Scheduling Independent Liveness Analysis for Register Binding in High Level Synthesis

Vito Giovanni Castellana, Fabrizio Ferrandi

Politecnico di Milano – Dipartimento di Elettronica ed Informazione
Via Ponzio 34/5, 20133, Milan, Italy
{*vcastellana,ferrandi*}*@elet.polimi.it*

***Abstract*— Classical techniques for register allocation and binding require the definition of the program execution order, since a *partial ordering relation* between operations must be induced to perform liveness analysis through data-flow equations. In High Level Synthesis (HLS) flows this is commonly obtained through the scheduling task. However for some HLS approaches, such a relation can be difficult to be computed, or not statically computable at all, and adopting conventional register binding techniques, even when feasible, cannot guarantee maximum performances. To overcome these issues we introduce a novel scheduling-independent liveness analysis methodology, suitable for dynamic scheduling architectures. Such liveness analysis is exploited in register binding using standard graph coloring techniques, and unlike other approaches it avoids the insertion of structural dependencies, introduced to prevent run-time resource conflicts in dynamic scheduling environments. The absence of additional dependencies avoids performance degradation and makes parallelism exploitation independent from the register binding task, while on average not impacting on area, as shown through the experimental results.**

I. Introduction

Register allocation and binding are the tasks of mapping storage values into physical registers. Usually physical registers are allowed to store different storage values, provided that their *life intervals* never overlap. Liveness analysis provides such information, and it is commonly performed iteratively solving the *Data-Flow (DF) equations* [1], starting from a graph representation of the subprogram highlighting its execution order (e.g. State Transition Graphs). In HLS the operations execution order is determined through the scheduling task, which associates each operation to a control step. Nevertheless, for some synthesis approaches, such an ordering is not statically computed, making DF equations ineffective: we refer to them as *dynamic scheduling* approaches. In dynamic scheduling architectures, execution order varies at run-time, according for example to different input parameters or variable latencies of the hardware resources [2]. In these cases the allocation and binding tasks may impact on performance: they are addressed *assuming* a static operation ordering: run-time adjustments are *constrained* through the insertion of dependencies denoting resource conflicts. Thus safety is obtained through a potentially unnecessary serialization of operations, mitigating the available parallelism exploitation. On the other side, adopting conservative approaches as exclusively bind hardware resources may lead to non-negligible increase in area. For this reason we introduce a novel Liveness Analysis methodology that does not require a pre-computed schedule, making it suitable for any dynamic scheduling approach. The proposed analysis can be exploited in the register binding task, with the aim of maximizing the parallelism exploitation.

978-3-9815370-0-0/DATE13/©2013 EDAA

II. Related Work

Common HLS approaches acquire liveness information assuming a partial ordering relation between operations, obtained through a static schedule [3][4]. Liveness information is then captured in the form of a Conflict Graph, and register binding often addressed as a graph coloring problem [5]. We refer to such techniques, where register binding exploits the scheduling task results, as *post-scheduling* approaches. On the other side, in *pre-scheduling* approaches allocation is performed before scheduling: in this case minimizing the number of allocated registers may lead to the introduction of false dependencies in order to avoid resource conflicts. Despite the absence of a pre-computed schedule, proposed register binding methodologies for dynamic scheduling architectures are usually designed as *post-scheduling* approaches. Examples are given in [6] and [7], where the authors propose a distributed controller for managing Speculative Functional Units (SFUs) in HLS. Data-dependent operations will wait until the needed results are correctly computed. Register binding is performed through a left-edge based algorithm [8], assuming a static execution order: structural dependencies are introduced to achieve safety in the case of runtime adjustments to the initial schedule, due to wrong predictions. In order to reduce the impact of structural dependencies on execution latency, the binding has been customized adopting a Least Recently Used policy, which tries to bind different registers for close in time operations. However this approach cannot provide a conflict free register assignment. The binding approach based on the liveness analysis proposed in this paper instead, completely avoids any runtime resource conflict, associating operations that may be executed concurrently to different registers. A similar idea has been introduced in [9], where register allocation is performed through the coloring of a *parallelizable conflict graph* (PCG) which avoids the insertion of false dependencies. The PCG is built starting from a pre-computed conflict graph, adding edges between storage values defined by possibly concurrent operations. Such approach is heavily influenced by the initial conflict graph construction, which still requires a statically defined operations ordering. Obviously this requirement induces severe restrictions to the number of feasible schedules: violating the

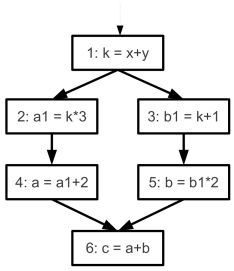

Fig. 1: *Example Dependencies Graph.*

previously selected ordering may lead to incorrect execution. On the contrary, the register binding based on the proposed methodology guarantees correctness without performance loss for *any possible* runtime schedule. In [10] the author propose a dynamic scheduling architecture supporting variable latency functional units. Focus is on parallelism exploitation: conflicts among registers are avoided adopting unique binding.

