertion of copy operations: a) data flow graph, b) schedule generated by TI C compiler (12 cycles, 12 instruction words), c) performance-optimal schedule (9 cycles, 16 instruction words), "||" denotes parallel execution of instructions

Partial Component Clustering for the problem of partitioning DFG nodes between the clusters. The main idea is to assign subgraphs ("components") of the DFG to clusters in such a way, that copy operations along critical paths are avoided. The initial assignment is afterwards iteratively improved by swapping component elements, while estimating the resulting schedule length with a simplified list scheduler.

A problem with this approach is that driving the partitioning phase by critical paths is mainly useful for such DFGs, where the critical path length is close to the actual minimum schedule length. In this case, nodes not lying on a critical path most likely can be scheduled in free instruction slots along the critical path (which is also the main motivation of the local critical path scheduling technique [9]). However, if there is a "wide" DFG, then the critical path length is only a very loose lower bound on the minimum schedule length, because the FUs become the limiting factor in scheduling. This can be shown by a small example for the TI C6201.

In fig. 1 a), a simple-structured DFG is shown, containing 8 LOAD instructions. Each LOAD uses a register from cluster A as a pointer to load a value from memory into a register within cluster B. Since a LOAD has 4 delay slots and all LOADs are potentially parallel, the critical path length is 5. Fig. 1 b) shows the schedule generated by the TI C6201 *assembly optimizer*. This tool is part of the software development toolkit for the TI C6201 [20]. It reads symbolic sequential assembly code (generated manually or by the TI C compiler) and performs partitioning, scheduling and register allocation. According to the restrictions that will be described in section 3, two LOADs can only be scheduled in parallel, if the pointers are located in different RFs. Since all pointers are initially located in RF A, the TI assembly optimizer generates fully sequential code. The three MOVE instructions at the end could also be scheduled in parallel to earlier LOADs, but the schedule length of 12 would not be changed, since the 4 delay slots of the last LOAD instruction would then need to be filled with NOPs.

In contrast, fig. 1 shows a better schedule (in fact the schedule generated by the algorithm presented in this paper) with a length of only 9 cycles. In the first 4 cycles, pointers located in A registers with an odd index are copied into RF B. In cycles 2 to 4, this allows to schedule two LOADs in parallel each. In cycles 6 to 9, the loaded values still residing in RF A are finally moved to their required locations in RF B. As can be seen, we have traded a larger code size for a faster schedule. One can easily show that the schedule from fig. 1 c) is performance-optimal.

In our approach, we take the mutual dependence between instruction partitioning and scheduling into account by *phase coupling*. Both phases are executed simultaneously in an interleaved fashion, so that the partitioning of instructions already takes into account the resource conflicts and communication restrictions exposed during scheduling.

