Available at
www.ComputerScienceWeb.com
POWERED BY SCIENCE DIRECT®

Signal Processing 83 (2003) 1411–1425

SIGNAL PROCESSING

www.elsevier.com/locate/sigpro

List-coloring of interval graphs with application to register assignment for heterogeneous register-set architectures

Thomas Zeitlhofer*, Bernhard Wess

Institute of Communications and Radio-Frequency Engineering, Vienna University of Technology, Gusshausstrasse 25/389, A-1040 Vienna, Austria

Received 28 November 2002

Dedicated to Prof. Wolfgang Mecklenbräuker on occasion of his 65th birthday

Abstract

This article focuses on register assignment problems for heterogeneous register-set VLIW-DSP architectures. It is assumed that an instruction schedule has already been generated. The register assignment problem is equivalent to the well-known coloring of an interference graph. Typically, machine-related constraints are mapped onto the structure of the interference graph. Thereby favorable characteristics with regard to coloring, the interval graph properties, get lost. In contrast, we present an approach that does not change the structure of the interference graph. Constraints implied by heterogeneous architectures are mapped to a specific coloring problem that is known as *list-coloring*. Exploiting the interval graph properties of the interference graph, we derive a list-coloring algorithm that allows us to generate optimum solutions even for large basic blocks. The proposed technique can also be applied to similar resource assignment problems like functional unit assignment.

Keywords: Interval graph; Graph coloring; List coloring; Register assignment; Heterogeneous register-set architectures

1. Introduction

Modern *very long instruction word* (VLIW) architectures for *digital signal processors* (DSPs) support instruction-level parallelism (ILP) and provide several functional units. The more the functional units operate at the same time, the higher the achievable performance. The design of the data paths becomes a critical factor because a lot of data have to be supplied in parallel in order to keep the functional units busy. So in general, quite a few registers have to be provided which increases the complexity of full interconnects between registers and functional units. Therefore, registers are often grouped into the so-called register files. Each register file is typically connected only to a subset of the functional units.

Determining register assignments for these heterogeneous register sets is a complex combinatorial problem and a critical phase in code generation that highly affects performance. If it is not possible to generate a register assignment for a given instruction schedule, additional register-to-memory transfer instructions have to be inserted which typically degrades runtime performance and increases power consumption.

* Corresponding author. Tel.: +43-1-58801-38962.
E-mail addresses: thomas.zeitlhofer@nt.tuwien.ac.at (T. Zeitlhofer), bernhard.wess@nt.tuwien.ac.at (B. Wess).
URL: http://swan.nt.tuwien.ac.at/tz

0165-1684/03/$ - see front matter
doi:10.1016/S0165-1684(03)00089-6

While this article presents an optimum solution for the register assignment problem, our proposed approach is applicable to other problems in VLIW-DSP code generation as well. If we consider the set of functional units, typically, each unit is dedicated to perform certain types of instructions. Often some instructions may also be executed on more than one functional unit, e.g. add-operations may be calculated by ALUs *or* multiply accumulate (MAC) units. Assigning functional units for these heterogeneous structures is also a complex phase in code generation and similar to the register assignment problem.

In a generalized view, both functional units and registers can be regarded as resources that have to be assigned for certain time intervals (compare Section 2). To avoid confusion between registers, functional units, and resources in general, we focus on the register assignment problem throughout this article.

If registers are assigned for certain time intervals, these time intervals are defined by the instruction schedule. Instruction scheduling and register assignment have a fundamental interaction related to the order in which they are performed:

(1) By doing register assignment prior to scheduling, overlapping register lifetimes can be taken into account during scheduling. Because of the reuse of registers and functional units, additional dependencies may be introduced which often unnecessarily restrict the search space for the scheduler.
(2) If register assignment is carried out after instruction scheduling, in general, the existence of a proper register assignment cannot be guaranteed a priori. Determining register assignments for a given schedule and heterogeneous architectures becomes a challenging task.

In this article the focus is on strategy 2 where registers are assigned for an already fixed instruction schedule. Our overall optimization goal is runtime efficiency. As we do not discuss scheduling techniques here, we assume that an optimized schedule has already been generated. For all techniques following strategy 2 the overall performance depends on this initial schedule.

The goal of register assignment is to keep program variables in registers. If the architecture does not provide a sufficient number of registers, program variables have to be stored in memory temporarily. [1] Transferring program variables to and from memory requires additional instructions that are denoted as *spill code*. In general, this degrades performance due to extra processor cycles or pipeline latencies. A register assignment is *proper* if no spill code is required. So we need to know whether a proper register assignment exists. Typically, heuristics are applied but may fail even if assignments exist. In contrast, our optimum approach generates a proper register assignment if it exists at all. So the register assignment problem we consider is in fact not an optimization problem but essentially a decision problem. In addition to the decision problem, at least one specific register assignment has to be constructed if it is known to exist.

A general target architecture model is assumed that provides heterogeneous registers and heterogeneous functional units. That is, registers or functional units are not arbitrarily interchangeable and therefore restrictions have to be satisfied. The proposed technique is applicable to program sections that are specified by homogeneous atomic data flow graphs [18,19] which are denoted as basic blocks in compiler terminology.

If register assignment is formulated as a graph-coloring problem, machine-related restrictions due to heterogeneous architectures are often mapped onto the interference graph [8]. The resulting coloring problem is a complex combinatorial optimization problem and often we have to rely on heuristics. The interference graph basically models interfering lifetimes of program variables. If these lifetimes correspond to continuous time intervals, the interference graph is an interval graph that has several special properties. Modifying the structure of this graph to include architectural constraints may destroy these properties. In this paper, we show that a systematic exhaustive exploration of the search space is tractable for typical real-world problems when architecture constraints are *not* mapped onto the structure of the interference graph but on a *list-coloring* problem for interval graphs. This means that it is possible to generate proper register assignments and there is no need for heuristics.

[1] As an alternative the register-set architecture may be modified when we consider the design of an application-specific instruction-set processor (ASIP).

This article is organized as follows. In Section 2 we consider a simple digital signal processing system to demonstrate the assignment problems with respect to functional units and registers in case of heterogeneous architectures. We discuss related work in Section 3 where different strategies are compared to handle register assignment problems depending on the order of scheduling and register assignment. This discussion gives an overview of heuristics that are typically applied. Additionally, a technique related to functional unit assignment is compared to our approach. In Section 4, the proposed technique is motivated by discussing the computational complexity. In a detailed analysis, we emphasize the special properties of interval graphs which are the basis of the proposed approach. This discussion is followed by a demonstration of the proposed technique in Section 5. By means of this example, an algorithm for the register assignment problem is developed. The performance of this algorithm is shown in Section 6 for several examples. Conclusions are given in Section 7.

2. Introductory example

To clarify the register assignment problem and the similarities of the functional unit assignment problem in case of heterogeneous architectures, we consider an IIR-filter in direct form II (Fig. 1) as a simple example.

