Placement-and-routing-based Register Allocation for Coarse-grained Reconfigurable Arrays

Bjorn De Sutter
Ghent University - IMEC
brdsutte@elis.ugent.be

Paul Coene Tom Vander Aa
Interuniversity Micro-Electronics Center
{coene,vanderaa}@imec.be

Bingfeng Mei
meibf@yahoo.com

Abstract

DSP architectures often feature multiple register files with sparse connections to a large set of ALUs. For such DSPs, traditional register allocation algorithms suffer from a lot of problems, including a lack of retargetability and phase-ordering problems. This paper studies alternative register allocation techniques based on placement and routing. Different register file models are studied and evaluated on a state-of-the art coarse-grained reconfigurable array DSP, together with a new post-pass register allocator for rotating register files.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—code generation, retargetable compilers

General Terms Algorithms, Design, Performance

Keywords register allocation, placement and routing, coarse-grained, reconfigurable arrays

1. Introduction

Many applications contain loops that exhibit large amounts of parallelism. To exploit this, processors have to offer a high execution bandwidth. In embedded application domains, VLIW DSPs are popular because of their good performance/power ratio and compiler support. Unfortunately, VLIW architectures do not scale well, as the power consumption and delay of a register file (RF) scales super-linearly with its number of ports. While clustering overcomes this problem to some extent, clustered VLIWs still feature quite large, multi-port, and hence power-hungry, RFs. Alternatively, direct connections between arithmetic logic units (ALUs) can be added to the DSP data paths. For example, by making the forwarding paths on pipelined processor explicitly accessible through the instruction set architecture (ISA), many RF accesses can be omitted for short-lived values [25]. This creates opportunities for lowering the number of RF ports without paying a price in obtained IPC.

Another example of architectures with explicit connections between ALUs are coarse-grained reconfigurable arrays (CGRAs) [22, 24], of which Figure 1 depicts an example. CGRAs can be seen as coarse-grained FPGAs in which look-up tables have been replaced by word-wide ALUs and RFs, and in which a new CGRA configuration can be loaded every cycle. As such, a CGRA can also be seen

LCTES '08 June 12–13, 2008,Tucson, Arizona, USA.
Copyright © 2008 ACM 978-1-60558-104-0/08/06. . . $5.00

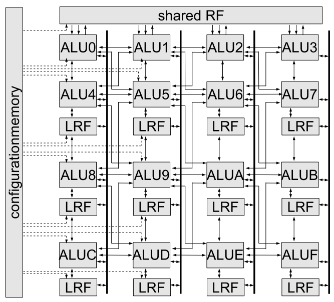


Figure 1. An example CGRA featuring one shared RF, 16 ALUs, 12 local RFs, and a sparse interconnect.

as an extension of a clustered VLIW to a more generic, 2D type of VLIW in which all components, including the multiplexers of the interconnect network, are programmed explicitly every cycle with an ultra long instruction word. An important difference between CGRAs and clustered VLIWs is that a typical CGRA ALU is not connected to a single RF via multiple implicit connections. Instead each ALU can be connected to multiple RFs via a heterogeneous interconnect topology. As such, it can occur that the two or more inputs of an ALU are connected to different sets of RFs. Conversely, a single RF port may be shared by more than one ALU. This allows data to flow between multiple RFs and ALUs, which is required for executing complex loops at high instruction-level parallelism (ILP), while still using single-ported, power-efficient RFs. Also, to reach higher clock frequencies, additional pipelining latches can be inserted nearly anywhere in the interconnect network of the data path. Because CGRA instances optimized for different application domains can differ significantly in the number of RFs and ALUs and in interconnect topology, CGRA code generation techniques ideally should be retargetable. For example, they should support RFs that are tightly coupled to single ALUs, as well as RFs that are shared between a number of ALUs.

To the best of our knowledge, traditional code scheduling and register allocation strategies fail in targeting CGRA architectures or clustered VLIW architectures with explicit forwarding paths. For one thing, when there are a large number of small RFs, splitting the register allocation in the separate cluster assignment [19, 10] task with or without instruction replication [2] and intra-cluster register allocation does not work anymore. Furthermore, with sparsely connected RFs and ALUs, the assumption of most code schedulers that an ALU is connected to its corresponding RF through exclusive implicit connections does not hold anymore. So even code generation algorithms that integrate cluster assignment with instruction scheduling and register allocation in one phase [31] do not scale to CGRA-like architectures.

