Proceedings of the 32nd DAC,
pp. 593–598, San Francisco (CA),
June 12–16, March 6–9, 1995.

Conflict Modelling and Instruction Scheduling in Code Generation for In–House DSP Cores

Adwin H. Timmer*/**, Marino T.J. Strik**, Jef L. van Meerbergen** and Jochen A.G. Jess*
*Eindhoven University of Technology, Department of Electrical Engineering,
Design Automation Section, P.O. Box 513, 5600 MB Eindhoven, The Netherlands
**Philips Research Laboratories, Prof. Holstlaan 4, 5656 AA Eindhoven, The Netherlands

Abstract

Application domain specific DSP cores are becoming increasingly popular due to their advantageous trade–off between flexibility and cost. However, existing code generation methods are hampered by the combination of tight timing and resource constraints, imposed by the throughput requirements of DSP algorithms together with a fixed core architecture. In this paper, we present a method to model resource and instruction set conflicts uniformly and statically before scheduling. With the model we exploit the combination of all possible constraints, instead of being hampered by them. The approach results in an exact and run time efficient method to solve the instruction scheduling problem, which is illustrated by real life examples.

1. Introduction

Predefined DSP cores which are tuned towards specific application domains are becoming increasingly popular, due to their advantageous trade–off between *flexibility* and *cost*. Such a core is relatively flexible in comparison to an ASIC: different algorithms can be mapped on it, while an ASIC is a tailored solution for only one algorithm. On the other hand, domain specific DSP cores are more targeted towards a specific application domain, making them more suitable for such a domain than general processors: dedicated hardware is available for time critical tasks (e.g. a module performing a FFT butterfly in a single cycle). These cores also have an advantage over the combination of general purpose and ASIC components, because there is no communication bottleneck between different parts. Therefore a new research topic is emerging: 'retargetable' code generation for domain specific DSP cores and other application specific instruction–set processors (ASIPs).

The size of the application domain of a core is inversely proportional to the required efficiency. Because of the relatively high efficiency required, the use of domain specific DSP cores leads to new design tools and methods [Paul92]. Experiments show cases in which the utilization of the operation processing units (OPUs) in the core exceeds 90% of the total cycle budget [Strik95]. So there is a need for a code generator capable of generating very efficient (compact) microcode under *tight feasibility constraints*. With tight feasibility constraints we mean that both timing (from the algorithm) and resource (from the DSP core and instruction set) constraints are present. The combination of these constraints results in high OPU utilization rates, while the only objective is to find a feasible (correct) mapping from algorithm to DSP core.

2. Contributions of this paper

Code generation can roughly be divided into three interdependent subtasks: code selection, instruction scheduling and register binding. Previous approaches concentrate on the *code selection* problem [Marw93], [Liem94], [Praet94] or the *register binding* problem [Cheng94], [Lann94]. However, under the regime of tight feasibility constraints, many instances appear where heuristic approaches for the *instruction scheduling* problem render unsatisfactory results (i.e. they do not find a feasible schedule within the throughput constraints although such schedules do exist).

The existing scheduling methods do not produce satisfactory results because they are hampered by the combination of tight timing and resource constraints instead of exploiting them. On one hand, in the field of software compilation, the completion time of an algorithm is not that important in comparison with the hard constraints on the throughput of DSP algorithms. An exception is [Chou94], but in that approach the resulting schedule is fully serial, so no parallelism in the datapath is possible (which is needed in DSP applications). On the other hand, in the field of hardware compilation, most architectural synthesis systems do not treat hard resource constraints correctly (i.e. they often just add resources in order to find a solution).

In this paper we will therefore concentrate on *modelling* resource and instruction set conflicts and *exploiting* the combination of all possible constraints, thus obtaining an exact and run time efficient method to solve the instruction scheduling problem. The exploitation of the constraints leads to a reduction of the scheduling search space to a point where the solution space can be searched exhaustively in many cases. The target cores we consider are in–house DSP cores for which the application domains are relatively small and the microcode efficiency must be high. As a consequence of the use of in–house DSP cores, we can control the core architectures and the corresponding instruction set definitions, so we can adjust them to facilitate our code generation approach [Strik95]. The exact contributions of this paper are as follows.

