Efficient Scheduling of DSP Code on Processors with Distributed Register Files

Bart Mesman [†‡] Carlos A. Alba Pinto [‡] Koen van Eijk [‡]

[†] Philips Research Laboratories, WAY4, Prof. Holstlaan 4, 5656 AA Eindhoven, The Netherlands
[‡] Department of Electrical Engineering, Eindhoven University of Technology, The Netherlands

Abstract

Code generation methods for digital signal processors are increasingly hampered by the combination of tight timing constraints imposed by the algorithms and the limited capacity of the available register files. Traditional methods that schedule spill code to satisfy storage capacity have difficulty satisfying the timing constraints. The method presented in this paper analyses the combination of limited register file capacity, resource- and timing constraints during scheduling. Value lifetimes are serialized until all capacity constraints are guaranteed to be satisfied after scheduling. Experiments in the FACTS *environment show that we efficiently obtain high quality instruction schedules for innermost loops of DSP algorithms.*

1. Introduction

The exponential growth in the number of gates that can be integrated on a single chip has made the subject of embedded systems the central focus of many design and research groups. Next to commonly used microprocessors such as MIPS and ARM, embedded digital signal processors (DSPs) comprise the performance backbone for application domains such as communication and multimedia. There are roughly two communities that attempt to design embedded DSP processors: the people involved in application domain specific instruction set processors (ASIPs) [9] and retargetable compilers, and the companies that make stand-alone DSPs such as TI and Motorola. Both communities have their specific 'cultural heritage':

ASIPs, primarily driven by criteria as code size, power dissipation, and performance, often embed architectures with highly irregular data-paths and relatively few registers. Far from an ideal compiler target, 800% overhead in schedule length and code size is not exceptional for ASIP compilers [12]. Clearly there is a need for compilation methods that can deal with such irregular data-paths, and especially with a *limited* number of *distributed* registers.

For stand-alone DSPs, performance and high-level programmability are considered most relevant. For compiling high-level code, a relatively simple machine model is essential, preferrably comprising an orthogonal instruction set and a single large register file [1] that takes care of all communication between functional units. Power dissipation, code size, and required clock frequencies have made some DSP companies to *partition* this register file in a number of files [6] with a *limited* number of registers, and compilers will have to be able to cope with that.

In both cases, scheduling and register binding for *distributed, fixed register files* has to be performed such that potentially severe *timing* and *resource* constraints are satisfied. Traditional approaches deal with scheduling and register binding in separate stages to reduce the complexity of the problem. This introduces the problem of *phase coupling*: If register binding is performed before scheduling, it is difficult to satisfy tight timing constraints, while if it is performed after scheduling, there is not much freedom anymore to reduce register pressure. Therefore, although the separation of scheduling and register binding results in methods that are run-time efficient, it makes it much more difficult to cope with the interaction of timing, resource, and register file capacity constraints.

The accepted way to deal with fixed register files in a compiler is to do register spilling [3]: When the register pressure is too high, values are selected that are written to a (background) memory. The additional load and store operations require rescheduling and complicate satisfaction of the timing constraints. Many designers schedule time-critical code completely by hand. This requires extensive knowledge of the processor architecture and instruction set and is very time consuming.

In this paper, we present a new method to combine register binding and scheduling that reduces the problem of phase coupling. The general idea is to 'alternate' between the two sub-problems of scheduling and register binding by making a decision in the register binder and subsequently analysing how that prunes the search space for scheduling. This paper extends previous work [2] by also allowing

pipelined schedules.

Our method selectively serializes the lifetimes of values residing in overloaded register files until it can guarantee that any completion of the schedule will result in a feasible register binding for every register file. Therefore it does not unnecessarily reduce the freedom to also satisfy resource and timing constraints. Also, more schedule freedom will be sacrificed for register files that are more severely constrained (either by a small capacity or by a large number of assigned values).

This paper is organized as follows. In Section 2 we will start with some basic definitions and assumptions. In Section 3 the problem statement is given, and the global solution strategy is proposed consisting of bottleneck identification and lifetime serialization. These two components are explained in more detail in the subsequent sections. Section 5 shows some experimental results.