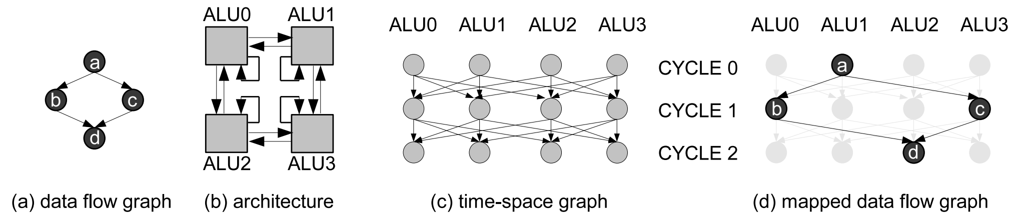


Figure 2. (a) A diamond-shaped data flow graph, (b) a simple data path consisting of ALUs only, (c) the corresponding time-space routing graph, and (d) the mapped data flow graph.

This paper studies placement and routing (P&R)-based code generation techniques as an alternative to existing register allocation algorithms. A comprehensive overview and a comparison of different RF models are presented, and a novel register allocation algorithm is presented that enables the use of more compact RF models, thus leading to significantly smaller exploration spaces and hence faster compilation. We do not claim that the proposed allocator or the used models can outperform known code generation techniques for more traditional (clustered) VLIW architectures. We do claim, however, that those known techniques do not apply to CGRA architectures, while P&R-based techniques do.

Section 2 presents the general principle of code generation based on P&R. Section 3 presents several models for different RF properties. Section 4 discusses the problem of rotating RF models in conjunction with modulo-scheduling, and Section 5 presents a new post-pass scheduler that solves this problem. An evaluation of the discussed models is presented in Section 6. Section 7 discusses related work, and conclusions are drawn in Section 8.

2. Placement and Routing based Code Generation

For architectures with programmable connections between ALUs, a compiler not only needs to schedule operations and perform register allocation. On top of that it also needs to decide how data values will be transported from their producing operations to their consuming operations. In other words, the compiler needs to decide through which connections the data will be moved from one ALU to another. Stated in terms that originate from the FPGA synthesis world [3], the compiler needs to place and route the code. Placement decides where and when each operation will be executed, and routing decides through which connections the data will flow as required by the data dependence graph (DDG).

To understand the concept of P&R in the context of compiler code generation, consider the DDG of a code fragment depicted in Figure 2(a) and the simple architecture in Figure 2(b). The problem of placement and routing consists of mapping the DDG onto a space-time graph that models the computational resources and the routing resources (the connections) of an architecture. In hardware synthesis terminology, the latter graph is called the routing resource graph (RRG) [3]. In this graph, each resource is replicated at every cycle in the schedule. In other words, each node $n(r, t)$ in the graph corresponds to resource r at cycle t. The RRG[1] modeling the example architecture is shown in Figure 2(c).

Figure 2(d) shows the mapped DDG. From this mapped DDG, all assembly code can be derived. In general, a DDG mapping onto a RRG is valid if each node in the RRG is part of at most one routed dependency. In hardware synthesis speak, each architecture pin (=node) should only be used in one routed net (=routed dependency) or, in other words, no pin should be overused.

[1] In this RRG, as in all later RRGs in this paper, ALUs are modeled by means of single nodes. Alternatively, all ALU input and output ports can be modeled as separate nodes, as in [23]. Not to overload the figures, and because this paper focuses on the RF modeling rather than on the ALU modeling, we omit the separate ALU port nodes from the RRGs.

Numerous algorithms have been devised to perform P&R in hardware synthesis [3], and variations have been designed for code generation purposes [17, 23]. Throughout most of this paper, the exact algorithm used is irrelevant. We therefore postpone references to any specific algorithm until the evaluation section.

3. Register File Models

Just like ALUs are modeled with nodes that are replicated every cycle in the RRG, RF resources will be modeled with nodes and edges. As such, each RF is modeled with an appropriate RRG subgraph. By adapting the edges in this subgraph, different RF properties can be modeled.