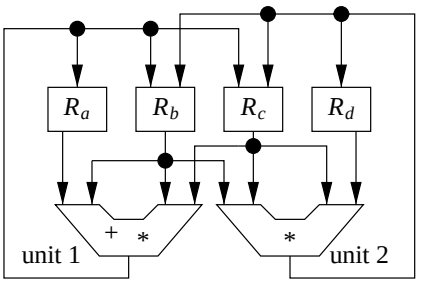
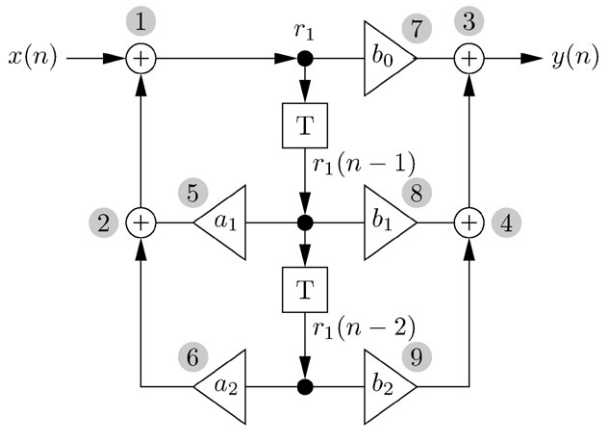


Fig. 2. Example: heterogeneous architecture.

Nine different operations $I_1, I_2, \ldots, I_9$ are required to compute one sample of the output signal $y(n)$. The intermediate results are denoted as $r_1, r_2, \ldots, r_9$ and correspond to nine symbolic registers. To calculate these results, the heterogeneous architecture shown in Fig. 2 is used. There are two functional units where unit 1 supports both addition and multiplication but unit 2 is a multiplier only. Additions are executed in a single clock-cycle while multiplications require two clock-cycles.

Analyzing data dependencies between the instructions $I_1, I_2, \ldots, I_9$ yields the precedence graph shown in Fig. 3.

A valid schedule is shown in Fig. 4(a). Here we exploit ILP so that it takes only eight cycles to compute nine instructions. Given this schedule where no



(a)

$$
\begin{aligned}
I_1: &\quad r_1 = x(n) + r_2\\
I_2: &\quad r_2 = r_5 + r_6\\
I_3: &\quad r_3 = r_4 + r_7\\
I_4: &\quad r_4 = r_8 + r_9\\
I_5: &\quad r_5 = a_1 * r_1(n-1)\\
I_6: &\quad r_6 = a_2 * r_1(n-2)\\
I_7: &\quad r_7 = b_0 * r_1\\
I_8: &\quad r_8 = b_1 * r_1(n-1)\\
I_9: &\quad r_9 = b_2 * r_1(n-2)
\end{aligned}
$$

(b)

Fig. 1. Example: IIR-filter: (a) signal flow graph and (b) nine operations.

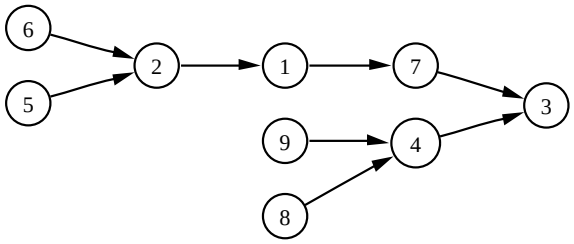


Fig. 3. Example: precedence graph.

data dependencies are violated, we have to answer the question if it is computable on the given heterogeneous architecture.

At first, a functional unit has to be assigned to each instruction. This problem can be formulated as a graph-coloring problem. A so-called interference graph is constructed where each vertex is related to an instruction or, to be more precise, represents the functional unit that executes this instruction. Two vertices are connected by an edge if the corresponding two functional units are active at the same time. Depending on the usage of functional units and their latency, the graph may become decomposed into subgraphs (Fig. 4(b)).

Our architecture provides two functional units. So, for the schedule to be feasible, the interference graph has to be colored using at most two colors. Since the functional units are not equivalent, there is a list of admissible functional units for each instruction. In graph terminology, this means that there is a list of admissible colors for each vertex. These lists are depicted in Fig. 4(b). The lists for the multiplications $(I_5, I_6, I_7, I_8, I_9)$ contain both functional units while the lists for the additions (I_1, I_2, I_3, I_4) only contain unit 1. Because the lists are not identical for all vertices, we are facing a *list-coloring problem* [12]. This is in contrast to the coloring problem for homogeneous architectures where all color sets are identical. For this simple example, the coloring problem can be solved by hand. In general, however, list-coloring is $\mathcal{NP}$-complete and heuristics have to be applied. We show in Section 4 that, by exploiting the interval graph properties of the interference graph, an optimum list-coloring technique is applicable to typical real-world problems.

For now we assume that unit 1 is assigned to instructions $I_1, I_2, I_3, I_4, I_6, I_7$ and unit 2 is assigned to instructions I_5, I_8, I_9.

The next phase in code generation is to assign registers that hold intermediate results. To simplify this

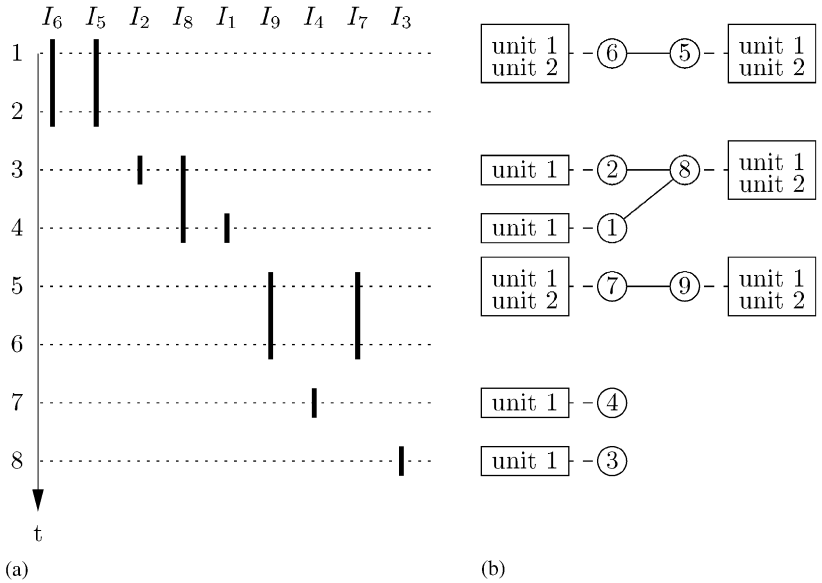


Fig. 4. Example: schedule and functional unit assignment: (a) usage of functional units and (b) functional unit interference graph.