2. Definitions and assumptions

A DSP application can be expressed as a data flow graph (DFG) [7], which describes the primitive operations performed in the algorithm, and the dependencies between those operations.

Definition 1 (Data Flow Graph) *A data flow graph DFG is a triple (V,* $\mathrm{E_d} \cup \mathrm{E_s}$*, w), where*

- V *is the set of vertices (operations),*
- $\mathrm{E_d} \subseteq \mathrm{V} \times \mathrm{V}$ *is the set of data precedence edges,*
- $\mathrm{E_s} \subseteq \mathrm{V} \times \mathrm{V}$ *is the set of sequence precedence edges, and*
- $w : \mathrm{E_d} \cup \mathrm{E_s} \rightarrow \mathcal{Z}$ *is a function describing the timing delay associated with each precedence edge.*

For reasons of simplicity, we assume that all operations have an execution delay of 1 clock cycle. In [11] it is shown how pipelined and multi-cycle operations can be modeled using precedence constraints.

The task of scheduling is to assign each operation $v \in V$ a start time $s(v)$. These start times are constrained by the precedences. A precedence edge $(v_i, v_j) \in \mathrm{E_d} \cup \mathrm{E_s}$ states that $s(v_j) \geq s(v_i) + w(v_i, v_j)$. A chain of precedence edges $v_i \rightarrow v_k \rightarrow ...v_l \rightarrow v_j$ with total added weight d is called a *path*, implying $s(v_j) \geq s(v_i) + \mathrm{d}$.

Definition 2 *Distance. The distance* $\mathrm{d}(v_i, v_j)$ *from operation* v_i *to* v_j *is the length of the longest path from* v_i *to* v_j*.*

A path in the graph thus represents a minimum timing delay. These distances are stored in a *distance matrix*, which is calculated using a longest-path algorithm [4].

A schedule also has to satisfy the resource constraints. In our approach, these constraints are modeled by introducing functional resources and associating a certain resource usage with each operation [13]. We will not formalize these resource constraints, because the focus of this paper is on register binding. In our work we also consider *pipelined* schedules [8] that execute with a period called the *initiation interval* II, where $\mathrm{II} \leq \mathrm{l}$, the latency of the schedule.

3. Problem statement and global approach

In this section we will define our scheduling and register binding problem. We will decompose the problem and construct a block diagram of the global approach.

We assume a given binding of values to register files, which is often implied by the binding of operations to funtional units, and depends on the specific architecture of the data-path. The binding of values to specific registers remains and is the topic of this paper.

Problem Definition 1 *Constrained Register Binding and Operation Scheduling Problem. Given a cyclic data flow graph (DFG), the resource constraints, a binding of values to register files, for each register file* RF *a fixed capacity* $c(RF)$*, an initiation interval II, and a latency l. Find an assignment of values to registers and a schedule* s *that satisfy the precedence constraints, the resource constraints, the capacity constraint, and the timing constraints II and l.*

Because decisions have to be made that effect the feasible search space in both the domain of register binding and the domain of scheduling, we decompose the problem in separate phases as depicted in Figure 1.

The left part, the constraint analyzer, generates additional precedence constraints that are implied by the combination of all constraints, including the given register binding. These additional precedences refine the ASAP-ALAP intervals, thus providing a much more accurate estimate of the set of feasible start times. It consists of two different analysis methods: *execution interval analysis* [13] and *register constraint analysis* [10] and [11].