- In section 3, we show how different resource constraints (with respect to OPUs, memory accesses, buses and multiplexers) can be modelled uniformly. Because in our case the instruction set

cannot steer all modules in the datapath simultaneously, the instruction set imposes additional restrictions on the amount of parallelism in the datapath. A method has been developed, so that these restrictions can be handled as if they are normal resource conflicts. This means amongst others that the instruction set conflicts are modelled statically before scheduling, thus making a compaction pass, used in other code generation systems like CodeSyn [Paul94], superfluous. Note that register file size constraints are not yet dealt with in the approach presented here. This is still a topic of further research.

- In section 4, we cast the different resource conflicts to a bipartite graph matching formulation. The formulation prunes the scheduling search space in polynomial time without limiting the solution space by exploiting combinations of resource and timing constraints. The method is based on the execution interval analysis of [Timm93], but is completely changed for our code generation. Because of the large number and the tightness of the different resource constraints, the approach is highly suitable for the retargetable code generation problem.

- In section 5, we propose an exact branch–and–bound method to solve the instruction scheduling problem. The approach searches for a correct ordering of the operations (from which a schedule can be derived in linear time), instead of directly generating exact time bounds for each operation. In section 6, results for real life examples show the efficiency of the approach.

3. Resource and instruction set conflicts

3.1. Register transfer generation

Preceding the instruction scheduling step, register transfers (RTs) and their dependencies are generated from an algorithmic input description using a generic architectural model, see figure 1. That figure shows a number of (possibly pipelined) OPUs. Each OPU input is connected with a register file (RF). The outputs of the OPUs are connected to RFs via buffers, buses and (optionally) multiplexers. RTs correspond to a complete (in this case single clock cycle) path from origin register files to a destination register file. So the RTs already contain the binding information on which resources actions from the input description are mapped. RTs are fully characterized by the resources that are used and the mode in

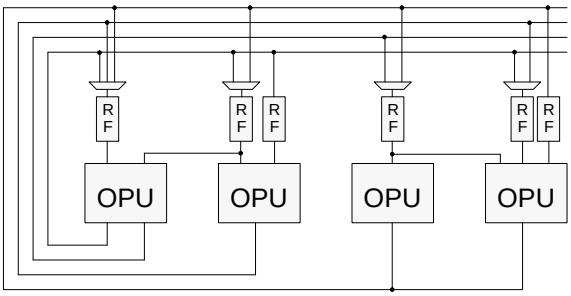

Figure 1: Generic datapath architecture.

```
Dest_1:  reg_2_ram_1 <– Source_1: reg_1_acu_1,
                        Source_2: reg_2_acu_1,
         acu_1          = add,
         buf_1_acu_1    = write,
         bus_1_acu_1    = 'add(Source_1,Source_2)',
         mux_2_ram_1    = pass[0, 1].
```

Figure 2: Example register transfer.

which these resources have to operate. An example is given in figure 2. The resources are found at the left hand side of the '=' sign and the mode (or usage) is positioned at the right hand side. Figure 2 shows an addition on an OPU called 'acu_1' and the storage of the result in a register of the OPU called 'ram_1' via one of the two available multiplexer inputs.

The RT generation step has equivalences to the instruction–set matching and selection techniques of other approaches like [Liem94]. However, in our case this step is done by an existing (architectural synthesis) RT generation tool from the Mistral 2^{TM} compiler [Nieu94]. The tool uses the architectural model of figure 1 as a starting point. Register files and busses that are merged in the actual core are taken into account by modifying the generated RTs [Strik95].