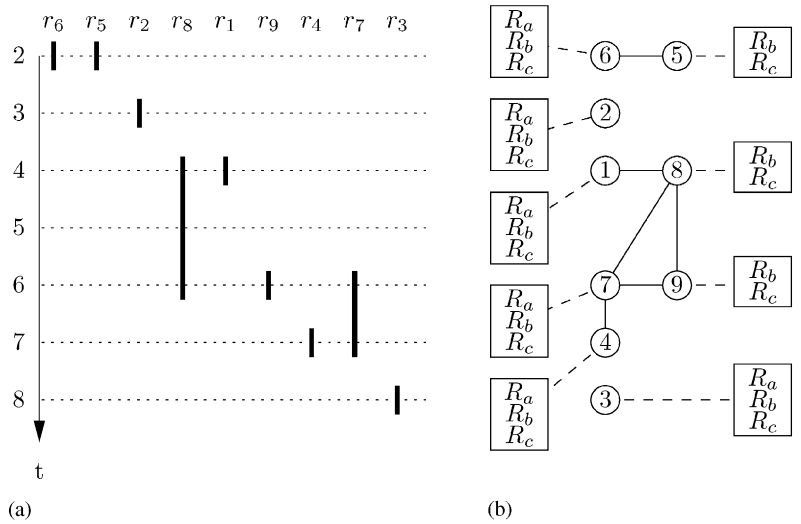


Fig. 5. Example: schedule and register assignment: (a) register usage and (b) register interference graph.

example, we will consider only the intermediate results $\{r_1, r_2, \ldots, r_9\}$ and not constants $\{a_1, a_2, b_0, b_1, b_2\}$ or delayed values $\{r_1(n-1), r_1(n-2)\}$ for which additional register and/or move instructions may be necessary. According to the given schedule, the live ranges for the intermediate results are shown in Fig. 5(a). Typically, register assignment problems are mapped also to graph-coloring problems. Here a vertex corresponds to a program variable and two vertices are connected by an edge if the corresponding variables are live at the same time. The resulting graph is shown in Fig. 5(b). Admissible target registers for instructions are defined by the intersection of two register sets. The first is defined by all registers the producing instruction can write to and the second is defined by all registers the consuming instruction can read from. There are four registers available but they cannot be used interchangeably. Consider r_9 which is produced by instruction I_9. Possible target registers are R_b, R_c, and R_d because I_9 is executed by unit 2. The value of r_9 is consumed by I_4 which is executed by unit 1. According to Fig. 2, I_4 can only use R_a, R_b, and R_c as source operands. We conclude that variable r_9 can only be placed either in register R_b or R_c. The lists of admissible registers for all program variables are shown in Fig. 5(b). We are facing a list-coloring problem again. This problem can be solved if $\{r_6, r_2, r_1, r_7, r_3\}$ are placed in R_a, $\{r_5, r_8, r_4\}$ are placed in R_b, and $\{r_9\}$ is placed in R_c.

This example shows that register assignment and functional unit assignment for heterogeneous architectures are challenging tasks. Typically, heuristics are used as discussed in Section 3. But heuristics may fail to find proper assignments even if they exist. Exhaustive search strategies on the other hand do not seem to be appropriate either because the problem is $\mathcal{NP}$-complete.

3. Related work

There is a strong relationship between the effective use of a machine's register set and the run-time performance especially for architectures supporting ILP. The process of deciding which values to keep in registers at each point in the generated code is called *register allocation*. Determining a specific register for each value to reside in is called *register assignment*. Machines with heterogeneous register sets provide several types of registers. In this case, register allocation and register assignment are different tasks. Admissible registers for each variable in the code depend both on the instructions' type and instruction schedule.

Typically, graph-coloring techniques are applied to register allocation. The first register allocators based

on graph coloring have been introduced in [7,11]. There are many techniques that improve these original register allocators, e.g. [1,5,6]. Often coloring techniques are also applied to register assignment.

Several scheduling techniques are known. For most applications, however, we have to rely on heuristics since the computational complexity of optimum techniques is intractable even for small-sized problems. Most popular are greedy techniques such as *list scheduling* and *critical-path strategies* [17]. For time-critical applications, modern heuristics, e.g. *simulated annealing* or *genetic algorithms*, may be applied to scheduling under pipelining constraints for architectures with several heterogeneous functional units [22].

Since scheduling and register allocation/assignment are highly interdependent, it is desirable to somehow solve these two problems simultaneously. Corresponding to strategy 1 described in Section 1, one approach is to find a register allocation/assignment without introducing additional dependencies. Then the scheduler still has all parallelization options. This has been proposed in [20] where the coloring of a so-called *parallel interference graph* provides a register allocation/assignment that does not introduce additional dependencies. Roughly speaking, the parallel interference graph combines the actual interference graph from an initial schedule with all other possible interference edges that are not already covered by data dependencies or machine constraints. Although the maximum degree of freedom is preserved for the scheduler, the number of registers required by this approach is not necessarily the minimum for a certain basic block since it depends on the initial schedule.

It is well known that efficient algorithms exist for the coloring of interval graphs [13] with a minimum number of equivalent colors. Although the parallel interference graph in [20] is derived from a register life-table which can be formulated as an interval graph, in general, it does not satisfy the interval graph properties any more. This is due to the fact that the parallel interference graph contains additional edges to model additional constraints. So even for homogeneous register sets, coloring of the parallel interference graph can be an $\mathcal{NP}$-hard problem.

Another approach using strategy 2 (Section 1) is discussed in [21]. The existence of a register assignment is guaranteed by performing a worst case analysis with respect to overlapping register lifetimes. Therefore, the binding of operands to orthogonal register subsets (register files) has to be fixed a-priori. To meet the worst case of overlapping register lifetimes, the schedule search space is reduced by serializing proper value lifetimes. While this reduces the complexity of the scheduling problem, it also eliminates valid solutions in contrast to [20].

In [10], a heuristic algorithm is proposed that alternately applies strategies 1 and 2. Based on an ASAP (*as soon as possible*) schedule that takes only data dependencies into account, register allocation/assignment is performed together with rescheduling to meet all constraints. Violations of constraints are thereby resolved one step at a time.

The assignment problem for heterogeneous functional units has been addressed in [9]. The proposed approach separates the construction of a specific assignment from the decision problem "does an assignment exist at all?". The decision problem may be of less complexity compared to the assignment problem. The idea is to quickly verify schedules with respect to the existence of proper functional unit assignments. Actual assignments are only generated if they are known to exist.

We have presented a detailed analysis of this technique in [23], thereby extending it to model heterogeneous register sets. In [23] only the decision problem has been discussed, while in the following sections we show how to generate actual register assignments. The formulation as a graph-coloring problem, namely list-coloring, is also a more general approach. In Section 4.3 it becomes clear that the technique [9,23] is only sufficient if the interference graph becomes decomposed into *complete* subgraphs.[2] This is very unlikely when considering register assignment problems but may be true when considering the interference of functional units [9]. If each functional unit has a single-cycle latency then we can construct one interference graph per clock-cycle. All functional units that are active at a certain clock-cycle interfere with each other. So these interference graphs are complete. Moreover, functional units that are active at a certain clock-cycle do not interfere with functional units

[2] A graph is complete if each pair of vertices is joined by an edge.

that are active in any other clock-cycle (single-cycle latency). The overall interference graph becomes decomposed into complete subgraphs.