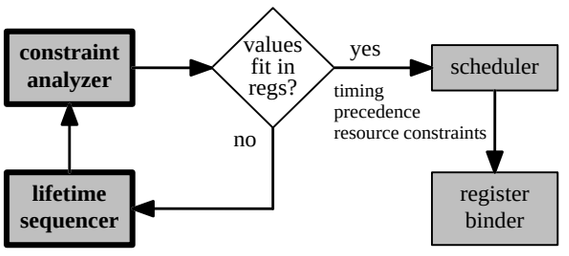


Figure 1. Global approach

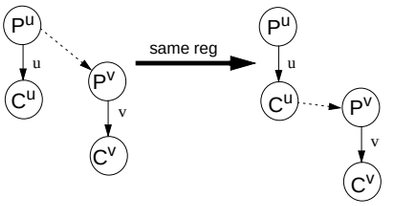


Figure 2. Value u must precede value v

The execution interval analysis considers precedence and resource constraints. It refines each operation's execution interval (the time interval in which the operation must start its execution) from the initial ASAP-ALAP interval by matching this interval to an interval in which the corresponding resources are available to this operation.

The register constraint analysis considers precedence, resource, and register binding constraints. It makes heavy use of the distance matrix, introduced in section 2, in the following way: Often the minimum delay between two operations is not only constrained by the precedences but also by resource or register constraints. A series of rules recognize the most occurring of these situations, and increase the minimum delay in the distance matrix correspondingly. These rules thus provide conditions (in terms of minimum distance) that are necessary for the feasibility of the register binding constraints. One such rule is graphically depicted in Figure 2. If values u and v must be allowed to use the same register and there is a precedence between the producers of u and v, then the lifetimes of u and v can only be serialized in the way indicated in Figure 2.

These analyses will guide the decisions made in the register binder and the scheduler and often prevent them from making decisions leading to infeasibility.

Based on the constraint analysis in Figure 1 an upper bound on the required number of registers is computed for a each register file (an exact figure is unknown because a complete schedule is not yet determined). When the upper-bound for each register file already respects the file's capacity, the additional precedences are transferred to a simple off-the-shelf scheduler and register binder for completion. However, in most cases and especially in the beginning of the process, the schedule freedom of the operations will be relatively large, resulting in many potential lifetime overlaps, thus inevitably violating some register file's capacity. In this case the lifetime sequencer in Figure 1, discussed in section 4, has to reduce the maximum number of overlapping values, by identifying one or more pair(s) of values that can be serialized. The constraint analyser subsequently calculates the effect of this serialization on the schedule freedom of all operations. This is necessary to prevent the lifetime sequencer (in subsequent iterations) from making serializations that are not possible. The lifetime sequencer and the constraint analyser alternate until the capacity of each register file matches the worst-case requirements.

An advantage of this new approach is that in practice a simple off-the-shelf scheduler and register binder can be used to complete the schedule. As the scheduler and its heuristics are not critical in this approach, we will not focus on them in this paper.

4. Lifetime sequencing

In this section we show how potential conflicts between pairs of values can be analysed before a complete schedule is known. This analysis uses the distance matrix to determine the 'worst case' lifetime overlap between values. These potential conflicts are used to identify and solve a bottleneck when some register file is in danger of being overloaded. The bottleneck identification is based on a coloring of a 'worst-case' *conflict graph*. Unlike traditional methods, our method does not require that all lifetimes are fixed when constructing the conflict graph.

4.1. Constructing a conflict graph

A conflict graph is an undirected graph CG(RF) = $(\mathrm{V}^c, \mathrm{E}^c)$, where the nodes in V^c represent the values in register file RF. There is an edge (u,v) $\in \mathrm{E}^c$ if the lifetimes of u and v overlap, and there is *no* edge (u,v) $\in \mathrm{E}^c$ if the lifetimes of u and v do *not* overlap. The triviality of the latter remark soon fades when we try to construct a conflict graph when the lifetimes are not fixed yet. Consider Figure 6 without folding (II $\geq l$); not two, but three different relations may exist between two values:

- There is no overlap. This is the case e.g. for values a and c. We say that a and c have *no conflict.*
- There is overlap for sure. This is the case e.g. for values a and b in the clock cycle that operation C executes. We say that a and b have a *strong* conflict.
- Unknown. This is the case e.g. for values b and e: if operation E precedes operation C by at least one clock cycle, b and e overlap. If not, b and e have no overlap. Since it is not yet determined whether or not E precedes C, it is simply unknown if b and e overlap. We say that b and e have a *weak* conflict.