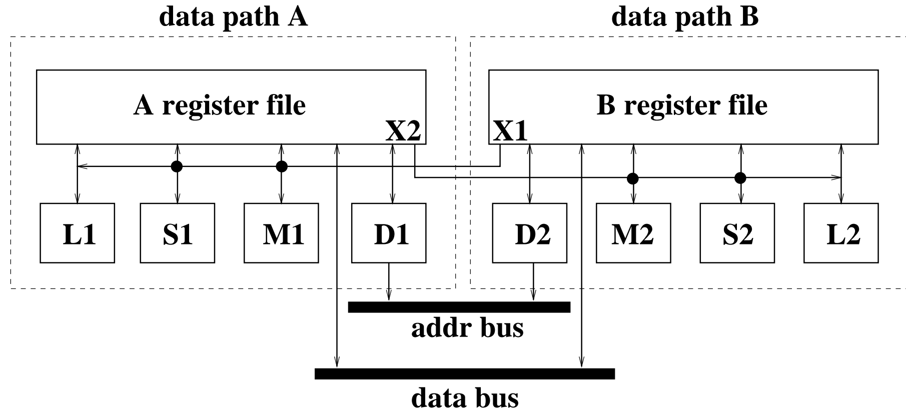

Figure 2. TI C6201 data path architecture

3. TI C6201 data path architecture

In order to illustrate the problem, this section outlines the data path architecture of a popular VLIW DSP, the TI C6201 (fig. 2). It shows a typical load-store architecture, where all computations take place on (general-purpose) registers. The data path consists of two symmetric clusters A and B. Each cluster has a local 16×16 bit RF and 4 FUs (called L, S, M, and D) working in parallel. Each FU type is capable of executing a certain subset of instructions. These subsets are partially overlapping, e.g., an ADD instruction may be executed on L, S, and D type FUs. Most instructions have a delay of one cycle (i.e., the result is available in the next cycle), but some instructions have a larger delay (e.g. 2 for a multiply, 4 for a load). The instruction pipeline is visible to the compiler (or programmer), so that NOPs have to be inserted in case the delay slots of an instruction cannot be filled with useful computations.

The FUs in both clusters primarily work on their local RF. The only exception is that, in each instruction cycle, at most one of the units L, S, and M may read at most one argument from the opposite RF. Such read operations take place via the *cross paths* X1 and X2. Since there is only one cross path per cluster, at most two values can flow between A and B in each instruction cycle. Such a transport may be a copy operation from one RF to the other (via a MOVE instruction), but the value read over the cross path may also be directly consumed by some FU in the same cycle in which it is transported. In the latter case the value does not get stored in the local RF of the FU for further uses.

Besides these general restrictions, there are some further important constraints to obey:

- Any copy operation blocks one FU for one cycle, since it is mapped to an addition of zero to the copied value. While D units may execute additions, they cannot receive arguments from the opposite RF. Thus, only L and S units can be used for copy operations.
- L units may receive either the left or the right argument from the opposite RF, while for S and M units the left argument *must* be read from the local RF.
- Memory addresses computed by D units may be used for a LOAD/STORE into/from the opposite cluster. Since there are two D units, the C6201 allows to issue up to two LOAD/STORE instructions in each cycle. However, in such a case, the memory addresses must be located in different RFs, and the same restriction holds for the values to be loaded or stored (e.g., two parallel loads into the RF of cluster A are invalid).

Due to these restrictions, the partitioning of instructions between A and B obviously has a large impact on the length of the schedule generated for a given code sequence. A pure FU load balancing between A and B is unlikely to lead to the optimum solution, because the constraints on the communication of values between A and B make it impossible to accurately predict the schedule length at an early point of time.

4. Problem definition

The scheduling problem we would like to solve can be stated as follows: Let B be a basic block, represented by an edge-weighted data flow graph $G = (V, E, w)$. We assume that code selection has already been performed, i.e., the DFG nodes in V represent concrete instructions, while DFG edges represent scheduling dependencies between instructions. Each edge e is weighted by an integer delay value $w(e)$.

We assume that the DFG nodes are not yet bound to one of the two clusters A and B. Thus, a *partitioning*

$$P : V \rightarrow \{A, B\}$$

of nodes between A and B must be computed. During scheduling it has to be decided, which FU a node is bound to, and at which point of time (or *control step*) its execution is started. For a given partitioning P, an *instruction schedule* is thus represented by two mappings

$$F : V \to \{L, S, M, D\}$$

$$C : V \to \mathbb{N}$$

We say that a schedule is *valid*, if for any node $v \in V$ the FU mapping is such that FU $F(v)$ belongs to cluster $P(v)$, $F(v)$ can implement the instruction represented by v, any FU is assigned at most one node per cycle, and the control step binding C does not violate any inter-instruction dependencies. The latter means for any node v with incoming edges

$$e_1 = (u_1, v), \ldots, e_k = (u_k, v)$$

the following constraint must hold:

$$C(v) \geq \max_{i=1}^{k}(C(u_i) + w(e_i))$$

The length $L(S)$ of a schedule $S = (F, C)$ is defined as the latest control step in which an instruction v, having a delay of $d(v)$, finishes its execution:

$$L(S) = \max_{v \in V} \quad (C(v) + d(v))$$

Our goal is to *simultaneously* compute a partitioning P and a valid schedule (F, C) of minimum length. However, since resource-constrained scheduling is NP-hard even for a fixed partitioning [21], in practice we have to resort to a technique that generally produces only "close-to-optimal" solutions.