In our example in Fig. 4(b) we consider functional units with different latencies. The subgraph $\{1,2,8\}$ is not complete and therefore the approach [9] provides only a necessary condition. So we must revise [23] and state that the extension of [9] to model heterogeneous register sets, in general, defines a necessary but *not* sufficient condition for the existence of a proper register assignment. In contrast to [9,23], the technique presented in this article allows us to generate register assignments and functional unit assignments for a much more general class of interference graphs, namely the interval graphs. In particular, the interference graphs are *not* required to become decomposed into complete subgraphs.

4. Register assignment and graph coloring

Register assignment problems are typically solved by coloring an interference graph G using heuristics [5,8].

4.1. Interference graph

The interference graph basically models concurrent lifetimes of program variables. The graph $G(V,E)$ contains one vertex $v_k \in V$ for each program variable $k = 1,\ldots,N$ and two vertices are connected by an edge $(v_i, v_j) \in E$ if the variables are live at the same time. If only continuous live ranges are considered then the interference graph is an *interval graph*. So interval graphs model the intersection of continuous time intervals and for each interval graph there exists a corresponding interval representation as shown in Fig. 6. A more precise definition of interval graphs is found in [13, Chapter 8].

A counter-example is shown in Fig. 7. For the interference graph in Fig. 7(b) there exists no interval representation because interval 4 will always overlap with interval 2 (Fig. 7(a)) if it has to overlap with intervals 1 and 3. But there is no edge between vertices 4 and 2 in Fig. 7(b).

Assume a homogeneous set of L registers. If we assign one out of L colors to each vertex in G such that different colors are assigned to adjacent vertices then this proper *L-coloring* [13, p. 7] corresponds to a valid register assignment. The minimum number of registers that are required is given by the *chromatic number* of the interference graph. If the chromatic number exceeds the number of available registers, spill code has to be inserted. It is well known that for an interval graph G the chromatic number equals the size ω of its *maximum clique* [13, p. 6] which is the maximum number of program variables that are active at the same time. A proper coloring can be calculated for an interval graph with linear-time complexity $O(|V| + |E|)$ [13, Chapter 8]. Unfortunately, this only applies to homogeneous register sets. For heterogeneous register sets the problem is more complex.

When modeling architecture irregularities, two different approaches are possible. Continuing our example (Fig. 6) let us assume that three registers $\{R_A, R_B, R_C\}$ are available. For each program variable the admissible registers are shown in Fig. 8. The register assignment problem can be mapped onto two different graph-coloring problems as shown in Fig. 9. Typically, the approach in Fig. 9(b) is proposed. Here the irregularities of the architecture are taken into account by modifying the interference graph. An

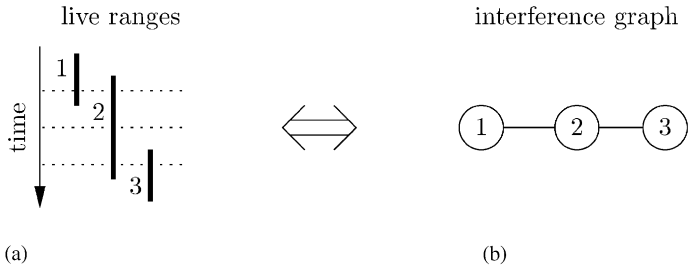


Fig. 6. Interference graph is an interval graph: (a) interval representation and (b) interval graph.

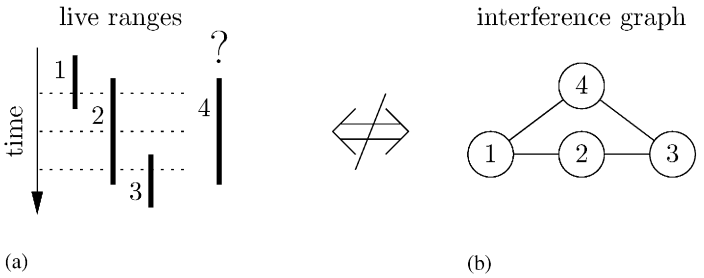


Fig. 7. Interference graph is *not* an interval graph: (a) no interval representation and (b) no interval graph.

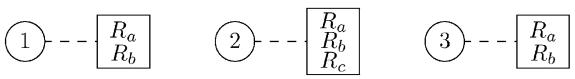


Fig. 8. Heterogeneous register-set architecture.

additional (pre-colored) vertex is added to the graph to guarantee that register R_c is not assigned to the variables 1 and 2. The resulting interference graph is no interval graph any more and the coloring problem becomes $\mathcal{NP}$-complete.

We propose the approach in Fig. 9(a). The interference graph is always guaranteed to be an interval graph, and architecture constraints are mapped to a *list-coloring* problem. In contrast to the L-coloring problem in case of homogeneous register sets, list-coloring is $\mathcal{NP}$-complete even for interval graphs. Of course, the complexity of the problem, in general, is not reduced by an alternative formulation. Nevertheless, in the following section a detailed analysis of the list-coloring problem shows the benefit of this approach.

4.2. List-coloring

Considering a list-coloring problem [12], the admissible colors for each vertex $v \in V$ are given by the set (color list) S_v. The overall number of colors available is $L = |\bigcup_{v\in V} S_v|$. In contrast to the L-coloring problem, now each vertex v has to be colored with one color out of its associated color list S_v. Different colors are assigned to adjacent vertices. So a proper list-coloring corresponds to a proper register assignment for heterogeneous register-set architectures.

The number of list-colorings to be investigated is bounded by

$$C_1 = \prod_{v\in V} |S_v| \leqslant L^{|V|} \tag{1}$$

if no further assumptions about the structure of the graph are made. From (1) follows that an exhaustive analysis of the search space is only tractable for a small number of program variables $|V|$.

But for an interference graph being an interval graph this bound is drastically reduced if we assume a given maximum number ω of concurrently active variables (Section 4.3)

$$C_2 < |V|L^{\omega}. \tag{2}$$

This means, if ω is considered a constant then the complexity is linear with respect to the number of program variables. Of course, ω may get as large as $|V|$ and the problem is still $\mathcal{NP}$-complete. But this worst case would correspond to a program that produces all intermediate results before consuming any of them. So for typical programs, ω will not exceed a reasonable maximum and therefore an optimum search strategy is applicable.

4.3. List-coloring and interval graphs

To show that the search space complexity is bounded by (2), we consider an order s on the vertices $v \in V$ where $s(l)$ is the lth vertex that is colored and $l = 1,\ldots,N$. Depending on the order s in which the vertices are colored a great number of colorings for the first l' $(1 < l' < N)$ vertices $s(1),\ldots,s(l')$ may be equivalent when coloring the remaining vertices $s(l'+1),\ldots,s(N)$.