For our purposes the following is the essential difference between a strong and a weak conflict: Values with a strong conflict can never reside in the same register, but values with a weak conflict can still be serialized. Serializing does have the drawback that schedule freedom is taken away: because

some distances increase (some paths get longer), the mobility of individual operations is affected. This is disadvantageous because intuitively it becomes 'harder' to find a feasible schedule. Therefore we want to select carefully which values to serialize, such that on the one hand, the number of weak conflicts in a potentially overloaded register file is reduced, and on the other hand, enough schedule freedom is left for processing subsequent overloaded register files. For this purpose it is convenient to have a clear criterion for each of the three possible relations between values u and v. Suppose that value u is produced by operation P^u and consumed by C^u, and value v is produced by operation P^v and consumed by C^v. In order to use the same rule both for the non-folded and the folded case, we also assume the following; There is always a latency constraint l, no matter how weak. For every A$\in$ V and B$\in$ V $\setminus$ sink, d(A,B) is restricted to $-l \leq \mathrm{d(A,B)} \leq \mathrm{l}-1$, and if there is no folding II is set at l.

4.1.1 No conflict

Values u and v have no conflict if their lifetimes can never overlap. There is no overlap between values u and v if and only if the lifetime of v is exactly contained in the interval in between two successive lifetimes of u. This is depicted graphically in Figure 3. This situation is recognized by the following lemma:

Lemma 1 *Values u and v have no conflict iff there exist iterations i and j such that $d(\mathrm{C}^u_i, \mathrm{P}^v_j) \geq 0$ and $d(\mathrm{C}^v_j, \mathrm{P}^u_{i+1}) \geq 0$.*

For the non-folded case we have either $i = j$ or $i = j+1$. This corresponds to the case that either C^u precedes P^v by at least 0 clock cycles (a value may be read and subsequently overwritten in the same clock cycle) or C^v precedes P^u by at least 0 clock cycles. Because we like to use the distance matrix for checking a conflict, we derive Lemma 2 which is equivalent to Lemma 1, as proven in the appendix.

Lemma 2 *Values u and v have no conflict iff*

$$\left\lfloor \frac{d(\mathrm{C}^u, \mathrm{P}^v)}{\mathrm{II}} \right\rfloor + \left\lfloor \frac{d(\mathrm{C}^v, \mathrm{P}^u)}{\mathrm{II}} \right\rfloor \geq -1 \qquad (1)$$

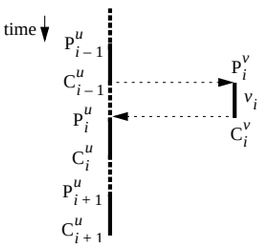


Figure 3. Values u and v have no conflict

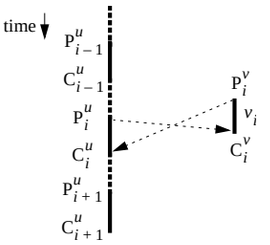


Figure 4. Values u and v have a strong conflict

4.1.2 Strong conflict

Values u and v have a strong conflict if their lifetimes overlap for sure. There is overlap between values u and v if and only if the lifetime of v can never be exactly contained in the interval inbetween two successive lifetimes of u. This is depicted graphically in Figure 4. This situation is recognized by the following lemma:

Lemma 3 *Values u and v have a strong conflict iff there exist iterations i and j such that $d(\mathrm{P}^u_i, \mathrm{C}^v_j) \geq 1$ and $d(\mathrm{P}^v_j, \mathrm{C}^u_i) \geq 1$.*

For the non-folded case we have $i = j$. This corresponds to the case where P^u precedes C^v by at least one clock cycle **and** P^v precedes C^u by at least one clock cycle. In Figure 6 for example, values a and b have a strong conflict, as depicted in Figure 5. Because we like to use the distance matrix for checking a conflict, we derive Lemma 4 which is equivalent to Lemma 3, as proven in the appendix.