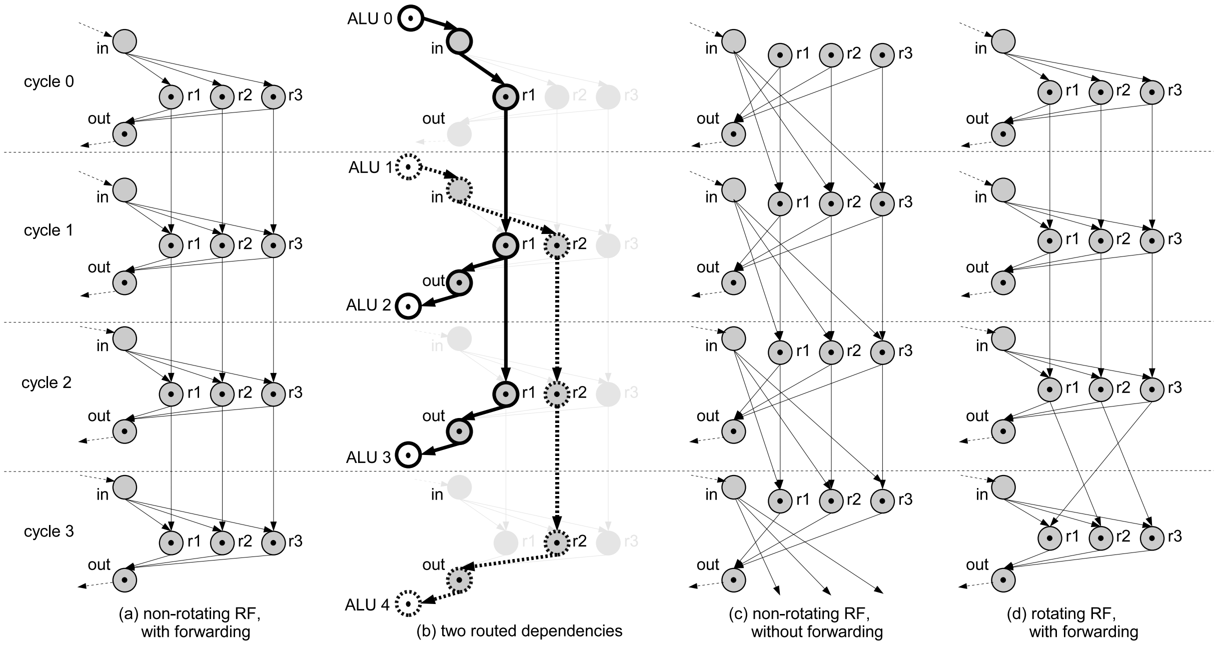
3.1 Basic Model

Figure 3(a) depicts the basic model of a RF with one write port, one read port, and three registers. The single write port is modeled by the `in` node. All connections from (potentially multiple) ALU output ports or from other components to the RF write port will be modeled with edges to the `in` node, in every cycle of the RRG. Likewise, the `out` node models the read port. Connections on the processor from this port to ALU input ports or other components will be modeled with edges starting at this `out` node. Such connections, that basically constitute the net-list of the architecture, are modeled with thin dotted edges as in Figure 3(a). When there are more ports, additional `in` and `out` nodes are added to the RF's RRG subgraph. Furthermore, each register is modeled with one internal node, in this case nodes `r1`, `r2`, and `r3`.

In the RF's RRG subgraph, edges from the `in` node to an internal node model the fact that a value written via the write port can be stored in a register. Edges from the internal nodes to the `out` node model the fact that a value stored in a register can be read via the read port. Finally, edges connecting the internal nodes in one cycle to their counterparts in the next cycle model the fact that stored values can stay in the RF from one cycle to the other, which is of course the main functionality of a RF.

Figure 3(b) shows how register allocation to this RF can be done by means of P&R. Two routed dependencies in the DDG are shown with bold edges, one with a solid net, and one with a dotted net. The solid net shows that a value produced on some `ALU0` in cycle 0 is written to `r1` in cycle. This value is later read in cycle 1 to be consumed on `ALU2`, and again in cycle 2 to be used on `ALU3`. In the mean time another value, corresponding to the second dependency shown with the dotted net, was produced by `ALU1` and written to `r2` in cycle 1. This value is read from `r2` in cycle 3, when it is consumed by `ALU4`.

The router does not need to do the register allocation explicitly: it just needs to find routes from ALUs on which producing opera-



ALU 0
ALU 1
ALU 2
ALU 3
ALU 4
cycle 0
cycle 1
cycle 2
cycle 3
in
out
r1
r2
r3
(a) non-rotating RF, with forwarding
(b) two routed dependencies
(c) non-rotating RF, without forwarding
(d) rotating RF, with forwarding

Figure 3. The RRG subgraphs of three RFs with three registers and one write and one read port.

tions have been placed to ALUs on which the consuming operations have been placed. Once these have been found, the register allocation is derived from the routes. This advantage of not needing to do explicit register allocation is really huge. For one thing, in case the router finds a route between two ALUs that does not pass through a RF's RRG subgraph, this implies that the router has found a direct connection between two ALUs, such as a forwarding path, that will be exploited in the generated schedule. To enable this exploitation, the routing algorithm itself does not need to be adapted. In fact, the routing algorithm does not even need to know whether there are direct connections or not, or whether there are RFs or not, or whether the RF ports are shared for multiple ALUs or not. As soon as the appropriate RRG is built, all edges and nodes in it are equal to the router. Consequently, a P&R routing algorithm using the above RRG model for RFs, however those RFs are connected to other components, will be able to exploit them. Likewise, the router will be able to store short-lived values in latches on forwarding paths or on buses. This can be observed from the RRG subgraphs of such components as depicted in Figure 4(b) and 4(c). These subgraphs show that latches and buses are in fact nothing more than RFs with limited storage times and with only one internal register (of which the node has been merged with the in node in the RRG subgraph). Just like the router will exploit RFs or direct connections, it will exploit latches and buses.