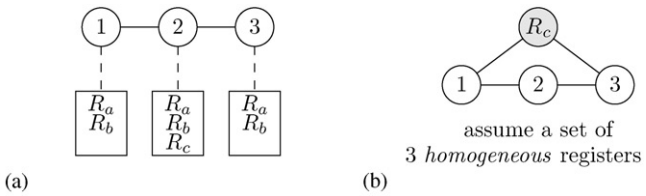


Fig. 9. Two different graph coloring problems: (a) list-coloring of an interval graph and (b) 3-coloring of a general graph.

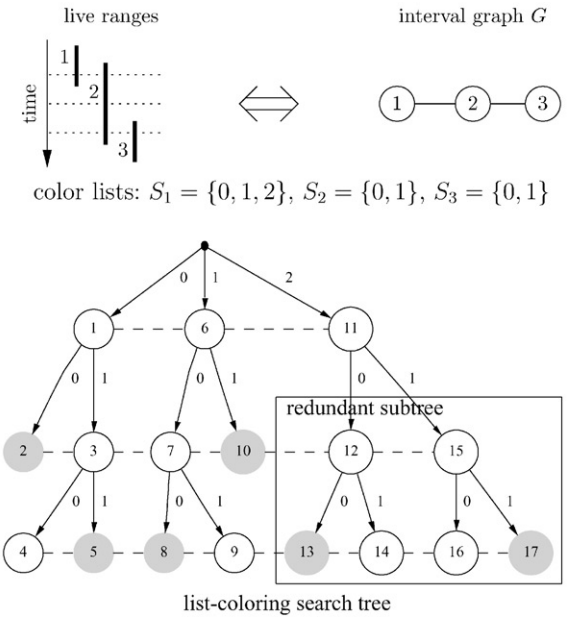


Fig. 10. List-coloring of a three-vertex interval graph.

To see this, consider the simple example in Fig. 10. We assume that three registers (r_0, r_1, r_2) are available and the vertices are colored in the order $s = [1,2,3]$ as shown in the list-coloring search tree. In this tree each path from the root to a leaf at the lowest level corresponds to a list-coloring of G.

At each level (indicated by dashed lines) a color has been assigned to a vertex of G. The edges in the search tree are labeled by the assigned colors. The vertex numbers in the search tree define the sequence in which the search tree is processed. Filled vertices indicate invalid colorings. The search tree path $1,2$ assigns color "0" to both vertices v_1 and v_2 of G. As these vertices are adjacent the coloring is invalid. Due to invalid partial colorings certain subtrees can be pruned from the search space.

Only four out of $\prod_v |S_v| = 12$ colorings are proper list-colorings. When processing the search tree they do not have to be considered each. At the search tree vertex 12 two vertices of the interval graph G have already been assigned a color ($v_1 \to 2$ and $v_2 \to 0$). For the coloring of v_3 this is equivalent to search tree vertex 7 ($v_1 \to 1$ and $v_2 \to 0$) because coloring v_3 only depends directly on the color assigned to v_2. The possible colorings for v_3 have already been determined in the subtree beneath search tree vertex 7. Accordingly, search tree vertex 15 is equivalent to search tree vertex 3. The whole subtree beneath search tree vertex 11 is redundant.

So there are two mechanisms which reduce the search tree complexity, namely, the pruning of invalid subtrees and the recognition of redundant subtrees. While invalid subtrees can hardly be quantified before actually assigning colors, the pruning of redundant subtrees may lead to significant decrease of search space complexity.[3] The existence of redundant subtrees clearly depends on the *order* in which vertices are colored. Assume that the vertices in example Fig. 10 are colored in order $s=[1,3,2]$. In this case the admissible colors for vertex v_2 depend on the chosen colors for vertices v_1 and v_3.

The existence of such an order is a property of the interval graph G. To see this, we consider the *maximal cliques* of G. Note the difference between a *maximum* clique which is the largest complete subgraph of G and a *maximal* clique which is not contained in any other clique [13, p. 6]. In our example in Fig. 10 the graph G

[3] A heuristic to reduce the width of the search tree is found in [16].

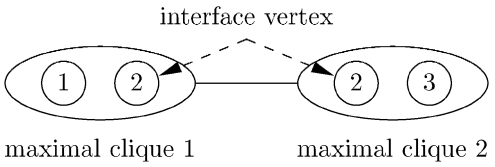


Fig. 11. Maximal cliques and interface vertices.

has two maximal cliques $\{1,2\}$ and $\{2,3\}$. From graph theory it is known if G is an *interval graph*, the maximal cliques of G can be linearly ordered such that, for every vertex x of G, the maximal cliques containing x occur consecutively. This can also be seen directly in the live range diagram from which the interval graph is derived (Fig. 10). The live ranges are always *continuous* so the ordering discussed above becomes obvious. This also shows what kind of restriction is not covered by our approach, namely, *discontinuous* live ranges which correspond to program variables that have to be placed into the same register.[4] Another representation for this property is given by the clique matrix. The maximal cliques-versus-vertices incidence matrix of an undirected graph G is called the *clique matrix* if all the maximal cliques are included. A matrix whose entries are zeros and ones is said to have the *consecutive ones property for columns* if its rows can be permuted in such a way that the ones in each column occur consecutively. Clique matrices of interval graphs have the consecutive ones property for columns [13, p. 174].

Let $\{X_i\}_{i\in I}$ be the collection of all subsets of V that induce a maximal clique of G. Let the maximal cliques be ordered according to the consecutive one's property $X_1, X_2, \ldots, X_{|I|}$. Vertices in X_i that are also contained in X_{i+1} are called *interface vertices*. All proper colorings of vertices in X_i do not have to be considered when coloring vertices in successor cliques X_j, $j > i$. Only proper colorings of the interface vertices are relevant (Fig. 11). The consecutive one's property guarantees that vertices which are *not* interface vertices do not interfere with any vertex in succeeding cliques. Note that interface vertices are not taken into account by the approaches [9,23], and therefore these methods

[4] More generally speaking, the color lists for each vertex of the interval graph must not depend on the color that is assigned to any vertex in the graph.

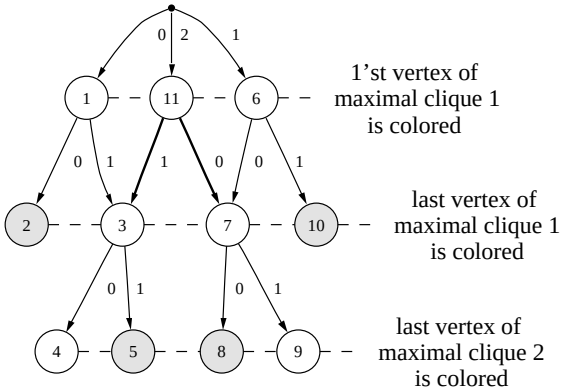


Fig. 12. Reduced list-coloring search tree.

are only sufficient, if there are no interface vertices at all (compare Section 3). The resulting search tree for the list-coloring problem in our example is shown in Fig. 12.