3.2. Resource conflicts

RTs can only be combined into a single instruction by a scheduler if there are no resource conflicts. If RTs do not use the same resources, then they can be combined. Otherwise it depends on the usage of these resources. At the left of figure 3, an RT is given that can be combined with the RT of figure 2: the usage of the shared resources is the same. The only difference between the two RTs is the destination RF (see the resources in bold in figure 3). At the right of figure 3, an RT has been given that *cannot* be packed into the same instruction as the RT of figure 2. The OPU is used differently (see the usage in bold in figure 3), which leads to a conflict.

All possible conflicts due to the resources can be modelled with the following overall conflict graph (OCG), which will be used by the graph matching formulation we introduce in section 4.

Definition 1:
OCG is an undirected graph represented by a tuple (V, E), where:
- V is the set of vertices representing all RTs;
- $E \subseteq V \times V$ is the set of edges;
  there is an edge $(v_i, v_j) \in E$ if and only if there is some resource that both $v_i \in V$ and $v_j \in V$ use, but in a different mode.

The OCG points out that the resource conflicts are modelled statically before scheduling. Two RTs can be packed into one instruction if they are not adjacent to each other in the OCG (and, of course, if dependency relations between RTs are not violated). For all OCG cliques only one RT at the time can be packed into one instruction. So solving the resource conflicts of a design problem can be interpreted as finding different independent sets of RTs for every instruction (or clock cycle) that do not violate the dependency relations between RTs.

```
Dest_1: reg_2_acu_1 <–  Source_1: reg_1_acu_1,
                       Source_2: reg_2_acu_1,
        acu_1          = add,
        buf_1_acu_1    = write,
        bus_1_acu_1    = 'add(Source_1, Source_2)',
        mux_1_acu_1    = pass[0, 1].
```

```
Dest_1: reg_2_ram_1 <–  Source_1: reg_1_acu_1,
                       Source_2: reg_2_acu_1,
        acu_1          = addmod,
        buf_1_acu_1    = write,
        bus_1_acu_1    = 'add(Source_1, Source_2)',
        mux_2_ram_1    = pass[0, 1].
```

Figure 3: RTs without *(left)* and with *(right)* a conflict with the RT of figure 2.

3.3. Instruction set conflicts

A given DSP core is not only specified by its datapath but also by its instruction set. In our case the instruction set cannot steer all modules in the datapath simultaneously, so it imposes additional restrictions on the amount of parallelism in the datapath. For example *load immediate* is often a separate instruction class (or 'optype') during which no other operations can take place.

In our approach these restrictions are modelled by adding extra edges to the OCG defined in the previous subsection [Strik95]. There is however a catch in the approach. The previous subsection showed that the resource conflicts from the datapath are modelled statically before scheduling by the OCG. The matching formulation we introduce in section 4 uses the OCG and the conflicts modelled by it. So the question arises whether such a static modelling of the instruction set conflicts imposes any restrictions or demands on the instruction set definition itself. (Recall that we consider in–house DSP cores, so we can control the definition of the instruction sets to make them suitable for code generation.)

Modelling the instruction set conflicts by additional edges in the OCG is only valid, if the result is that *every* arbitrary independent set of RTs from the OCG always corresponds to a legal instruction. This means amongst others that the NOP (no operation) must be a possible instruction, as well as each individual RT class on its own. So modelling the instruction set conflicts by additional OCG edges puts some special demands on the definition of the instruction sets, see also [Strik95]. However, these demands are very well acceptable in real life situations and have no influence on the efficiency of the implementation.

4. Bipartite graph matching formulation

4.1. Background

The quality of the result and the run times of (exact) scheduling approaches heavily depend on powerful pruning techniques. An important device to support pruning is the operation execution interval (OEI). An OEI constrains the interval of clock cycles to be assigned to an RT in any schedule. If resource constraints are not considered, then an OEI is given by the ASAP and ALAP cycles of the RT under the assumption of unlimited resources. Recent research [Timm93] showed, that the search space of schedulers that are both resource and time constrained can be pruned considerably by reducing the OEIs. That (polynomial run time) approach is based on a graph matching formulation exploiting both resource and time constraints and does not exclude any possible schedule.