Whenever there is a data dependence between two instructions assigned to different clusters, then the schedule must also comprise either a copy operation or a direct transfer via a cross path. Which alternative is better depends on the resources currently available and is thus determined dynamically in our scheduling approach.

5. Partitioning algorithm

The proposed scheduling technique consists of two interleaved phases. In phase 1, tentative instruction partitioning is performed. Then, for the given partitioning, a schedule is computed in phase 2. The cost of the schedule (the number of instructions cycles needed to execute it) is used to measure the quality of the partitioning. Then, based on this feedback, phase 1 tries to find an improved partitioning, for which phase 2 is invoked again, and so forth. This process is iterated, until a certain termination criterion is met.

```
algorithm PARTITION
input: DFG G with N nodes;
output: P: array[1..N] of {0, 1} ;
var
  int i,r,cost,mincost;
  float T;
begin
  T = 10;
  P := RANDOMPARTITIONING();
  mincost := LISTSCHEDULE(G,P);
  while T > 0.01 do
    for i = 1 to 50 do
      r := RANDOM(1,n);
      P[r] := 1 - P[r];
      cost := LISTSCHEDULE(G,P);
      delta := cost - mincost;
      if delta < 0 or RANDOM(0,1) < exp(-delta/T)
      then mincost := cost;
      else P[r] := 1 - P[r];
      end if
    end for
    T = 0.9 * T;
  end while
  return P;
end algorithm
```

Figure 3. Partitioning algorithm

For partitioning in phase 1, we use a *simulated annealing* (SA) algorithm [22]. Similar to genetic algorithms [23], SA is suitable for nonlinear optimization problems, since it is capable of escaping from local optima in the objective function. The basic idea is to simulate a *cooling process*. Starting with an initial *temperature* and an initial solution, in each step the current solution is randomly modified. If the new solution is better, then it is accepted as the new current solution. Otherwise, it depends on the cost difference to the previous solution and the current temperature whether the new solution is accepted. During the annealing process, the temperature is lowered step by step, and the probability of accepting worse solutions decreases. Our concrete SA algorithm is shown in fig. 3.

Initially, a random[3] partitioning P is used. Then, the input DFG is scheduled by function LISTSCHEDULE, which implements phase 2 of our approach. In each iteration of the SA algorithm, the current partitioning is modified by inverting the cluster flag (0 denotes cluster A, 1 denotes cluster B) for one randomly selected instruction. The quality of the new partitioning is evaluated again by a call to

[3] During experimentation we observed that using a heuristic seed for SA in this case does not produce better solutions.

LISTSCHEDULE. If the new partitioning results in a shorter schedule, then it is accepted as the new optimum. Also worse solutions may be accepted, so that the SA algorithm generally does not get trapped in a local optimum. In case the new partitioning is not accepted, the previous one is restored by re-inverting the cluster flag. This process is iterated, until the "temperature" (parameter T) is "frozen". Finally, the resulting partitioning is emitted, and a last run of LISTSCHEDULE can then be used to compute the final schedule.

Note that, although possible, it would not be a good approach to solve the entire problem with the SA algorithm, since the search space would get extremely large if we also integrated the computation of the FU and control step mappings F and T into the SA. Instead, for phase 2, we use a fast list scheduling algorithm, which aims at constructing the best schedule for a *given* partitioning by using a number of heuristics. This scheduling algorithm is presented in the following section.

6. Scheduling algorithm

The scheduling main routine is a conventional list scheduling algorithm [9] which, in our case, besides the input DFG G additionally takes a given partitioning P as an input (fig. 4). While there are unscheduled nodes left, the next node to be scheduled is picked by subroutine NEXTREADYNODE, which returns a node whose DFG predecessors have already been scheduled. In case of alternative ready nodes, a node with a minimum ALAP (as late as possible) time is heuristically selected. Each selected node is placed into the schedule by function SCHEDULENODE, which forms the core of the scheduling algorithm. Finally, the length of the schedule (in instruction cycles) is returned.