By exploiting the specific properties of interval graphs, the maximum number of colorings that have to be investigated can be drastically reduced. The complexity of the list-coloring procedure depends on the number of maximal cliques $|I|$, clique sizes $|X_i|$, and on the color set size $|S_v|$ of each vertex $v \in V$. So as stated in (2), an upper bound for the colorings to be investigated is given by

$$C_2 = \sum_{i\in I} \prod_{v\in X_i} |S_v| < |V| L^{\omega} \quad \text{with}$$

$$L = \left| \bigcup_{v\in V} S_v \right|, \quad \omega = \max_i |X_i|. \tag{3}$$

To summarize these results, the complexity of list-coloring an interval graph is mainly determined by the size ω of its maximum clique. So the complexity of generating a proper register assignment for heterogeneous register-set architectures is mainly dependent on the maximum number of program variables that are live at the same time. Considering optimum search strategies, the schematic search tree of the list-coloring problem for interval graphs is shown in Fig. 13.

4.4. Related topics of graph theory

Note that this is an application of the general concept called tree- or path-decomposition. A

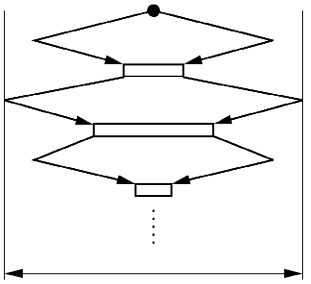


Fig. 13. Search tree corresponding to the list-coloring problem for interval graphs.

path-decomposition of a graph $G(V,E)$ is a sequence of subsets of vertices $(X_1,X_2,\ldots,X_r)$, such that

- $\bigcup_{1\leqslant i\leqslant r} X_i = V$;
- for all edges $(v,w)\in E$, there exists an i, $1\leqslant i\leqslant r$, with $v\in X_i$ and $w\in X_i$;
- for all $i,j,k\in[1\ldots r]$, if $i\leqslant j\leqslant k$, then $X_i\cap X_k\in X_j$.

The pathwidth of a given path-decomposition $(X_1,X_2,\ldots,X_r)$ is $\max_{1\leqslant i\leqslant r}\{|X_i|-1\}$. The ordering of the maximal cliques according to the consecutive one's property is in fact a path-decomposition of G with pathwidth $\omega-1$. Given a tree-/path-decomposition with its tree-/pathwidth bounded by a constant many problems that are hard for arbitrary graphs can be solved in polynomial or even linear time. For interval graphs with maximum clique size ω there exists a path-decomposition of width $\omega-1$. For a detailed discussion of tree- and path-decompositions see [2,3]. Application of these concepts to list-coloring problems are found in [14,15].

4.5. Further reduction of complexity

Considering the factor L^{ω} in (3), we see that the complexity is also reduced if L can be decreased. This can be achieved by considering equivalent registers. We define a register class to be a group of registers that can always be used equivalently. So a register class may for instance correspond to a register file. The number of registers in a certain class is denoted as the cardinality of this class.

Table 1
Color sets S_v

	S_0	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9
r_0	1	0	0	0	1	1	1	1	0	1
r_1	1	1	1	1	1	1	1	1	1	0
r_2	1	0	1	1	1	0	1	1	1	1
r_3	1	1	0	1	1	0	1	1	0	1

"1" in row i and column j means that $r_i\in S_j$.

If it is possible to group registers into classes then L is reduced to the number of different register classes. Using this concept we get a unified approach to register assignment by graph coloring. The case of homogeneous registers is now a special case of our proposed coloring approach. [5] In this case the number of register classes is 1 and the cardinality of this class is simply the number of available registers. Bound (3) gives linear complexity as expected.

List-coloring with colors of higher cardinality is done in two steps. First register classes are assigned to program variables. So register classes now define the set of available colors.

After the first step the graph is separated into subgraphs of equally colored vertices. To actually assign a specific register, the subgraphs may now be colored as in the case of homogeneous register sets.

5. Example

Let us present our proposed approach step by step using the more complex example shown in Fig. 14. We assume that four registers $r_0,\ldots,r_3$ are available and the color sets for each vertex of the interval graph in Fig. 14 are given in Table 1.

Step 0: A necessary condition [6] for the existence of a proper list-coloring without introducing spill code is given by $L\geqslant\omega$. For the example in Fig. 14 the number of registers is $L=4$ and $\omega=3$. So this necessary condition is met. [7]

[5] Otherwise the (simple) homogeneous case would have been the most complex when using the list-coloring bound (3).

[6] This condition is also sufficient in case of homogeneous register sets.

[7] When this condition is not satisfied we know that no proper register assignment exists. As discussed in Section 1, spill code has to be generated or the register-set architecture has to be modified.

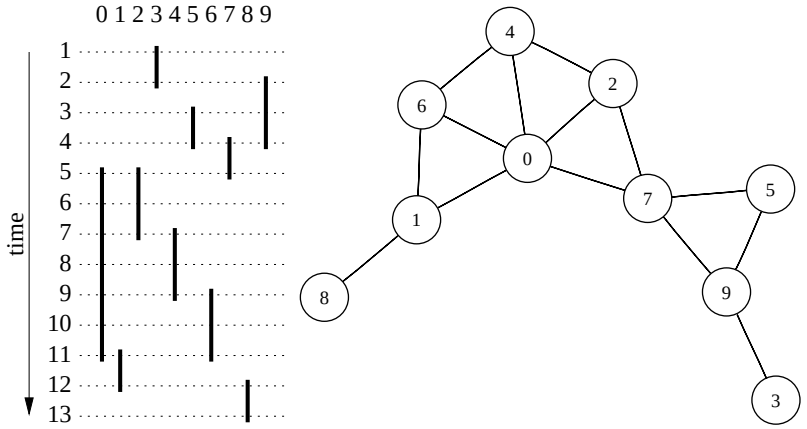


Fig. 14. Live ranges of program variables and corresponding interval graph G.

Table 2
Clique matrix – consecutive one's property

	0	1	2	3	4	5	6	7	8	9
$\{3,9\}$	0	0	0	*1*	0	0	0	0	0	*1*
$\{5,7,9\}$	0	0	0	0	0	*1*	0	*1*	0	*1*
$\{0,2,7\}$	*1*	0	*1*	0	0	0	0	*1*	0	0
$\{0,2,4\}$	*1*	0	*1*	0	*1*	0	0	0	0	0
$\{0,4,6\}$	*1*	0	0	0	*1*	0	*1*	0	0	0
$\{0,1,6\}$	*1*	*1*	0	0	0	0	*1*	0	0	0
$\{1,8\}$	0	*1*	0	0	0	0	0	0	*1*	0

Step 1: All maximal cliques of G have to be constructed. This can be done with linear complexity for interval graphs [13, Chapter 4.7]. For our example we find the maximal cliques: $\{3,9\}$, $\{1,8\}$, $\{5,7,9\}$, $\{0,1,6\}$, $\{0,4,6\}$, $\{0,2,7\}$, $\{0,2,4\}$.