3.2 Registers with and without Forwarding

When taking a closer look at the edges in Figure 3(a), it becomes clear that the register file modeled here is a register file with an internal forwarding network. Indeed, a data dependence can be routed through it within a single cycle, which models that a value being written to the register is already available at its output.

To model a non-rotating RF that lacks internal forwarding, it suffices to change the edges of the RF's RRG subgraph, as depicted in Figure 3(c). Indeed, in this RRG subgraph, a net modeling a dependence cannot enter the RF and exit it in the same cycle: a value being written to the RF can first be read one cycle later.

3.3 Rotating and Non-rotating Registers

Just like it suffices to adapt an RF's RRG subgraph to model the presence or absence of internal forwarding, it suffices to adapt the edges to turn a non-rotating RF into a rotating one. Figure 3(d) depicts the RRG subgraph of a RF of which the registers rotate between cycle 2 and 3. Obviously, mixed combinations of RFs in which only part of the registers rotate can also be modeled easily.

3.4 Multi-nets and Instruction Replication

The data dependence modeled with the solid net in Figure 3(b) is a dependence from one producer to two consumers. Its net goes from one source node (`ALU0` in cycle 0) to two sink nodes (`ALU2` in cycle 1 and `ALU4` in cycle 2). As it has multiple sink nodes, this net is called a multi-net. An important question to answer is the following: where can the subnets constituting a multi-net diverge? In Figure 3(b), this happens at the internal node `r1` in cycle 1.

The answer to this question is simple: wherever the instruction encoding of the architecture allows you to have diverging multi-nets. For example, when a RF read port is programmed on a CGRA, the address of the register addressed must be set. Whether the value will be propagated over more than one outgoing connection or not does not matter: only one address needs to be set. So a multi-net can diverge at `out` nodes of RF RRG subgraphs. Similarly, a register can be read in some cycle and still hold the same value for later cycles. Therefore a multi-net can also diverge in internal nodes of RF RRG subgraphs.

When a value is written through a RF write port, the destination address needs to be set. As only one address can be encoded for each port, the value can only be written to one register — through one port that is. This limitation can be modeled in a P&R algorithm by disallowing multi-nets to diverge at `in` nodes of RF RRG subgraphs. To let a router actually take these limitations into account, it suffices to annotate all RRG nodes with a `diverge` attribute. Thus, this does not complicate routers significantly. In all the RRGs depicted in this paper, the nodes that allow diverging multi-nets are marked with a black dot in the middle of the nodes.

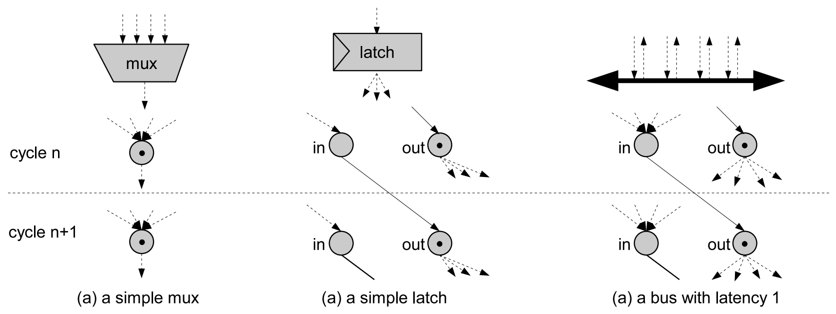


Figure 4. The RRG subgraphs of muxes, latches and buses.

Please note that, depending on how the programming of muxes, ALUs, RFs, buses, etc. is encoded in the CGRA's ISA, diverging may occur at many nodes other than RF nodes. For example, a value being put on a bus may be read by multiple components attached to the bus. When such a diversion happens, the disjoint parts of the subnets of a multi-net may be routed through different (sets of) RFs. Or it may happen that one subnet passes through a RF, while another subnet does not. Both cases correspond to instruction replication [2] as used on clustered VLIWs, where instructions (and hence there computation results) are replicated on multiple clusters to reduce the required amount of intercluster communication. With P&R-based code generation, this "replication" over multiple clusters or RFs can be performed as a side-effect of the routing. No additional, architecture-dependent compiler heuristics or algorithms are needed.

3.5 Schedule and Allocation Validity

The validity of the schedule and the register allocation determined by the result of a P&R using the above models is easily checked: it suffices that each RRG node occurs in only one net. In that case each resource will be used at most once in every cycle. No additional requirements need to be tested besides this sufficient condition. This means that all information needed to generate a valid schedule is present in just the DDG and the RRG. With respect to the quality of the generated schedules, this implies that the quality depends solely on the ability of the P&R strategy to explore all valid placements and routings on an RRG. The P&R algorithm does not need to know that it models an architecture with specific RFs. In other words, no heuristics ever need to be implemented that depend on specific RF properties.