Lemma 4 *Values u and v have a strong conflict iff*

$$\left\lfloor \frac{d(\mathrm{P}^u, \mathrm{C}^v) - 1}{\mathrm{II}} \right\rfloor + \left\lfloor \frac{d(\mathrm{P}^v, \mathrm{C}^u) - 1}{\mathrm{II}} \right\rfloor \geq 0 \qquad (2)$$

4.1.3 Weak conflict

There is weak overlap if the inequalities 2 and 1 are invalid. In Figure 6 for example, values a and e weakly overlap, as depicted in Figure 5.

4.2. Coloring and bottleneck identification

In the previous subsection we showed how to construct a conflict graph with three possible relations between values. In this subsection we use the conflict graph to identify two values u and v that should be serialized in order to reduce the potential overload on the corresponding register file.

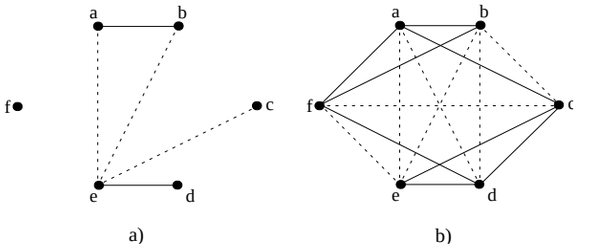
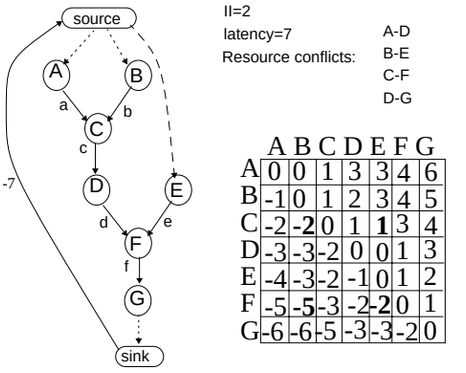


Figure 5. Conflict graph for Figure 6. A drawn edge means strong overlap. A dashed edge means weak overlap. All values are mapped on one register file. a) not folded b) II=2



	A	B	C	D	E	F	G
A	0	0	1	3	3	4	6
B	-1	0	1	2	3	4	5
C	-2	**-2**	0	1	**1**	3	4
D	-3	-3	-2	0	0	1	3
E	-4	-3	-2	-1	0	1	2
F	-5	**-5**	-3	-2	**-2**	0	1
G	-6	-6	-5	-3	-3	-2	0

Figure 6. Example of a precedence graph and the corresponding distance matrix after resource constraint analysis

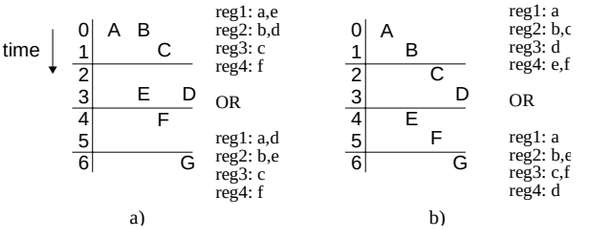


Figure 7. The only two feasible schedules for the example of Figure 6

We make two different conflict graphs: a weak conflict graph WCG, that includes both weak and strong conflicts, and a strong conflict graph SCG, that includes only strong conflicts. We color both conflict graphs by applying the exact sequential coloring algorithm from [5]. Coloring the weak conflict graph gives a 'worst case' coloring based on the worst case overlap. Coloring the strong conflict graph gives a 'best case' coloring based on the minimum overlap. From this coloring we extract for each node w in the conflict graph the *saturation number*, the number of different colors in the neighbourhood of w (the nodes connected to w in the conflict graph).

The choice of value u is based on the highest saturation number in WCG, the weak conflict graph. Since there may be a lot of values with the same saturation number in WCG (especially in the beginning of the search process, see Figure 5b), highest saturation number in the strong conflict graph is used as a second criterion. This yields a value u that is a bottleneck in the coloring.