Step 2: The maximal cliques have to be ordered according to the consecutive one's property. Again this can be achieved with linear complexity [4]. The properly ordered clique matrix for the example interval graph in Fig. 14 is given by Table 2. This order also becomes obvious when considering the live ranges in Fig. 14. The maximal cliques can be identified at time-slots $\{2,4,5,7,9,11,12\}$ and so the given order (Table 2) corresponds to the "natural" order defined by increasing time-slots.

Step 3: Choose an ordering s of all vertices $v \in V$ such that the maximal cliques are processed in the order found in step 2. Note that the simplest way to construct such a sequence of vertices is the natural order resulting from sorting vertices according to the start times of their corresponding time intervals. In the example we get the sequence $s = [3,9,5,7,0,2,4,6,1,8]$. In Fig. 15 the computation of coloring sequence s is demonstrated by using a path-decomposition of G as discussed in Section 4.4.

Table 3
List-coloring – a valid solution

Vertex	0	1	2	3	4	5	6	7	8	9
Register	r_0	r_3	r_1	r_1	r_2	r_1	r_1	r_2	r_1	r_0

Step 4: Given an order s on the vertices $v \in V$ where $s(l)$ is the lth vertex that is colored and $l = 1,\ldots,N$ together with the set of color lists $S = \{S_1, S_2, \ldots, S_N\}$ a proper list-coloring of G can be generated.

This is achieved by Algorithm LC (see below) which processes the *whole* search tree of the list-coloring problem as discussed in Section 4.3. The recursive algorithm is started for $l = 1$ and traverses the search tree in a *depth-first search* manner. Since we are typically interested in only one proper list-coloring the search may be terminated as soon as a proper list-coloring is found. In our example the first solution (Table 3) is already found after $|V| = 10$ iterations. [8]

[8] There also exist solutions that require only three registers but minimizing the number of registers is not subject to the considered decision problem (Section 1).

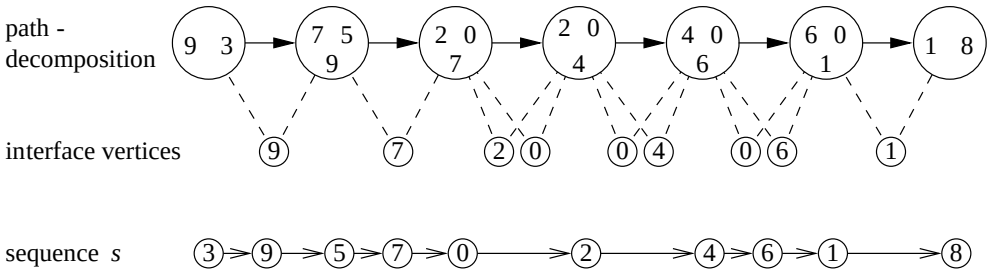


Fig. 15. Path-decomposition, interface vertices, and coloring sequence s.

Computational complexity: The upper bound (1) shows that at maximum $C_1 = 36864$ colorings have to be considered. For the proposed approach the upper bound (3) gives at maximum $C_2 = 197$ colorings to consider.

So the proposed approach reduces the complexity by orders of magnitudes. The pruning ratio is $\frac{197}{36864} \approx 5 \times 10^{-3}$.

6. Experimental results

We have presented an optimum technique for the $\mathcal{NP}$-complete list-coloring problem (Section 4.2). Our technique may successfully be applied to a great number of register assignment problems. The coloring problem is still $\mathcal{NP}$-complete so we know that certain register assignment problems will be intractable. Presenting "typical" real-world examples is therefore not a trivial task and may be misleading. The complexity of our approach is determined by a number of parameters. The most critical parameter is the size of the maximum clique or with respect to register assignment the number of program variables that are concurrently active. Also the number of register classes, the overall number of registers, and the number of program variables affect complexity. In this section we present different combinations of these parameters to investigate the limits of our approach.

We have applied the proposed approach to a number of different interval graphs. To analyze the essential parameters independently we generated these graphs synthetically. Based on the number of vertices and the intended size of the maximum clique, the execution time of a hypothetical program is estimated. The start times for live ranges of program variables are equally distributed inside this time interval. To generate typical interval graphs, the lengths of these live ranges are distributed according to a triangulated probability density function. Basically, this means that shorter live ranges are more likely than very long live ranges.

Algorithm LC list_coloring(l, S)

```
 1: if l > |V| then
 2:   return {stop recursion}
 3: end if
 4: v ← s(l)
 5: S' ← S {store color lists}
 6: for all colors c ∈ S'_v do
 7:   f(v) = c {assign v the color c}
 8:   S ← S' {restore color lists}
 9:   if v is the last vertex of a maximal clique then
10:     C ← set of colors of this clique's interface vertices
11:     if C has already been found for l then
12:       continue with next color c {skip subtrees}
13:     end if
14:   else
15:     save C for current l
16:   end if
17:   for all {S_u | u adjacent to v and s^-1(u) > s^-1(v)} do
18:     S_u = S_u \ {c} {remove c from S_u}
19:     if |S_u| = 0 then
20:       try next color c
21:     end if
22:   end for
23:   list_coloring(l + 1, S)
24: end for
```

Remark. When considering register classes as discussed in Section 4.5, S_u contains colors and their cardinalities. Removing color c from S_u (line 18) means decrementing the cardinality of c for vertex u. Color c is only removed from S_u when the cardinality reaches zero.

The color lists are also generated synthetically. Here we consider two parameters, namely the overall

Table 4
Experimental results

Interval graph	$\lvert V\rvert$	ω	L	$\prod_v \lvert S_v\rvert$	$\sum_i \prod_{v\in X_i} \lvert S_v\rvert$	Pruning ratio
1	10	4	6	3×10^{5}	1740	5.8×10^{-3}
2	50	4	6	2×10^{26}	5354	2.6×10^{-23}
3	200	4	6	3×10^{105}	20972	7.1×10^{-102}
4	400	4	6	7×10^{204}	36692	4.9×10^{-201}
5	10	6	10	5×10^{6}	52512	9.7×10^{-3}
6	50	6	10	7×10^{34}	591164	8.8×10^{-30}
7	100	6	10	1×10^{69}	1.8×10^{6}	1.6×10^{-63}
8	10	8	13	1×10^{8}	2.8×10^{6}	2.7×10^{-2}
9	50	8	13	2×10^{40}	1.3×10^{8}	6.0×10^{-33}
10	100	8	13	2×10^{78}	4.0×10^{8}	2.0×10^{-70}
11	50	10	16	2×10^{45}	2.7×10^{10}	1.0×10^{-35}
12	100	10	15	3×10^{92}	1.0×10^{11}	3.4×10^{-82}

number of registers and the number of register classes. Except for one example (interval graph 12, Table 4) the number of register classes equals the overall number of registers. Thus, we investigate the most complex case (Section 4.5).