Subroutine SCHEDULENODE shown in fig. 5 uses a number of heuristics to avoid additional instruction cycles caused by the need to communicate values between clusters A and B. Its inputs are the current schedule S, the node m to be inserted into S, and the current partitioning P. The main strategy is to insert m into the earliest possible control step without violating resource and dependency constraints. Initially, this control step is given by the ASAP (as soon as possible) time of m. However, if some predecessor of m has been scheduled in control step t, and the delay of the corresponding instruction is d, then m cannot be scheduled earlier than at time $t + d$. These tests are performed in subroutine EARLIESTCONTROLSTEP.

The control step number cs into which m will be placed is iteratively incremented until a valid control step without resource conflicts has been found.

For a given value of cs, subroutine GETNODEUNIT searches for a free FU f_m, capable of executing the instruction represented by m in step cs on the cluster defined by

```
algorithm ListSchedule
input: DFG G, partitioning P;
output: schedule length;
var m: DFG node;
  S: schedule;
begin
  mark all nodes as unscheduled;
  S := ∅;
  while (not all nodes scheduled) do
    m := NextReadyNode(G);
    S := ScheduleNode(S, m, P);
    mark node m as scheduled;
  end while
  return Length(S);
end algorithm
```

Figure 4. Main scheduling algorithm

```
algorithm ScheduleNode
input: current schedule S, node m, partitioning P;
output: updated schedule S containing node m;
var cs: control step number;
begin
  cs := EarliestControlStep(m) - 1;
  repeat
    cs := cs + 1;
    f_m := GetNodeUnit(m, cs, P);
    if f_m = ∅ then continue; /* try next cs */
    if (m has an argument on a different cluster) then
      CheckArgTransfer();
      if (at least one transfer impossible) then continue;
      else TryScheduleTransfers();
  until (m has been scheduled);
  if (m is a LOAD instruction) then
    DetermineLoadPath(m);
  end if
  if (m is a CSE with more than 2 uses) then
    InsertForwardCopy(S, m);
  end if
  return S;
end algorithm
```

Figure 5. Scheduling algorithm for a single node

$P(m)$. In case of multiple free FUs the selection is made arbitrarily. Note that this selection may still be revised later during *version shuffling*: If no free FU is directly found, then version shuffling [9] is applied to the current control step cs. Version shuffling tries to rearrange the FU binding of instructions already scheduled in cs, such that one FU capable of executing m gets free. If version shuffling fails to free a resource, then cs is incremented, and resource allocation is repeated.

Even if a free resource f_m has been found, scheduling m at time cs might still fail due to the need to provide m's arguments to f_m. If m is assigned to the same cluster as its arguments, then no further actions are required due to the orthogonality of FUs and the RF in each cluster. If, however, an argument resides in the opposite RF, then its transfer between the clusters needs to be scheduled as well. There are two possibilities for this[4]:

1. The transfer takes place in control step cs via a cross path.
2. The transfer takes place via a copy operation scheduled earlier than cs.

The best alternative is determined heuristically. Without loss of generality (since A and B are symmetric), let m be assigned to cluster A, let x be the left argument of m computed on cluster B in control step $t(x) < cs$, and let y be the right argument of m computed on cluster B in control step $t(y) < cs$. The cases that m has less than two arguments or the arguments already reside in the RF of $P(m)$ are simple special cases.

Subroutine CHECKARGTRANSFER checks three possibilities of transporting each argument $a \in \{x, y\}$ from A to B.

1. If a is a *common subexpression* (CSE) in the DFG, then it might be the case, that a copy operation from B to A had already been scheduled for another use of a by an instruction that was scheduled earlier. If that copy operation does exist and happens to be scheduled in a control step in the interval $[t(a)+1, cs-1]$ then it can be *reused*.
2. It is checked, whether a new copy operation from B to A could be inserted in a control step $s \in [t(a)+1, cs-1]$. This is possible if both a resource for a copy operation (either L1 or S1) and cross path X1 are free in s.
3. It is checked, whether the transfer from B to A could take place in cs via cross path X1. This is the case, if X1 is not yet blocked in cs and if the FU f_m selected for m allows to read the argument via X1, dependent on its position (left or right).

[4] Theoretically, there is also a third possibility: copying a value via the memory. However, as both LOADs and STOREs have a significant delay, it is very unlikely that a benefit will result. Therefore, we neglect this option.