For that reason, we believe P&R-based code generation techniques to be highly retargetable, and definitely more easily retargetable than traditional instruction scheduling and register allocation techniques. We will demonstrate this even more in Section 4, when simple changes to the RRG are presented to support modulo-scheduling of loops.

Another important advantage of not needing to impose additional requirements on validity or of not needing to implement additional heuristics is that all resources can be exploited maximally: when all nodes in an RF's RRG subgraph are used in nets during some cycles, the RF is used to its full capacity. This contrasts with heuristics that may steer register allocation algorithms in a direction that is good on average, but suboptimal in certain cases. In our case, the ability to find the optimal solutions depends on the ability of the router to explore all valid placements and routings. While this is far from guaranteed, as P&R is an NP-complete problem [21], it is at least independent of the specific target architecture for one which is developing or using a compiler.

3.6 Compact Models

With the RF RRG subgraphs presented until now, RRG graphs become huge. As a result, the number of routes that need to be explored by P&R algorithms becomes huge as well. To reduce that number, more compact models with fewer nodes are wanted. To that end, alternative models in which all internal register are modeled with a single node can be used. Figure 5(b) presents such a compact model for a rotating RF with two registers, of which the original model is depicted in Figure 5(a). With the compact model every node gets an associated capacity that models the number of values that can pass through it simultaneously. The validity check for a schedule and allocation then simply needs to check whether the number of nets routed through a node is not higher than the node's capacity.

Especially for large RFs, the compact model can result in much smaller RRGs, and thus in more complete, or faster, exploration of the P&R solution space.

Of course, the downside of the compact models is that no actual register allocation has been performed. Compared to clustered VLIWs, only a cluster assignment has been performed by a router using the compact models: it has been decided which value will be stored in which RFs, but not in which registers inside those RFs. For that reason, a post-pass register allocator will need to allocate the live ranges that correspond to the placed and routed code.

For this post-pass register allocation, there is one important requirement that differentiates it from traditional register allocators: as P&R-based techniques are usually much slower than traditional code generation techniques, one cannot afford having to redo a placement and routing because of a failing post-pass register allocation. In other words, the register allocator should find a valid allocation without needing changes to the found placement. Still in other words, no spill code can be inserted!

Fortunately, for normal code, the capacity checks performed on RF RRG nodes guarantee that a valid schedule and a valid allocation exist because the number of registers that needs to be stored in a RF never exceeds the RF's capacity. As we will see in the next section, however, this guarantee is not as easily obtained for cyclic code such as in software-pipelined loops.

4. P&R-based Modulo Scheduling

4.1 Modulo Scheduling

To map software-pipelined loops onto a CGRA, the presented models support a form of modulo-scheduling [29]. The objective of modulo-scheduling is to engineer a schedule for one iteration of a loop such that this same schedule can be initiated at regular, as short as possible, intervals, taking into account data dependences and resource constraints. This interval in terms of cycles is termed *initiation interval* (II) [18], essentially reflecting the performance