III. Preliminary Notions and Definitions

a) Flow Graphs: Denoting with V the set of vertices and with E the set of edges, a node $v \in V$ of a graph $G(V,E)$ has *out edges* that lead to *successor* nodes and *in-edges* that come from *predecessor* nodes. The set *out(v)* represents the set of *out edges*, while *in(v)* represents the set of *in edges*. Moreover we denote with $pred(v) \subset V$ the set of predecessors of node v and with $succ(v) \subset V$ the set of successors of v. Given $e \in E$, $source(e) \in V$ represents the source node of e, while $target(e) \in E$ represents the target node. A *directed path* $p = a \rightarrow b$ is a sequence of edges $e_0, e_1, ..., e_n$ such that $source(e_0) = a, source(e_1) = target(e_0)$, ... , $target(e_n) = b$.

b) Uses and Defs: Representing with *VAR* the set of variables in a subprogram P, and with $G(V,E)$ a graph representation of P: the set $def(x) \subset V$ of a variable $x \in VAR$ is the set of graph nodes that define it; the set $use(x) \subset V$ of a variable $x \in VAR$ is the set of graph nodes that use it; the set $def(v) \subset VAR$ of a graph node $v \in V$ is the set of variables that it defines; the set $use(v) \subset VAR$ of a graph node $v \in V$ is the set of variables that it uses.

c) Liveness Analysis and Data-Flow Equations: A variable $x \in VAR$ is said to be *alive on edge* $e \in E$ if $\exists$ a directed path p from e to a node $b \in use(x)$. A variable $x \in VAR$ is *live-in* at node $v \in V$ if $\exists e \in in(v)$ such that x is alive on e; similarly $x \in VAR$ is *live-out* at node $v \in V$ if $\exists e \in out(v)$ such that x is alive on e. *Live-in* and *live-out* sets capture liveness information useful to perform register allocation and binding. Such information is obtained from *use* and *def* through *Data-Flow equations*:

$$live\_in(v) = use(v) \cup (live\_out(v) \setminus def(v)) \quad (1)$$

$$live\_out(v) = \bigcup_{s \in succ(v)} live\_in(s) \quad (2)$$

Liveness analysis is performed iteratively solving the Data-Flow (DF) equations, until the least fixed point is reached [1].

(a)
Live-out(6) = { }
Live-in(6) = {a, b}
Live-out(5) = {a, b}
Live-in(5) = {a, b1}
Live-out(4) = {a, b}
Live-in(4) = {b, a1}
Live-out(3) = {a, b1}
Live-in(3) = {a, k}
Live-out(2) = {b, a1}
Live-in(2) = {b, k}
Live-out(1) = {k, a, b}
Live-in(1) = {x, y}

(b)
Live_p(6) = { }
Live_p(5) = {a, b, a1, k}
Live_p(4) = {a, b, b1, k}
Live_p(3) = {a, b, a1, k}
Live_p(2) = {a, b, b1, k}
Live_p(1) = { }

Fig. 2: *Liveness analysis results solving standard DF equations (a), $live_p$ sets(b), for the example DG in Figure 1.*

IV. Schedule-Independent Liveness Analysis

DF equations as formulated in (1) and (2) require a graph representation of the specification characterized by a single execution flow, which reflects a given execution order. For dynamic scheduling architectures a single flow representation will not adequately represent the run-time execution order, and standard liveness analysis will produce inaccurate information. However, even if a static schedule is not available, it is always possible to partially define execution order according to the dependencies between operations. If operation a depends on operation b, then a must be executed after b. A convenient way to represent such relations is a Dependencies Graph (DG), capturing both control and data dependencies. In the following, the DG is assumed to embed also flow information (e.g. loop back-edges). If there exist a path $p = x \rightarrow y$ between two nodes x and y of the DG, then y will be executed after x. Such a relation is not defined for all the possible pairs of nodes: if such a path does not exist, then it is not possible to state, simply analyzing the DG, which operation will be executed first; obviously in this case it is also not possible to establish if the two operations will be executed concurrently. In Figure 2(a) the results obtained through DF liveness analysis on an example DG (Figure 1) are shown. Analyzing the live-in/live-out sets it results that variables $a1$ and $b1$ are never simultaneously alive. According to this result, these variables may be mapped on the same physical resource, since their life intervals do not overlap, not ensuring correctness for all the possible execution orders. For example, if operations 2 and 3 complete their execution in the same control step, then $a1$ and $b1$ cannot be stored in the same physical register. Notice that for nodes 2 and 3 it is not possible to compute an order relation. Such nodes can be denoted as *parallel* nodes because they may be executed concurrently. It is possible to define a binary relation $||$ among the nodes of the DG: given two nodes a and b, $a||b$ iff there is not a path $p = a \rightarrow b$ or $p = b \rightarrow a$, i.e. they are *parallel*. In order to guarantee correctness in the presence of parallel nodes two sets are defined. Denoting with