Because of the large number and tightness of the different resource and instruction set constraints, such a pruning approach is highly suitable for the retargetable code generation problem. We therefore apply a similar analysis based on bipartite graph matching to map algorithms to DSP cores. However, a rigorous adaptation of the standard approach is needed to make it suitable for our code generation, due to the following reasons.

- Originally only resource constraints with respect to OPUs were taken into account. The technique is extended for all other resource types that are part of an RT, i.e. constraints with respect to memory accesses, buses, multiplexers and the instruction set are also considered.

- In many cases, the loops in a signal flow graph (SFG) have to be 'folded' (see section 4.3) to satisfy the throughput constraints. Cyclic signal flow graphs were not considered in [Timm93], and loop folding also results in extra timing constraints for the RTs because the consumption of a value must occur before a new version of the value is produced.

- In general the number of times a certain resource is occupied is not known beforehand in the RT model we use. If two RTs do not have any resource conflicts although they do use the same resources (i.e. they use resources in the same mode), then it depends on the final schedule whether these resources are used once or twice for the two RTs.

4.2. Module execution intervals

The considerations mentioned above have a large impact on the calculation of the so called *module execution intervals* (MEIs), which account for the resource (or instruction set) conflicts. MEIs can be calculated for each resource separately and are part of the bipartite graph matching formulation. Within the interval of each MEI, the corresponding resource has to be occupied by some RT (so the numbers of OEIs and MEIs for each resource are equal). Consequently, if the number of cycles of a MEI is equal to one, then the resource must be utilized in that cycle.

For reasons of simplicity, in the above the MEIs were said to be calculated for each resource separately. However, because in general the number of times a certain resource is occupied is not known beforehand (see the previous subsection), it is very cumbersome and difficult to calculate the MEIs per separate resource, see [Timm95b]. For the same reason, such a calculation will inevitably be less accurate than the original approach of [Timm93]. As the objective of the matching formulation we present in this section is to model the combination of resource and timing conflicts between different RTs as accurately as possible, the formulation would not be as powerful as the original approach.

Luckily it is possible to overcome these problems, namely by calculating the MEIs differently (i.e. not per separate resource). Every clique of RTs from the OCG represents RTs that have resource conflicts with each other. It is possible to construct a clique cover such that *all* edges in the OCG are induced (at least) once by a clique of RTs from the clique cover. MEIs can now be calculated for each clique from such a cover, leading to a more accurate, more powerful and much simpler approach than an approach in which MEIs are calculated per resource [Timm95b]. Note that a clique from the OCG can incorporate resource conflicts from different resources; the MEIs are therefore not calculated for each individual resource anymore.

4.3. Definition and calculation of MEIs

The schedule of a SFG can be divided into a preamble, a loop body and a postamble. RTs are scheduled for the first time in either the preamble or the loop body. In our case, the delay of each RT is one clock cycle. The throughput of a schedule is given by the data introduction interval (dii), so the execution of an RT is repeated every dii cycles. The schedule of an RT is therefore fully defined by the first clock cycle in which it is executed together with the dii. The schedule of each RT corresponds to the occupation of resources during one *time potential* (i.e. 'a specific instruction cycle that returns every dii cycles') in the loop body. The number of times a SFG is 'folded' depends on the latency. If the latency equals the dii, then the SFG is not folded. If the latency is twice the dii, then the SFG is folded once, if the latency is three times the dii, then the SFG is folded twice, etcetera.

Resource conflicts occur when two RTs are scheduled at the same time potential in the loop body, while they use the same resource in a different mode. Consider a clique from the OCG clique cover. Let V' be the set of RTs in that clique, let Φ be the set of feasible (i.e. correct) schedules, and let $\sigma_\phi(v)$ be the time potential at which $v \in V'$ is scheduled in case of some schedule $\phi \in \Phi$. A schedule ϕ imposes a notion of order on the set V' by ordering the set according to the time potentials, see definition 2. Note that two scheduled RTs from an OCG clique cannot have equal potentials.