The first criterion for choosing value v is also the highest saturation number in WCG, because we are only interested in potential bottlenecks. The second criterion however is the *lowest* saturation number in the strong conflict graph, SCG. The rationale behind this last criterion is that value v should be such that it has a lot of freedom to serialize with other values. The alternative is that both values u and v are fundamental bottlenecks, which would imply that two values with long lifetimes are pushed in the same register, which is clearly not a wise decision. Furthermore, we maintain the restriction that u and v have a weak conflict, because strong conflicts cannot be serialized, and values having no conflict need not to be serialized.

We will use the example in Figure 6 to illustrate the binding process. The distance matrix after resource constraint analysis is given in the same figure. It is used to construct the conflict graph in figure 5b). Because the worst case conflict graph is complete, the priority function will generate a choice of u and v which is as sensible as any other. Suppose that values b and e are chosen. The constraint analysis applies the rule shown in Figure 2 on the distance B $\rightarrow$ E of 3 (k=1) to serialize C $\rightarrow$ E with a delay of 2 clock cycles, and on the distance E $\rightarrow$ B of -3 (k=-2) to serialize F $\rightarrow$ B with a delay of -4 clock cycles. The effect is that the four numbers that are printed in bold in the distance matrix in Figure 6 are increased by one clock cycle, and the conflict graph is updated. Subsequently, values c and f are chosen. The constraint analysis now reduces the mobility to zero, so the schedule is fixed, as depicted in Figure 7b). There is also another schedule with the initial choice to serialize b and e, which explains the fact that there was still schedule freedom after this initial choice. For this particular example, the choice of which values to serialize is not very crucial.

4.3. Decisions in serialization

Suppose that the decision is made that values u and v are serialized. Let value u be produced by operation P^u and consumed by C^u, and let value v be produced by operation P^v and consumed by C^v. Serializing u and v can be done in two ways: $\mathrm{C}^u \rightarrow \mathrm{P}^v$ or $\mathrm{C}^v \rightarrow \mathrm{P}^u$. Often the constraints are such that the constraint analyzer is able to exclude one of

these possibilities. If this is not the case, a decision should be made. This decision is based on sacrificing the least possible schedule freedom; Let x=d(C^u, P^v) and y=d(C^v, P^u). Then $x \leq 0$ and $y \leq 0$, otherwise u and v would already have been serialized. Making a decision amounts to increasing either x or y to zero. The least schedule freedom is sacrificed when the smallest increase is made, so if x≥y we serialize $C^u \rightarrow P^v$, otherwise $C^v \rightarrow P^u$. If the constraint analyzer detects infeasibility as a result of this choice, the alternative serialization is taken. If that also yields infeasibility, extra sequence edges are introduced to explicitly model that u and v have a strong conflict, and another pair of values is chosen to repeat the process.

5. Results

In this section, we present the experimental results obtained with the proposed method. All experiments are run on a machine with a 233 MHz Pentium II processor.

The proposed techniques are especially intended to handle inner loops from DSP algorithms under tight timing constraints. As examples, we use the inner loop of a fast fourier transform (FFT) algorithm, a fast discrete cosine transform (FDCT) algorithm and a Loeffler algorithm that performs an 8-point 1-dimensional inverse discrete cosine transform. Each example is mapped to a relatively simple architecture in which each resource type has a dedicated register file. The characteristics of the various examples are shown in Table 1. The latency shown in the third column is the minimum latency obtainable for that constraint set. The table also shows the results obtained by a branch-and-bound scheduler [13] followed by a register binder based on exact minimum graph coloring. These results are used as a reference point for the method proposed in this paper.

To evaluate the proposed method, we have applied it to the examples of Table 1 with various register file capacity constraints. The branch-and-bound scheduler is used to complete the partial schedule resulting from value lifetime serialization. The results are shown in Table 2

For each problem instance, Table 2 lists the register file capacity constraints, the run time (including the time needed for scheduling), and the impact of serialization on the mobility of the operations (the numbers before respectively after the arrow denote the mobility before and after serialization). The mobility is defined as the average difference between the as late as possible (ALAP) start time and the as soon as possible (ASAP) start time of the operations: $\frac{1}{|V|}\sum_{v_i \in V} \mathrm{ALAP}(v_i) - \mathrm{ASAP}(v_i)$. Mobility is a good indication of the schedule freedom.