If none of the three cases holds for either x or y, then the arguments cannot be provided to m in time, and the process is repeated with an incremented control step number. Otherwise, the (possibly alternative) transfer possibilities are passed to subroutine TRYSCHEDULETRANSFERS, which tries to organize the transfer of x and y with a minimum amount of resource blocking, such that m can be scheduled in cs. Due to the limited space, we only summarize the most important concepts of this subroutine:

- Generally, whenever possible, priority is given to the reuse of copy operations, since a copy reuse does not block any further resources. The probability of reusing copies is increased by another heuristic (*forward copy insertion*) described below.
- If copies cannot be reused, priority is given to using the cross path X1 in control step cs rather than inserting a new copy in a step $t < cs$. The reason is that the latter possibility will not only block X1 in t, but also an FU, which might later be better used for another instruction.
- A very important technique is the exploitation of *commutativity* of operations (add, multiply, or, ...) in certain situations. If cross path X1 is free in cs, but the left argument x cannot be read via X1 (since f_m is not an L unit, cf. section 3), then swapping the arguments can still enable to schedule the transfers without inserting an additional control step. Likewise, if f_m is an L unit and X1 would be selected for transporting x, then (for commutative operations) the arguments are heuristically swapped. The reason is that implementing the transfer of x via X1 would prevent to revise the FU binding of m later during version shuffling. Instead, reading x as the right argument (which is not restricted to L units only) allows to later reassign m to another FU whenever L needs to be freed for another instruction.

Note that TRYSCHEDULETRANSFERS might still fail, even though each single argument could be transported: For instance, this is the case, if the transfer of both x and y could only take place via cross path X1 in the current control step, or if both x and y need to be copied, but there are insufficient resources. In this case, cs needs to be incremented, and the process is repeated.

Otherwise, node m is assigned to control step cs, and all required resources are marked as being blocked. Finally, two further heuristics are applied, from which the scheduling of subsequent nodes in general benefits:

DETERMINELOADPATH: If m is a LOAD instruction, then the RF which the loaded value will be written to is not fixed by the partitioning P. This is due to the fact, that memory addresses computed in one cluster can be used for loading a value into the RF of the opposite one. Only the cluster for computing the memory address itself is prescribed by P. As mentioned in section 3, two memory accesses can only be issued in parallel, if they load to (or store from) different RFs. We model this restriction by two virtual resources M_A and M_B. Whenever both are still free in the control step selected for m, this freedom can be exploited: The result of m is written to the RF of that cluster, to which the majority of uses of m are assigned by P. In case there is no such majority, the choice is made arbitrarily.

INSERTFORWARDCOPY: If m is a common subexpression (scheduled in cs) with more than two uses, m is not on a critical path in the DFG, and the majority of instructions using m as an argument are assigned to the opposite cluster, then a copy to that cluster is inserted in the earliest possible control step after cs. This heuristic enables the reuse of copy operations for the majority of uses of m.

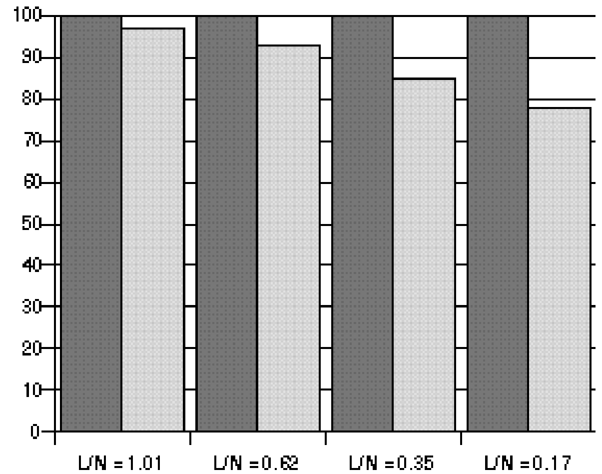

Figure 6. Relative length of generated schedules

7. Experimental results

In this section, we experimentally evaluate our technique by comparing the performance of generated schedules with schedules generated by the TI C6201 assembly optimizer, that has already been mentioned in section 2. We present results of a statistical analsis, followed by results for a set of real DSP code examples.