Definition 2: Given some schedule $\phi \in \Phi$, we define $\prec_\phi$ as a linear ordering relation on the set V' as follows:

$$\forall_{\phi \in \Phi} \; \forall_{v,w \in V'} : \; v \prec_\phi w \Leftrightarrow \sigma_\phi(v) < \sigma_\phi(w).$$

As $\prec_\phi$ is a linear ordering it can be used to assign an integer value $i \in I$, $I = [1, |V'|]$ to any $v \in V'$. We capture this by defining a bijective function $\varepsilon_\phi: I \rightarrow V'$. Thus $\varepsilon_\phi(i)$ is the i^{th} RT under the linear order induced by the schedule ϕ. We are now prepared to formally introduce the notion of a MEI.

Definition 3: Consider the set of RTs from V' assigned value $i \in I$ over the set of all schedules Φ. Then the module execution interval MEI(i) is defined by the following interval of time potentials:

$$\text{MEI(i)} = \left[\min_{\phi \in \Phi}\left(\sigma_\phi\left(\varepsilon_\phi(i)\right)\right), \; \max_{\phi \in \Phi}\left(\sigma_\phi\left(\varepsilon_\phi(i)\right)\right) \right].$$

Thus, for any schedule $\phi \in \Phi$, the time potential of the i^{th} RT from V' must be within the interval of time potentials of MEI(i). Unfortunately, it is not possible to calculate the exact bounds of the MEIs in polynomial time (otherwise the existence of a feasible schedule under given time and resource constraints could be decided in polynomial time). So we have to be content with (conservative) estimates of those bounds which preserve the integrity of the solution space, see property 1.

Property 1:
The estimates of the first time potential, first(MEI), and the last time potential, last(MEI), of a MEI have to satisfy the following.

$$\forall_{\phi \in \Phi} \; \forall_{i \in I} : \sigma_\phi\left(\varepsilon_\phi(i)\right) \geq \text{first(MEI(i))} \wedge \sigma_\phi\left(\varepsilon_\phi(i)\right) \leq \text{last(MEI(i))}$$

Let FTP(v) be the first time potential in which register transfer $v \in V'$ can be scheduled, let the set V' be ordered by increasing FTP (if two RTs have the same FTP then this tie is broken in an arbitrary way), and let $V'(i)$, $i \in I$, be the i^{th} RT in that order. If MEIs are calculated per OCG clique, then the following two properties hold (note that property 3 does not hold in case MEIs are calculated per separate resource).

Property 2: Start of MEI(i) cannot be smaller than the i^{th} FTP.

$$\forall_{i \in I} : \text{first(MEI(i))} \geq \text{FTP(V(i))}.$$

Property 3: At each time potential, at most one MEI can start.

$$\forall_{2 \leq i \leq |V'|} : \text{first(MEI(i))} \geq \text{first(MEI(i–1))} + 1.$$

Theorem 1: Algorithm 1 calculates (estimates) for all MEI(i), $i \in I$, the value first(MEI(i)) while satisfying property 1. The last time potentials of the MEIs can be determined similarly.

Proof. The proof follows directly from property 2 and 3. ■

Algorithm 1: calculate / estimate first time potentials of MEIs.

first(MEI(1)) := FTP(V'(1));
for (i := 2 to $|V'|$) →
    first(MEI(i)) := max {FTP(V'(i)), first(MEI(i – 1)) + 1};

4.4. Bipartite schedule graphs

With the calculated MEIs, all resource / instruction set conflicts can be cast to *bipartite schedule graphs* (BSGs). In the following definition these BSGs incorporating both RTs and MEIs are given.