The number of vertices, the maximum clique size, and the number of colors have been varied. Note that in the examples the whole search tree has been processed regardless of proper colorings already found. As the search is performed in a depth-first manner, problems of higher complexity may also be solvable depending on register pressure.

Processing the whole search tree is completed within less than one minute [9] for examples 1–8 and 12 where the complexity is further reduced by grouping the 15 registers in five classes (compare Section 4.5). In examples 9–11 the amount of memory required to hold the complete search tree exceeded a reasonable maximum. Nevertheless, a proper coloring has been found within seconds. So, in general, an exhaustive analysis of the search space is easily feasible for program widths of up to 8 for 13 register classes or for program widths of up to 10 for five register classes. Single solutions may also be found for problems of higher complexity.

The number of program variables is not a critical parameter with respect to complexity. Considering 100 or more program variables does not render our approach intractable. Note that such large basic blocks are not typical. But they may occur for instance when unrolling multi-rate systems, e.g. we analyzed basic blocks with 100 program variables and a maximum clique of size 7 for multi-rate wave digital filters that are implemented on an ASIP.

[9] Experiments are computed using a standard PC at 1.4 GHz.

7. Conclusions and future work

We have presented an optimum approach to solve register assignment problems that appear when generating code for heterogeneous register-set architectures. Register assignment is considered as coloring an interference graph. In contrast to traditional approaches, machine-related constraints are not mapped onto the structure of the graph but on a specific coloring problem. Thereby the corresponding interference graph is guaranteed to be an interval graph since it represents conflicting lifetimes only. Although list-coloring for interval graphs is still a complex combinatorial optimization problem, the special properties of the interval graph are exploited to develop a list-coloring algorithm that generates optimum solutions for real-world problems. Additionally, we have shown that this technique can also be applied to the similar problem of functional unit assignment.

The complexity analysis shows that the algorithm is capable of computing optimum solutions even for large basic blocks. The limiting factor with respect to complexity is either the maximum number of program

variables that are concurrently active when considering the register assignment problem or the maximum number of parallel operations when considering the functional unit assignment problem. In case of increasing program width, heuristics can be used to color or spill certain vertices in order to reduce the chromatic number of the interference graph. The heuristically reduced problem can be solved exactly. This combination with standard coloring heuristics is subject to future work.

References

[1] D.A. Berson, R. Gupta, M.L. Soffa, HARE: a hierarchical allocator for registers in multiple issue architectures, Technical Report TR 95-06, Computer Science Department, University of Pittsburgh, February 1995.

[2] H.L. Bodlaender, A tourist guide through treewidth, Acta Cybernet. 11 (1993) 1–21.

[3] H.L. Bodlaender, A partial k-arboretum of graphs with bounded treewidth, Theoret. Comput. Sci. 209 (1998) 1–45.

[4] K. Booth, G. Lueker, Testing for the consecutive ones property, interval graphs, and graph planarity using Pq-tree algorithms, J. Comput. System Sci. 13 (1976) 335–379.

[5] P. Briggs, K.D. Cooper, L. Torczon, Improvements to graph coloring register allocation, ACM Trans. Programming Languages Systems 16 (3) (1994) 428–455.

[6] D. Callahan, B. Koblenz, Register allocation via hierarchical graph coloring, in: Proceedings of the ACM Conference on Programming Language Design and Implementation, Toronto, Ontario, Canada, 1991, pp. 192–202.

[7] G.J. Chaitin, Register allocation & spilling via graph coloring, ACM SIGPLAN Notices 17 (6) (1982) 98–105.

[8] G.J. Chaitin, M.A. Auslander, A.K. Chandra, J. Cocke, M.E. Hopkins, P.W. Markstein, Register allocation via coloring, Comput. Languages 6 (1981) 47–57.

[9] Z. Chamski, C. Eisenbeis, E. Rohou, Flexible issue slot assignment for VLIW architectures, Fourth International Workshop on Software and Compilers for Embedded Systems, Schloss Rheinfels, St. Goar, Germany, September 1999.

[10] W.-K. Cheng, Y.-L. Lin, Code generation of nested loops for DSP processors with heterogeneous registers and structural pipelining, ACM Trans. Design Automat. Electron. Systems 4 (3) (1999) 231–256.

[11] F.C. Chow, J.L. Hennessy, Register allocation by priority-based coloring, ACM SIGPLAN Notices 19 (6) (1984) 222–232.

[12] R. Diestel, Graph Theory, Springer, Berlin, 2000.

[13] M.C. Golumbic, Algorithmic Graph Theory and Perfect Graphs, Academic Press, New York, 1980.

[14] S. Gravier, D. Kobler, W. Kubiak, Complexity of list coloring problems with a fixed total number of colors, Discrete Appl. Math. 117 (2002) 65–79.

[15] K. Jansen, P. Scheffler, Generalized coloring for tree-like graphs, Discrete Appl. Math. 75 (1997) 135–155.

[16] D.J. Kolson, A. Nicolau, N. Dutt, K. Kennedy, Optimal register assignment to loops for embedded code generation, Technical Report 95-46, University of California, Irvine, July, 1995.

[17] D. Landskov, S. Davidson, B. Shriver, P.W. Mallett, Local microcode compaction techniques, ACM Comput. Surveys 12 (3) (1980) 261–294.

[18] E.A. Lee, D.G. Messerschmitt, Static scheduling of synchronous data flow programs for digital signal processing, IEEE Trans. Comput. C-36 (1) (1987) 24–35.

[19] E.A. Lee, D.G. Messerschmitt, Synchronous data flow, Proc. IEEE 75 (9) (1987) 1235–1245.

[20] P. Pinter, Register allocation with instruction scheduling: a new approach, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, Albuquerque, 1993, pp. 248–257.

[21] K. van Eijk, B. Mesman, C.A.A. Pinto, Q. Zhao, M. Bekooij, J. van Meerbergen, J. Jess, Constraint analysis for code generation: basic techniques and applications in FACTS, ACM Trans. Design Automat. Electron. Systems 5 (4) (2000) 774–793.

[22] T. Zeitlhofer, B. Wess, Operation scheduling for parallel functional units using genetic algorithms, in: Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, Phoenix, AZ, USA, 1999.

[23] T. Zeitlhofer, B. Wess, Integrated scheduling and register assignment for VLIW-DSP architectures, in: Proceedings of the 14th IEEE International ASIC/SOC Conference, Washington, DC, USA, 2001, 339–343.