$$parallel(x) = \{y | \nexists \ a \ path \ p = x \rightarrow y \text{ and } \nexists \text{ a path } p = y \rightarrow x\} \quad (3)$$

the set of *parallel* nodes with respect to node x, we define:

$$live_p(n) = \bigcup_{s \in parallel(n)} live\_in(s) \cup live\_out(s)$$

Sets $live_p(n)$ are introduced in order to take in account interferences between variables alive at *parallel* nodes. The main reason for keeping these sets separated from $live\_in(n)$

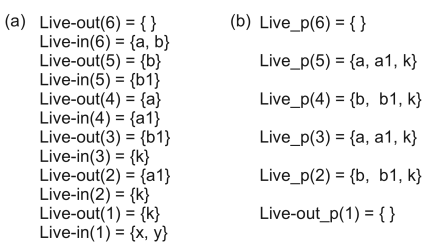

Fig. 3: *Proposed liveness analysis results for the example DG in Figure 1.*

and $live\_out(n)$ is that this will facilitate the construction of the *conflict graph* representing the interferences between storage values. The introduced sets are computed when the DF equations (as in (1) and (2)) have been already solved. Figure 2(b) reports $live_p$ sets for the previous example. However, there is another issue to be considered, concerning *death* of variables. In the proposed example, variable k results *dead* on exit on both nodes 2 and 3, since there are no uses of k reachable from these nodes. Nevertheless, k cannot be considered dead until both 2 and 3 are completed; the equation defining $live_p$ is modified in order to consider this aspect:

$$live_p(n) = \bigcup_{s \in parallel(n)} live\_in(s) \cup live\_out(s) \cup dead(s,n) \quad (4)$$

where

$$dead(s,n) = \{x | \nexists \text{ a path } n \rightarrow u, u \in use(x), x \in use(s)\},$$

i.e. $dead(s,n)$ is the subset of $use(s)$ of variables that are not live out at s, but are used on a parallel node. Applying these equations, the sets $live_p$ for nodes 2 and 3 now include variable k. Even if the computation of $live_p$ sets allows a conservative construction of a conflict graph, the DF equations, in their standard formulation, produce over-conservative sets. For instance, in the considered example variables a and b results alive on exit at node 1, even if they have not already been defined. To avoid this issue, DF equations can be modified as follows:

$$live\_in(n) = use(n) \cup (out(n) \setminus def(n)) \quad (5)$$

$$live\_out(n) = \bigcup_{s \in succ(n)} in'(s) \ , \ in'(s) \subset in(s) \quad (6)$$

where

$$in'(s) = \{x | \exists \text{ a path } d \rightarrow n, d \in def(x)\}$$

i.e. $in(s)$ is the subset of $in(s)$ of variables whose definition reaches node n. The resulting sets, for the considered example, are shown in Figure 3. Summarizing, in the proposed approach liveness information is obtained iteratively solving the modified DF equations, as formulated in (5) and (6), and then computing $live_p$ sets as in (4), which characterize the relations between variables alive at parallel nodes.

d) Mutual Exclusiveness: In this section, parallel nodes have been described as nodes such that there is not a path between them in the DG. According to this definition, operations belonging to mutually exclusive branches will be detected as parallel. This can be avoided refining the set $parallel(x)$, defined in (3) as follows: if node x lays on the i-th branch of a n-ary branch node d, then any node y, laying on the j-th branch of d ($j \neq i$), is removed from the set $parallel(x)$.

TABLE I: *High Level Synthesis results: Register Binding.*

	Conflict-Free		Non Conflict-Free	
Benchmark	**# Regs**	**# Regs (Unique)**	**# Regs (Coloring)**	**# Regs (Left Edge)**
crc32	13	19	7	7
ethernet	13	42	8	8
gcd	15	37	9	9
sha-1	32	161	15	15
cftmdl	39	72	14	14
chem	176	341	176	176
dir	65	121	63	63
lee	27	72	8	8
matmul	16	28	16	16
mcm	46	94	30	30

Mutually exclusive nodes can be detected considering the control dependencies in the DG.