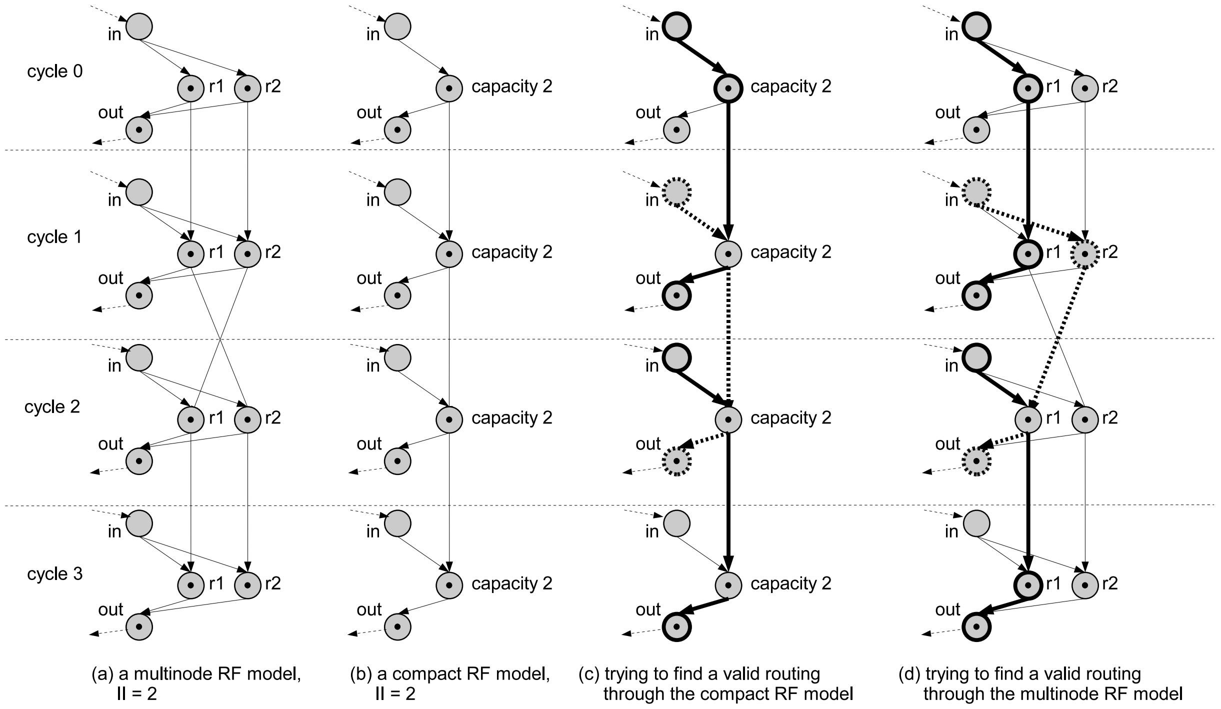

cycle 0
cycle 1
cycle 2
cycle 3
in
out
r1
r2
capacity 2
(a) a multinode RF model, II = 2
(b) a compact RF model, II = 2
(c) trying to find a valid routing through the compact RF model
(d) trying to find a valid routing through the multinode RF model

Figure 5. Depicted are (a) A multi-node RF model for $II = 2$ for a rotating RF with one read port, one write port and two registers, (b) the corresponding single-node model, and (c-d) a mapping trial of two life ranges onto the RF models.

of the scheduled loop. Generally, modulo-scheduling algorithms rely on an abstract architecture model called *modulo reservation table* (MRT) [18] to impose the modulo resource constraints. In the MRT, there are (issue width) * II reservation slots; one per issue slot per time slot. Thus, modulo-scheduling is simplified into an acyclic scheduling problem on the MRT.

In order to enable modulo-scheduling using P&R algorithms that use our RRG models, it suffices to add a MRT on top of the RRG. Basically, instead of modeling each resource of the architecture in the MRT, each node $n(r, t)$ in the RRG is mapped on the MRT entry $n(r, t \mod II)$. For modulo scheduling, the validity check now is no longer performed directly on nodes in the RRG, but instead on the MRT entries. Instead of counting the nets routed through each node in the RRG, we now count, for each entry in the MRT, the nets routed through the nodes that were mapped onto that entry, and apply the validity check on that count. Apart from that, nothing fundamental needs to change to the P&R technique to let it generate modulo schedules.

4.2 Rotating Register Files

In Figure 6(a), the live ranges of three variables v_0, v_1, and v_2 are depicted for a software-pipelined loop with $II = 3$. The schedule of one iteration is 7 cycles long, hence there are 3 pipeline stages. The value $v_{1,i-1}$ of variable v_1, i.e., the value of v_1 produced in iteration $i - 1$, is live from cycle 0 (when it is produced) to cycle 6 (when it is last consumed) in the depicted time domain. As such, it overlaps with the live range of $v_{1,i}$. As long as both values are live together, they both need to be stored, and therefore at least two registers are required (assuming that the values are stored in a RF). These overlaps are depicted in Figure 6(B) in a modulo-time graph. As the code that produces $v_{1,i}$ is exactly the same as the code that produced $v_{1,i-1}$ 3 cycles earlier, $v_{1,i}$ will be moved to the same register R to which $v_{1,i-1}$ was moved 3 cycles earlier. By consequence, the only way to store both constants somewhere together is to copy or to move the value $v_{1,i-1}$ to a location other than R before it is overwritten in R with $v_{1,i}$.

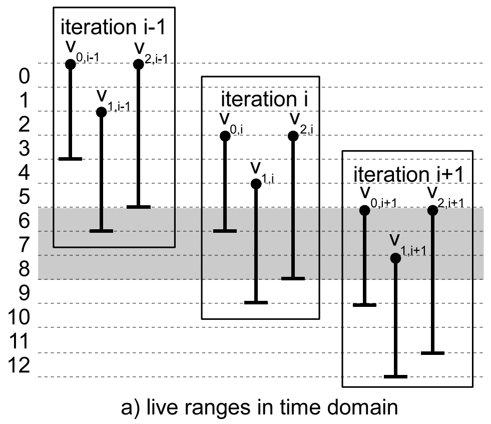
In general, in modulo-scheduled loops, multiple values of the same variable need to be stored (in an RF or in latches or in connections with high latency) simultaneously whenever a live range is longer than II cycles. To support this without having to insert explicit copy operations or move operations in the schedules, rotating RFs can be used. These RFs rotate at the end of a pipeline stage, being every II cycles, and rotate values from previous iterations into neighboring registers before they are overwritten by values from new iterations. We refer to [16] for additional information on the use of rotating registers in software-pipelined loops. What is important in the context of this paper, is that registers typically rotate at the end of each pipeline stage of a software-pipelined loop, i.e., after every II cycles.