7.1. Statistical evaluation

For sake of a broad evaluation, we have first performed a statistical analysis based on four sets of 100 randomly generated DFGs each. An experimental evaluation using random inputs bears the disadvantage that we do not get results for "real" problems. However, we used this method, because showing that the technique produces good results on the average indicates that it will generally also achieve good results for "real" problem instances. In addition, using a sufficiently large input data base ensures the reproducibility of results, even without having access to the detailed benchmarks.

Since our approach does not capture register allocation effects, all DFGs have been generated in such a way, that no extra instructions due to register spilling were required. Since the TI C6201 has a total of 32 general-purpose registers, this is not a severe restriction, but spilling is only rarely required also in realistic code examples. The DFG sets are parameterized by the degree of potential *instruction-level parallelism* (ILP). This ILP degree is inversely related to the L/N ratio, where L is the critical path length, and N is the number of DFG nodes.

If L is close to N, then the concrete partitioning algorithm plays no substantial role for the result quality. The reason is that L is a lower bound on the schedule length, and that due to the large number of FUs available, nodes not lying on the critical path can most likely be scheduled in free instruction slots "along" the critical path. This means that the resulting schedule length is mostly identical or only slightly larger than L.

However, this situation is different in case of a low L/N ratio. In this case the number of nodes is much larger than the theoretical minimum schedule length, and the available resources become the limiting factor. Therefore, careful partitioning and scheduling become extremely important.

In our experimentation, each DFG has been scheduled by the technique described in this paper. Additionally, a sequentialized version of the same DFG has been scheduled by the TI assembly optimizer. Finally, a custom analysis tool was used to determine the code size and the performance of both schedules. The performance results are shown in fig. 6.

For each L/N ratio, the left bar shows the relative number of instruction cycles (average over 100 DFGs) of schedules generated by the TI assembly optimizer (set to 100 %), while the right bar shows the results generated by our approach. For the ratio $L/N = 1.01$, the results do not differ significantly, since due to the reasons explained above both schedulers were able to achieve the theoretical limit L in most cases. However, as can be seen, the difference grows with a decreasing L/N ratio. For highly parallel DFGs

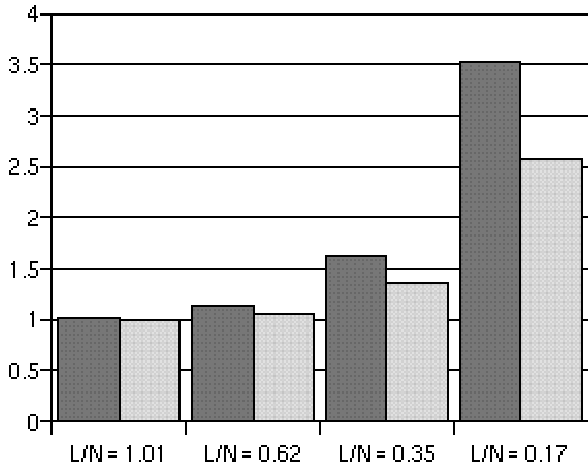

Figure 7. Performance compared to lower bound L

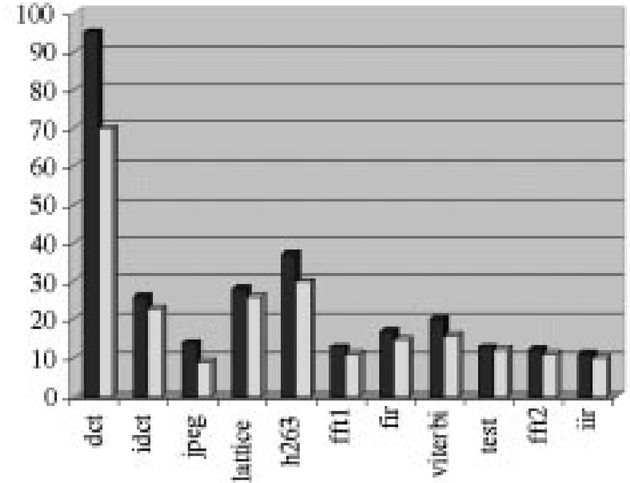

Figure 8. Performance results for real DSP code

($L/N = 0.17$), which for instance typically occur in unrolled loops, schedules generated by our approach on the average need only 78 % of the instruction cycles compared to schedules generated by the TI assembly optimizer. This large improvement is due to the better partitioning and utilization of available communication resources between clusters A and B.