V. Conflict Graph Construction

Liveness analysis allows the definition of an interference relation among the storage values, usually represented through a Conflict Graph (CG). Overlapping life-intervals may be detected looking at live-out sets. Given a node v in the DG, storage values belonging to $live\_out(v)$ *interfere* with each other, since the associated variables are simultaneously alive. Storage values belonging to $live\_out_p(v)$ in stead *may not interfere each other*; however each storage value $x_p \in live\_out_p(v)$ interfere with each storage value $x \in live\_out(v)$. This properties avoid the insertion of unnecessary interferences. Consider as an example the DG in Figure 1: it results $2||3$, $2||5$, $4||3$, $4||5$. $Live\_out_p$ for node 2 contains both b and $b1$, but b and $b1$ will never be simultaneously alive. Thus, while storage values belonging to $live\_out(v)$ will be *cliqued* in the CG, storage values belonging to $live\_out(v)_p$ will not. This is the reason for keeping these sets separated. Once built the CG, register binding may be addressed through standard techniques such as vertex coloring algorithms.

VI. Experimental Results

In order to validate the described methodology, a register allocator based on the proposed liveness analysis has been implemented in C++, and integrated in a HLS flow [11]. Such allocator performs the register binding task by coloring a CG. We choose to target a dynamic scheduling architecture, composed by a datapath and a distributed controller, as described in [10]. The execution of each operation starts when all its dependencies are satisfied. This check is performed directly at run-time without considering any pre-computed schedule. This makes such architecture a good candidate for validating our approach, which aims to provide a conflict free register assignment regardless to the run-time schedule. All the considered applications have been synthesized adopting the proposed register binding, and allocating a register for each storage value (*unique binding*). This comparison is motivated by the fact that unique binding always produces a conflict free binding, thus not impacting on performance. The same Functional Unit binding has been adopted for both the approaches, in order to isolate the effects of the register binding. FUs sharing is managed through dedicated logic, without introducing false dependencies imposing serializations. For each benchmark, both the approaches led to the same execution latency (in terms of clock cycles): this result confirms the proposed methodology to produce a conflict-free binding.

TABLE II: *Design Compiler Synthesis Results.*

	Conflict-Free									Non Conflict-Free				
	Proposed Approach				Unique Binding					Vertex Coloring				
Bench	**SEQ**	**CMB**	**CON**	**TOT**	**SEQ**	**CMB**	**CON**	**TOT**	**Gain**	**SEQ**	**CMB**	**CON**	**TOT**	**Gain**
crc32	2189	5125	1716	9032	3262	5188	1903	10354	-12.8%	1251	7880	2168	11300	-20.1%
ethernet	2189	5504	2040	9734	6837	5520	2747	15104	-35.5%	1430	5874	2045	9350	+4.11 %
gcd	2681	4899	2250	9831	6094	4796	2739	13630	-27.8%	1609	4600	1991	8200	+19.9%
sha-1	5720	53394	14952	74067	28779	55310	19095	103184	-28.2%	2681	86527	21469	110678	-33.1%
cftmdl	6971	32557	9681	49209	12870	32812	10675	56358	-12.7%	2502	31114	8505	42121	+16.83%
chem	31460	486579	112141	630180	61133	487025	116870	665028	-5.2%	31460	486817	112235	630512	-0.05%
dir	11619	156088	36421	204128	21629	156654	38171	216454	-5.7%	11261	156444	36455	204160	-0.02%
lee	4826	30320	8447	43594	12870	30583	9763	53216	-18.1%	1430	29774	7747	38951	+11.9%
matmul	2860	43901	10078	56839	5005	44026	10454	59486	-4.45%	2860	44020	10122	57002	-0.3%
mcm	8222	89471	21846	119540	16803	90330	23491	130624	-8.48%	5362	89343	21399	116104	+2.9%

Synthesis Results Our approach has been compared with three different register binding algorithms: unique binding, left edge and standard vertex coloring based binding. Table I compares the number of allocated registers. Moreover the datapaths designs obtained through our algorithm, unique binding and vertex coloring, have been synthesized by means of Synopsys Design Compiler, using Nangate 45nm Open Cell Library. Table II reports the obtained results, indicating non-combinational (*SEQ* columns), combinational (*CMB*), interconnection (*CON*) and overall (*TOT*) area costs in terms of library units. Percentage average gains adopting the proposed approach are also shown.