4.3 Determining Register File Capacities

Now suppose we are using a P&R-based code generation technique to map a modulo-scheduled loop with $II = 2$ on an architecture with a RF with 2 registers, with one write port and with one read port. The corresponding RRGs with the multi-node and with the compact models are depicted in Figure 5(a) and Figure 5(b).

We will now study whether it is possible to accommodate two live ranges in this RF: one range from cycle 0 to cycle 1, and a second range from cycle 1 to cycle 2. These ranges are depicted in solid and dotted edges in Figure 5(c). The first range is depicted twice to illustrate that the resources it uses are not only used at cycles 0 and 1, but also at $0 + II$ and $1 + II$, at $0 + 2 * II$ and $1 + 2 * II$, and so on.

When the capacity is chosen to be two, as indicated in Figure 5, the allocation on the compact model as depicted in Figure 5(c) seems valid: at most two nets, corresponding to the two live ranges,



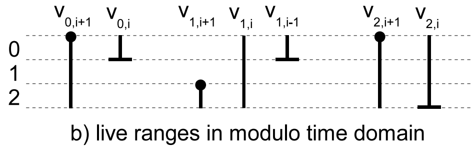


Figure 6. Lifetimes in consecutive iterations when $II = 3$.

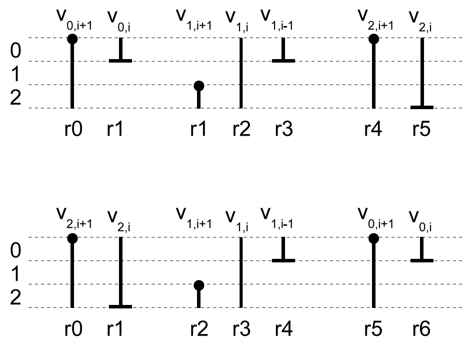


Figure 7. Two possible solutions for rotating RF allocation.

are routed through a node. When looking at the RRG mapping in Figure 5(d) however, which is one of the possible multi-node model mappings corresponding to the compact model mapping, we note that the schedule is not valid. Indeed, node `r1` in cycle 2 is used in two nets. A similar problem occurs for all other multi-node model mappings that correspond to the compact model mappings.

This example shows that choosing the number of registers in a rotating RF as the capacity of the internal node of an RRG subgraph does not guarantee that a valid allocation exists. To provide such a guarantee, which we need as argued in Section 3.6, we rely on the proof by Touati and Eisenbeis [32] that a valid register allocation in a rotating RF exists when the number of simultaneously live values in an RF, termed *cyclic register requirement* or CRR, is at least one smaller than the number of registers of that RF, i.e., if

$$\text{CRR} + 1 \leq \#\text{RF} \tag{1}$$

This means that the number of DDG edges that can be routed through an internal RF node in the compact model needs to be limited to #RF-1. In other words, when the capacity in a RF RRG graph is set to #RF-1, the existence of a valid allocation is guaranteed. The only remaining problem then is to find this allocation.

5. Traveling Salesman Solution for Rotating RFs

Before we present our post-pass register allocator that can find a valid allocation, we want to repeat that introducing spill code is not an option. The context of our register allocator is that in which a P&R algorithm has spent a (very) long time on finding a valid schedule for which it has guaranteed that a valid allocation exists. Our job is to find at least one such valid allocation without requiring updates to the found schedule for inserting spill code.

Consider the example in Figure 6. Different values of v_0, v_1, ... v_{N-1}, each belonging to a different iteration, exist simultaneously in the software pipelined loop. These so-called *circular excessive values* $v_{0,i}$, $v_{0,i-1}$, ...[2] must be mapped to adjacent rotating registers, such that the rotation mechanism in the RF will address the correct values when a new iteration is started, which coincides with the start of a new software pipeline stage. Defining a sequence S as a mapping of $\{0, 1, ..., N-1\}$ to $\{v_0, v_1, ..., v_{N-1}\}$, the rightmost circular excessive value of the ith element in the sequence $v_{S(i)}$ can share a register with the leftmost one of the next value in the sequence $v_{S((i+1) \bmod N)}$ when the modulo lifetime of $v_{S(i)}$ ends before the modulo lifetime of $v_{S((i+1) \bmod N)}$ begins. For example, performing an allocation in the order (v_0, v_1, v_2) (top Figure 7) uses 6 registers, while the order (v_2, v_1, v_0) (Figure 7b) gives 7. In the first solution, $v_{0,i-1}$ and $v_{1,i}$ can share register `r1`, thereby reducing the total number of required registers. This illustrates clearly that the chosen sequence influences the quality of the solution.