Definition 4: The bipartite schedule graph BSG(V') for OCG clique V' under the given timing and resource constraints is an undirected bipartite graph represented by a tuple (N, A), where:

- $N = V' \cup R'$ is the set of vertices with $V' \cap R' = \emptyset$, $|V'| = |R'|$, and $R' = \{\text{MEI(i)} \mid 1 \leq i \leq |V'|\}$;
- $A \subseteq V' \times R'$ is the set of edges; initially there is an edge $(v, n) \in A$ if and only if $v \in V'$ can be scheduled in the interval of time potentials of module execution interval $n \in R'$.

For each feasible schedule a corresponding *complete* matching exists (but not all complete matchings represent feasible schedules). The OEIs of the RTs can be reduced by identifying the irreducible components [Dulm63] of the BSGs. Edges not belonging to these components cannot be part of any complete matching, so they can be removed without excluding any schedule. Such a removal can reduce the incident OEI, because an OEI cannot be larger than the union of adjacent MEIs (after the time potentials in the MEIs are translated back into clock cycles). A more thorough discussion on BSGs and reducing OEIs can be found in [Timm93].

If an edge removal leads to the reduction of an OEI, a new run can be started to reduce the OEIs even more (note that after an OEI reduction, the OEI can be a set of non–overlapping intervals of clock cycles instead of one interval of clock cycles). Such a run starts with determining new OEIs based on the reduced OEIs and the dependency relations in the SFG (so the reduction of the OEIs in one BSG can induce the reduction of OEIs in other BSGs). Also the MEIs are calculated from scratch in a new run. The union of all OEIs contains $O(|C| \cdot |V|)$ cycles, where $|C|$ is the latency, so the number of runs is in the worst case $O(|C| \cdot |V|)$. However, in practice the algorithm already stops after a few runs. With this approach the execution intervals of the RTs (and consequently the scheduling search space) can be reduced by exploiting both resource and timing constraints.

5. Instruction scheduling

A common approach to schedule RTs is to assign them directly to specific cycle steps. In [Timm95a], it is proven that the existence of a schedule for a given SFG with timing and resource constraints can be decided more efficiently by finding *correct orderings* of the RTs in all BSGs of a design problem. With a correct ordering we mean an ordering that corresponds to (i.e. can be induced by) a feasible schedule, see definition 5.

Definition 5: A linear ordering $\prec$ of the RTs in a BSG is correct if there is a schedule that induces that ordering, i.e. if

$$\exists_{\phi \in \Phi} : v \prec w \Leftrightarrow \varepsilon_{\phi}^{-1}(v) < \varepsilon_{\phi}^{-1}(w).$$

If a linear ordering on the RTs in a BSG is imposed, then each RT is adjacent to *at most* one MEI, see [Timm95a]. A correct ordering implies a bijection between OEIs and MEIs and, consequently, defines a complete matching in the BSG. In case of a correct ordering, all adjacent reduced OEIs and MEIs will have equal clock cycle intervals after the execution interval analysis of section 4 leaves its iteration (otherwise an OEI or MEI could be reduced further and the analysis would continue with a new run). This leads to the following theorem.

Theorem 2: If for each OCG clique from an OCG clique cover of a design problem a linear ordering on the corresponding RTs is imposed and the set of these orderings is not detected as infeasible by the execution interval analysis of section 4, then these orderings are correct and a feasible schedule can be derived from the result of the execution interval analysis in linear time. The feasible schedule is derived by scheduling all RTs in the first cycle of their reduced OEI after the execution interval analysis has left its iteration. The proof of this theorem can be found in [Timm95a].

Because the execution interval analysis runs in polynomial time, it follows from theorem 2 that the correctness of some ordering or complete matching of the RTs can be checked in polynomial time. A matching can represent more than one schedule, so the number of different matchings is equal to or less than the number of different schedules. Because the check for correctness is not more accurate when the RTs are directly assigned to specific cycle steps, it is more efficient to search for a correct ordering instead.