Table 1. Examples and reference results

example	$\|V\|$, $\|E_d\|$	II / lat.	time (s)	RF sizes
fft256	30, 43	4 / 13	0.1	3, 3, 1, 2
fdct	42, 43	18 / 18	0.1	9, 4
loef	56, 57	26 / 28	0.4	8, 4, 10

Table 2. Results of proposed method

example	RF caps	time (s)	mobility
fft256	1, 4, 1, 2	0.1	$0.7 \rightarrow 0.3$
	2, 2, 1, 2	0.4	$2.3 \rightarrow 0.0$
	2, 3, 1, 1	0.8	$2.1 \rightarrow 0.0$
	3, 2, 1, 1	0.9	$2.1 \rightarrow 0.0$
	4, 1, 1, 2	0.1	$0.7 \rightarrow 0.4$
fdct	9, 4	2.3	$9.5 \rightarrow 4.0$
	6, 4	2.7	$9.5 \rightarrow 2.0$
	8, 2	0.9	$9.5 \rightarrow 1.4$
loef	8, 4, 10	3.5	$14.4 \rightarrow 3.1$
	4, 3, 8	4.9	$14.4 \rightarrow 1.0$

The experimental results for the example fft256 clearly show that the proposed method is steered by the individual register file constraints; despite the presence of tight timing and resource constraints, the approach is able to generate many different schedules dependent on the settings of the individual capacity constraints. We consider this feature very important for handling heterogeneous register file architectures. By integrating the phases of scheduling and register binding our method is also able to significantly reduce the register pressure compared to an approach that performs register binding a posteriori. For the example 'loef', this results in a reduction from 22 to 15 in the total number of registers.

6. Conclusions and further research

In this paper, we presented a new approach for register binding and scheduling in the context of distributed register file architectures. Register file capacity constraints are taken into account during the first phase of scheduling, while there is still enough freedom to reduce register pressure. Constraint analysis techniques are used to capture the interaction between the precedence, timing and resource constraints. By constructing a conflict graph that models the strong and weak conflicts between values, the bottlenecks for register binding are identified. These bottlenecks are subsequently reduced by serializing value lifetimes. This results in a partial schedule that can be completed by a conventional scheduler without violating the register file capacities. Although we have not directly addressed the problem

of spilling values to background memory, we feel that the proposed method can help to avoid unnecessary spill code.

The results in section 5 show that our method is able to satisfy register file capacities under tight timing and resource constraints. The method provides a good balance between solution quality and run time efficiency.

A. Strong or no overlap

First we prove Lemma 2. Let P_i^u denote the i^{th} execution of P^u, the operation that produces value u. Then Let k be the largest value such that $d(\mathrm{C}^u, \mathrm{P}^v) \geq k \times \mathrm{II}$ and let l be the largest value such that $d(\mathrm{C}^v, \mathrm{P}^u) \geq l \times \mathrm{II}$. Because $s(\mathrm{A}_k) = s(\mathrm{A}_0) + k \times \mathrm{II}$ we have that $d(\mathrm{C}^u, \mathrm{P}^v) \geq k \times \mathrm{II}$ is equivalent to $d(\mathrm{C}_k^u, \mathrm{P}_0^v) \geq 0$, and that $d(\mathrm{C}^v, \mathrm{P}^u) \geq l \times \mathrm{II}$ is equivalent to $d(\mathrm{C}_0^v, \mathrm{P}_{k+1}^u) \geq (k+1+l) \times \mathrm{II}$. Now Lemma 1 applies if and only if $(k + 1 + l) \times \mathrm{II} \geq 0$. Because $\mathrm{II} \geq 0$ this condition is equivalent to $k + l \geq -1$. Now by definition $k = \left\lfloor \frac{d(\mathrm{C}^u, \mathrm{P}^v)}{\mathrm{II}} \right\rfloor$ and $l = \left\lfloor \frac{d(\mathrm{C}^v, \mathrm{P}^u)}{\mathrm{II}} \right\rfloor$, so equation 1 follows. Q.E.D.