Comparison with unique binding The provided HLS results show that the proposed binding allocates 55.2% fewer registers, on average, when compared with the unique binding, that is the only other conflict free strategy. This result has been confirmed by the synthesis experiments, that reported a 55.8% average reduction in non-combinational area. In addition, despite the introduction of steering logic due to the register sharing it is also observed, an average reduction for both the combinational and interconnection parts: 0.48% and 6.92% respectively.

Comparison with non conflict-free approaches In the conducted experiments, left edge and vertex coloring have been considered as non conflict-free approaches. In fact, they exploit standard liveness analysis, where the ordering relation for solving the DF equations is dictated by the scheduling task. As a result, on the contrary compared to unique and proposed bindings, for these approaches parallelism exploitation may be limited by resource conflicts. Table I shows that left edge and vertex coloring lead to allocation of the same number of registers, that is the optimal one for the considered schedules. These more aggressive register sharing techniques allocate on average 21.7% less registers when compared with the proposed one, and 64.9% less registers with respect to unique binding. In two cases (*chem*, *matmul*) our algorithm is able to obtain the same number of registers of left edge and vertex coloring. However it must be taken into account that, since these strategies do not consider the impact of interconnections [12], massive sharing may not always lead to better results in area. This guess is confirmed by the synthesis results for the vertex coloring based datapaths, for which we registered an impact in terms of combinational and interconnection costs: adopting our approach, we obtained an area reduction for both the components (3.67% and 2.03% on average respectively). The effects of the multiplexers allocation arise more clearly considering, for example, the results for the *sha-1* benchmark, where vertex-coloring allows a very aggressive register sharing. In this case we reported for our approach an area reduction of 38.29% for the combinational part, and 30.35% for the interconnections. The greater number of allocated registers for our approach, results in a non-combinational area increase of 27.31%: nevertheless the overall results, including combinational and interconnection costs, demonstrate an overall average area reduction of 1.81%. In the two designs in which the same number of registers is used (*chem* and *matmul*), the area results are nearly the same for both the approaches. These results demonstrate that while providing a conflict free register binding, the proposed approach is not affected on average by area overheads.

VII. Conclusions

In this paper we presented a Liveness Analysis methodology which does not require a pre-computed scheduling nor the definition of a partial ordering relation among operations, making it suitable for dynamic scheduling architectures. Such analysis allows the construction of a conflict graph where storage values alive at parallel nodes always interfere: performing the register binding exploiting this graph avoids the insertion of false dependencies to prevent resource conflicts. The approach always leads to a conflict-free binding, avoiding performance loss due to unnecessary serializations while ensuring correctness, as confirmed by the experimental results. When compared with conventional approaches, which instead cannot guarantee maximum parallelism exploitation in dynamic environments, our approach leads to the allocation of a greater number of registers: however we observed for different examples, and on average, an overall area reduction, due to the cost of interconnections and steering logic.

References

[1] A. Appel and J. Palsberg, *Modern compiler implementation in Java*.

[2] S. M. Mueller, "On the scheduling of variable latency functional units," in *Proceedings of the eleventh annual ACM symposium on Parallel algorithms and architectures*, ser. SPAA '99, 1999.

[3] R. Beidas and J. Zhu, "Scalable interprocedural register allocation for high level synthesis," in *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, 2005.

[4] P. Brisk, F. Dabiri, R. Jafari, and M. Sarrafzadeh, "Optimal register sharing for high-level synthesis of ssa form programs," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 25, no. 5, 2006.

[5] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein, "Register allocation via coloring," *Computer Languages*, vol. 6, no. 1.

[6] A. Del Barrio, M. Molina, J. Mendias, R. Hermida, and S. Memik, "Using speculative functional units in high level synthesis," in *Design, Automation and Test in Europe Conference*, 2010.

[7] A. Del Barrio, S. Memik, M. Molina, J. Mendias, and R. Hermida, "A distributed controller for managing speculative functional units in high level synthesis," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 30, no. 3, 2011.

[8] G. D. Micheli, *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education, 1994.

[9] S. S. Pinter, "Register allocation with instruction scheduling," *SIGPLAN Not.*, vol. 28, 1993.

[10] Pilato, C., Castellana, V.G., Lovergine, S., and Ferrandi, F., ""A Runtime Adaptive Controller for Supporting Hardware Components with Variable Latency"," *NASA/ESA Conference on Adaptive Hardware and Systems*, 2011.

[11] "PandA: A framework for Hardware-Software Co-Design of Embedded Systems. http://panda.dei.polimi.it."

[12] D. Chen and J. Cong, "Register binding and port assignment for multiplexer optimization," in *Proceedings of the 2004 Asia and South Pacific Design Automation Conference*, 2004.