Fig. 7 shows the performance results from a different perspective. For each L/N ratio, the average schedule length is compared to the critical path length (set to 1) in the DFG. For $L/N = 1.01$, both the TI assembly optimizer and our technique were able to achieve the theoretical limit in most cases. Again, the difference grows with decreasing L/N ratio. For $L/N = 0.17$, the TI assembly optimizer generates schedules of an average length of $3.5 \cdot L$, while our technique achieves $2.5 \cdot L$.

Since our approach tends to make more intensive use of copy operations, this performance improvement has to be paid with increased *code size*. The average overhead in code size as compared to the TI assembly optimizer ranged between 5 and 10 %.

7.2. Performance for real benchmarks

Fig. 8 shows performance results for a set of basic blocks extracted from realistic DSP programs. These are relatively small, compute-intensive kernels with a DFG size between 18 (iir) and 300 (dct) nodes. For compilation of the C source code, we have used our compiler platform LANCE [24]. The left bars show the number of instruction cycles of machine code generated by the TI assembly optimizer, while the right bars show the corresponding results for our integrated scheduling technique. The benchmarks are ordered by increasing L/N ratio, ranging from 0.11 (left) to 0.61 (right). As predicted by the results of the above statistical evaluation, the performance gain tends to fall with increasing L/N ratio. The performance improvements compared to the TI scheduler range between 7 % (iir) and 26 % (dct). Thus, the statistical evaluation corresponds well with results obtained for realistic applications.

Finally, we need to mention the runtime requirements of our scheduling technique. The TI assembly optimizer apparently uses a purely heuristic partitioning and scheduling approach, and is therefore comparatively fast. On the other hand, the simulated annealing (SA) technique (section 5) is generally known to be runtime intensive for large optimization problems. However, in our approach we have limited the use of SA to the partitioning task only, while the detailed scheduling is performed by an efficient heuristic. This "hybrid" approach allows us to schedule even large DFGs within reasonable time. For DFGs with approximately 100 nodes, the runtime for partitioning and scheduling on a Sun Ultra-1 workstation is typically in the order of 10 CPU seconds. In the area of embedded systems, where code quality is of much higher concern than compilation speed, this runtime is definitely acceptable.

8. Conclusions

VLIW DSPs are finding increasing use in the design of embedded systems. Compiler support for such DSPs is very important, since assembly-level programming of VLIW DSPs is an extremely time-consuming task. In this

paper we have presented a dedicated instruction scheduling technique for VLIW DSPs that show a clustered data path. For such architectures, the phases of scheduling and partitioning instructions between the clusters are highly interdependent. We have proposed a technique that tightly couples these two phases in order to achieve high code performance, and we have given experimental evidence that this technique generates faster code than a commercial code generator in case of the TI C6201 DSP. In order to measure the code quality improvements for an existing machine, the list scheduler has been developed specifically for this CPU with its special architectural restrictions. Porting the technique to other machines certainly requires to redesign the list scheduler. However, the approach in general is machine-independent, and our goal has been to demonstrate the optimization potential of phase-coupled partitioning and scheduling. In fact, the TI C6201 is a very interesting example for this purpose, because its capability of using cross paths for "volatile" copy operations adds another dimension to the search space.

Future work will deal with the adaptation of the scheduling algorithm to further processor architectures and the integration of register allocation, as well as integration with global scheduling techniques. Possible improvements of the presented technique include adaptations of the simulated annealing algorithm towards the concrete input DFGs, instead of using a fixed cooling schedule. In addition, the scalability of the proposed algorithm w.r.t. the number of functional units and inter-cluster communication resources should be investigated.