Now we prove Lemma 4 on strong conflicts. Let k be the largest value such that $d(\mathrm{P}^u, \mathrm{C}^v) \geq 1 + k \times \mathrm{II}$ and let l be the largest value such that $d(\mathrm{P}^v, \mathrm{C}^u) \geq 1 + l \times \mathrm{II}$. Because $s(\mathrm{A}_k) = s(\mathrm{A}_0) + k \times \mathrm{II}$ we have that $d(\mathrm{P}^u, \mathrm{C}^v) \geq 1 + k \times \mathrm{II}$ is equivalent to $d(\mathrm{P}_k^u, \mathrm{C}_0^v) \geq 1$, and that $d(\mathrm{P}^v, \mathrm{C}^u) \geq 1 + l \times \mathrm{II}$ is equivalent to $d(\mathrm{P}_0^v, \mathrm{C}_k^u) \geq 1 + (k + l) \times \mathrm{II}$. Now Lemma 3 applies if and only if $(k + l) \times \mathrm{II} \geq 0$. Because $\mathrm{II} \geq 0$ this condition is equivalent to $k + l \geq 0$. Now by definition $k = \left\lfloor \frac{d(\mathrm{P}^u, \mathrm{C}^v) - 1}{\mathrm{II}} \right\rfloor$, and $l = \left\lfloor \frac{d(\mathrm{P}^v, \mathrm{C}^u) - 1}{\mathrm{II}} \right\rfloor$, so equation 2 follows. Q.E.D.

References

[1] *Trimedia TM-1 Media Processor Data Book*. Philips Semiconductors, Trimedia Product Group, 1997.

[2] C. A. Alba-Pinto, B. Mesman, and K. A. van Eijk. Register files constraint satisfaction during scheduling of dsp code. In *Symposium on Integrated Circuits and Systems Design*, Natal, Brazil, Oct. 1999.

[3] G. Chaitin. Register allocation and spilling via graph coloring. In *ACM Symposium on Compiler Construction*, pages 98–105, 1982.

[4] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to algorithms*. MIT Press, 1990.

[5] O. Coudert. Exact coloring for real-life graphs is easy. In *Proceedings of the 34th ACM/IEEE Design Automation Conference*. ACM and IEEE Computer Society, 1997.

[6] P. Faraboschi, G. Desoli, and J. Fisher. Clustered instruction-level parallel processors. Technical Report HPL-98-204, Hewlett-Packard, 1998.

[7] D. Ku and G. D. Micheli, editors. *High Level Synthesis of ASICs Under Timing and Synchronization Constraints.* Kluwer Academic Publisher, 1992.

[8] M. Lam. Software pipelining: An effective scheduling technique for vliw machines. In *SIGPLAN Conference on Programming Language Design and Implementation*, June 1988.

[9] R. Leupers, W. Schenk, and P. Marwedel. Microcode generation for flexible parallel architectures. In *Working Conference on Parallel Architectures and Compiler Technology*, 1994.

[10] B. Mesman, M. Strik, A. Timmer, J. van Meerbergen, and J. Jess. A constraint driven approach to loop pipelining and register binding. In *Proceedings of the Design Automation and Test in Europe*, Paris, 1998. IEEE Computer Society Press.

[11] B. Mesman, A. Timmer, J. van Meerbergen, and J. Jess. Constraint analysis for dsp code generation. *IEEE Transactions on Computer-Aided Design*, 18(1):44–57, Jan. 1999.

[12] P. Paulin and C. Liem. Embedded systems: Tools and trends, tutorial. In *Proceedings of the European Design and Test Conference*, Paris, Mar. 1996. IEEE Computer Society Press.

[13] A. Timmer, M. Strik, J. van Meerbergen, and J. Jess. Conflict modelling and instruction scheduling in code generation for in-house dsp cores. In *Proceedings of the 32nd ACM/IEEE Design Automation Conference*. ACM and IEEE Computer Society, 1994.