The above leads to the following scheduling approach. We start from the initial BSGs and match the MEIs one–by–one to specific RTs, while removing the edges that can no longer be part of a complete matching. These edges are the ones previously connected to the RT and the MEI that have just been matched (except for the one between them), together with other edges that are no longer part of any irreducible component. So each time a matching between an RT and a MEI has taken place, the whole execution interval analysis can be rerun to continue the pruning of the search space. The matching of RTs and MEIs is a process in which the initial BSGs get more and more sparse (an edge that is once removed does not return in a BSG as long as a matching is not revoked). The search space is also getting smaller during this process because the OEIs and MEIs are reduced more and more as well.

The priority functions in this instruction scheduling process are as follows. First we look for the MEI with the smallest end potential. This MEI is matched with an RT, and the RT is also matched with the first MEIs in the other BSGs of which the RT is an element. Then the next MEI with the smallest end potential (which is not yet matched) is selected, then matched with an RT and so on. If a matching leads to an infeasible schedule, then the matching is revoked and another RT is matched to the MEI. So a branch–and–bound approach is applied to obtain an exact scheduler.

6. Experiments and results

In [Strik95], a DSP core together with an instruction set has been given. We have mapped the examples of table 1 onto this core, and tried to obtain the highest throughput possible. The examples range from a simple delay line to a portable audio application (which is a real life industry example). The instruction–set scheduler based on graph matching has been implemented in C++ using the architectural interface of the NEAT (New Eindhoven Architectural synthesis Toolbox) system [Heij94]. In table 1, we have compared our approach with an industrial high–level synthesis (HLS) list scheduler.

The table shows, that our approach finds the guaranteed optimal throughput within acceptable run times for all but two cases. The most interesting examples are the largest examples 4a/b and 5a/b. Example 4b shows the largest difference between the achieved throughputs of both schedulers. The largest example 5a/b consists

Table 1: Throughput results for various examples.

Example	#OCG cliques	size largest clique	lower bound throughput	HLS scheduler	NEAT instruction scheduler	CPU** NEAT
1a: RAM delay line (12 RTs, unfolded)	4	4	5	5*	5*	0.3 sec
1b: RAM delay line (12 RTs, folded once)	4	4	4	5	4*	0.3 sec
2a: FIR filter (37 RTs, unfolded)	11	7	17	17*	17*	1.2 sec
2b: FIR filter (37 RTs, folded once)	11	7	9	9*	9*	1.2 sec
3a: FIR & Bass Boost (114 RTs, unfolded)	12	16	30	31	30*	7.9 sec
3b: FIR & Bass Boost (114 RTs, folded once)	12	16	25	26	26	8.0 sec
4a: Sym. FIR & Bass B. (288 RTs, unfolded)	22	29	36	43	38	56.2 sec
4b: Sym. FIR & Bass B. (288 RTs, folded)	22	29	29	36	29*	56.3 sec
5a: Portable audio appl. (358 RTs, unfolded)	21	58	62	67	62*	138.3 sec
5b: Portable audio appl. (358 RTs, folded)	21	58	58	61	58*	139.1 sec

* the throughput equals the lower bound estimation, i.e. is guaranteed to be optimal.
** measured on a HP 9000/735 workstation.

of 58 multiplications, 58 additions, clip actions and delays. The throughput constraint of this real life application is 64 cycles. The results on example 4a show, that a dedicated instruction–set scheduler exploiting the combination of resource and timing constraints is needed to meet this throughput constraint without folding. The result on example 5b shows, that our approach succeeds in generating a schedule in which the multiplier and ALU in the DSP core have a 100% utilization during all clock cycles.