References

[1] Siemens: www.siemens.de/ic/products/cd/english/index, 1999

[2] V. Zivojnovic, J.M. Velarde, C. Schläger, H. Meyr: *DSP-Stone – A DSP-oriented Benchmarking Methodology*, Int. Conf. on Signal Processing Applications and Technology (ICSPAT), 1994

[3] M. Levy: *C Compilers for DSPs flex their Muscles*, EDN Access, Issue 12, www.ednmag.com, 1997

[4] M. Coors, O. Wahlen, H. Keding, O. Lüthje, H. Meyr: *TI C62x Performance Code Optimization*, DSP Germany, 2000

[5] Texas Instruments: TMS320C62xx CPU and Instruction Set Reference Guide, www.ti.com/sc/c6x, 1998

[6] Philips Semiconductors: www.trimedia.philips.com, 2000

[7] P. Briggs: *Register Allocation via Graph Coloring*, Doctoral thesis, Dept. of Computer Science, Rice University, Houston/Texas, 1992

[8] M. Lam: *Software Pipelining: An Effective Scheduling Technique for VLIW machines*, ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 1988, pp. 318-328

[9] S. Davidson, D. Landskov, B.D. Shriver, P.W. Mallett: *Some Experiments in Local Microcode Compaction for Horizontal Machines*, IEEE Trans. on Computers, vol. 30, no. 7, 1981, pp. 460-477

[10] J.A. Fisher: *Trace Scheduling: A Technique for Global Microcode Compaction*, IEEE Trans. on Computers, vol. 30, no. 7, 1981, pp. 478-490

[11] A. Aiken, A. Nicolau: *A Development Environment for Horizontal Microcode*, IEEE Trans. on Software Engineering, no. 14, 1988, pp. 584-594

[12] A. Capitanio, N. Dutt, A. Nicolau: *Partitioning of Variables for Multiple-Register-File Architectures via Hypergraph Coloring*, in: M. Cosnard, G.R. Gao, G.M. Silberman (eds.): IFIP Trans. A-50 – Parallel Architectures and Compilation Techniques, North Holland, 1994

[13] B.R. Rau, V. Kathail, S. Aditya: *Machine-Description Driven Compilers for EPIC and VLIW processors*, Design Automation for Embedded Systems, vol. 4, issue 2/3, Kluwer Academic Publishers, 1999

[14] B.W. Kernighan, S. Lin: *An Efficient Heuristic Procedure for Partitioning Graphs*, Bell Sys. Tech. Journal, Vol. 49, 1970

[15] M.F. Jacome, G. de Veciana: *Lower Bound on Latency for VLIW ASIP Data Paths*, Int. Conf. on Computer-Aided Design (ICCAD), 1999

[16] E. Özer, S. Banerjia, T.M. Conte: *Unified Assign and Schedule: A New Approach to Scheduling for Clustered Register File Microarchitectures*, MICRO-31, 1998

[17] J. Sanchez, A. Gonzales: *Instruction Scheduling for Clsutered VLIW Architectures*, Int. Symp. on System Sythesis (ISSS), 2000

[18] M.M. Fernandes, J. Llosa, N. Topham: *Partitioned Schedules for VLIW Architectures*, Int. Parallel Processing Symp. (IPPS), 1998

[19] P. Faraboschi, G. Desoli, J.A. Fisher: *Clustered Instruction-Level Parallel Processors*, Technical Report HPL-98-204, HP Labs, USA, 1998

[20] E. Stotzer, B. Huber, R. Tatge, A. Ward: *Programming a VLIW DSP in Assembly Language*, Proc. 2nd International Workshop on Compiler and Architecture Support for Embedded Systems (CASES), 1999

[21] M.R. Gary, D.S. Johnson: *Computers and Intractability – A Guide to the Theory of NP-Completeness*, Freemann, 1979

[22] S. Kirkpatrick, C.D. Gelatt, M.P. Vecchi: *Optimzation by Simulated Annealing*, Science, Vol. 220, 1983

[23] L. Davis: *Handbook of Genetic Algorithms*, Van Nostrand Reinhold, 1991

[24] LANCE Software: University of Dortmund, Dept. of Computer Science 12, ls12-www.cs.uni-dortmund.de/~leupers, 2000