7. Conclusions

In this paper, we presented a code generation approach for in–house DSP cores. The approach models resource conflicts (originating from both a DSP core and an instruction set) uniformly and before scheduling. The different resource conflicts are cast to a bipartite graph matching formulation to prune the scheduling search space. In this way the instruction scheduling step can exploit the combination of all possible constraints instead of being hampered by them. From the matching formulation, we have derived an exact (branch–and–bound) method to solve the instruction scheduling problem. The branch–and–bound process does not assign a clock cycle to each register transfer directly, but tries to find a correct ordering of the transfers instead. From such an ordering a schedule can be derived in linear time. Real life examples illustrated the quality and run time efficiency of the approach.

References

[Cheng94] W.–K. Cheng and Y.–L. Lin, "Code Generation for a DSP Processor", Proc. Int. Symp. on HLS, pp. 82–87, Niagara–on–the–Lake (Canada), May 1994.

[Chou94] P. Chou and G. Borriello, "Software Scheduling in the Co–Synthesis of Reactive Real–Time Systems", Proc. of the 31st DAC, pp. 1–4, San Diego (CA), June 1994.

[Dulm63] A.L. Dulmage and N.S. Mendelsohn, "Two Algorithms for Bipartite Graphs", J. Soc. Indust. Appl. Math., Vol. 11, No. 1, pp. 183–194, 1963.

[Heij94] M.J.M. Heijligers, H.A. Hilderink, A.H. Timmer and J.A.G. Jess, "NEAT: An Object Oriented High–Level Synthesis Interface", Proc. ISCAS–94, pp. 1.233–1.236, London (UK), May 1994.

[Lann94] D. Lanneer, M. Cornero, G. Goossens and H. De Man, "Data Routing: a Paradigm for Efficient Data–Path Synthesis and Code Generation", Proc. Int. Symp. on HLS, pp. 17–22, Niagara–on–the–Lake (Canada), May 1994.

[Liem94] C. Liem, T. May and P. Paulin, "Instruction–Set Matching and Selection for DSP and ASIP Code Generation", Proceedings ED&TC (EDAC–ETC–EuroASIC) '94, pp. 31–37, Paris (France), March 1994.

[Marw93] P. Marwedel, "Tree–Based Mapping of Algorithms to Predefined Structures", Digest of Technical Papers of ICCAD–93, pp. 586–593, Santa Clara (CA), Nov. 1993.

[Nieu94] K. van Nieuwenhoven, J. de Moortel, D. Genin and S. Note, "Mistral 2 a True Architectural Synthesis™ Tool: from a Behavioral Specification down to a Register Transfer Level Description", DSP Applications and Multimedia, Oct. 1994.

[Paul92] P.G. Paulin, "DSP Design Tool Requirements for the Nineties: An Industrial Perspective", 6th Int. HLS Workshop, Laguna Niguel (CA), Nov. 1992.

[Paul94] P.G. Paulin, C. Liem, T.C. May and S. Sutarwala, "CodeSyn: A Retargetable Code Synthesis System", Int. Symp. on HLS, Niagara–on–the–Lake (Canada), May 1994.

[Praet94] J. Van Praet, G. Goossens, D. Lanneer and H. De Man, "Instruction Set Definition and Instruction Selection for ASIPs", Proc. Int. Symp. on HLS, pp. 11–16, Niagara–on–the–Lake (Canada), May 1994.

[Strik95] M. Strik, J. Van Meerbergen, A. Timmer, J. Jess and S. Note, "Efficient Code Generation for In–House DSP–Cores", Proc. ED&TC (EDAC–ETC–EuroASIC) '95, Paris (France), March 1995.

[Timm93] A.H. Timmer and J.A.G. Jess, "Execution Interval Analysis under Resource Constraints", Digest of Technical Papers of ICCAD–93, pp. 454–459, Santa Clara (CA), Nov. 1993.

[Timm95a] A.H. Timmer and J.A.G. Jess, "Exact Scheduling Strategies based on Bipartite Graph Matching", Proc. ED&TC (EDAC–ETC–EuroASIC), Paris (France), March 1995.

[Timm95b] A.H. Timmer, Ph.D. Thesis (to appear in 1995).