In general, the register allocation problem can be solved by finding a sequence that meets the rotating RF capacity constraint. It is useful to define a distance function $d(v_a, v_b)$ as the number of unused register time slots when v_b follows v_a in the allocation sequence. In Figure 7, $d(v_0, v_1) = 1$, $d(v_1, v_2) = 2$, $d(v_2, v_0) = 0$, and $d(v_1, v_0) = 2$.

The total number R_S of registers used by the sequence S can then be written as:

$$R_S = \frac{\sum_{i=0}^{N-1} L(v_{S(i)}) + d(v_{S(i)}, v_{S((i+1) \bmod N)})}{II} \tag{2}$$

in which $L(v_i)$ is the life time of v_i: the number of cycles elapsed between its production and last consumption. Because of constraint (1) on the CRR, there exists at least one valid sequence S_{valid} that uses a small enough number of registers to guarantee allocatability. Looking at equation (2), the number of registers used by a sequence S_{min} can be minimized by minimizing $\sum_{i=0}^{N-1} d(v_{S_{min}(i)}, v_{S_{min}((i+1) \bmod N)})$, since the sum of the life times is the same for all sequences and depends only on the schedule of the software pipelined loop. Since S_{min} does not use more registers than S_{valid}, it must be a valid, allocatable sequence.

Minimizing the sum of distances can be regarded as an asymmetric traveling salesman problem (TSP), which consists of finding the cheapest round trip that visits each city exactly once and then returns to the starting city, given a number of cities and the costs for traveling from from each city to each other city. In asymmetric variants, the cost for going from city A to city B is not necessarily the same as for going from B to A. Applied to the register allocation problem, we consider v_0, v_1, v_2, .., v_{N-1} to be cities, with their distances as defined above. An exact solution can be found using the algorithm described in [7].

By using this algorithm as a post-pass register allocator, we have a method to use more compact RF models. Since our method excludes the many schedules in which we would be able to find a valid allocation even if CRR=#RF, setting the capacity of a RFs internal node to #RF-1 can in theory result in suboptimal exploitation of the

[2] The ordering of the circular excessive values needs to correspond to the direction in which registers are rotated and the signs (addition or subtraction) in the formulas.

RFs. Fortunately, we will see in the evaluation that we pay no such price in practice.

6. Experimental Evaluation

To evaluate the strength of our register allocation models and the associated register allocator, we ran them in a modulo-scheduling P&R-based compiler for the ADRES (Architecture for Dynamically Reconfigurable Embedded Systems) template [22]. ADRES processors feature a VLIW operating mode for executing non-loop code, and a CGRA like the one depicted in Figure 1 for executing high-ILP, modulo-scheduled loops that have been transformed into hyperblocks [20] to expose the ILP.

6.1 Compilation Framework

The ANSI C compiler that targets ADRES processors is called DRESC (Dynamically Reconfigurable Embedded Systems Compiler). It uses the RRG with a MRT on top of it to schedule code for the array. More details on the used compiler algorithm can be found in [23]. Here we limit the discussion to some ADRES/DRESC features that are important for this evaluation.

First, it is important to know that the VLIW mode and the array mode share the L1 scratch-pad memory. Hence data can be passed between the two modes via the shared memory. Furthermore, the VLIW mode and the array mode share one RF, like the shared RF in Figure 1. This RF can hence also be used to pass data from the VLIW mode to the array mode and vice versa. When this needs to happen, the placer of the compiler's P&R algorithm places virtual operations corresponding to live-in and live-out dependencies on the shared RF's nodes in the RRG, and the router routes the nets corresponding to those dependencies, just like it routes any other dependencies. This way, only the placement step needs to be adapted to deal with live-in and live-out values that are passed through the shared RF. As for modeling RFs with different properties, the router does not need to be adapted at all.

The P&R algorithm in DRESC is based on congestion negotiation [3] and simulated annealing (SA). The SA starts with an invalid, congested schedule. Congestion in this context means that a resource (a connection or an ALU) is used to perform more tasks in a cycle than it can actually execute. For an ALU, for example, congestion corresponds to executing more than one operation in the same cycle. For a connection, it corresponds to propagating more than one value, or, in other words, being used to route two nets in the same cycle. The cost function used in our SA includes congestion. The SA starts with a schedule with a high cost function, i.e. a lot of congestion, and then tries to minimize the congestion by moving operations around in the schedule until a congestion-free, valid schedule is found. During every iteration of the SA, operations are moved to random new positions in the RRG, after which their dependences are rerouted, and the new cost, including the new congestion, is computed. If the delta in cost of such a move is acceptable for the SA, the move is accepted. Otherwise, alternative moves are tried.

We should note that simply by using SA, common wisdom tells that long compilation times can be expected. In our case, every tried move of an operation in the RRG during the SA involves computing new routes for its incoming and outgoing nets. Therefore the router is the inner loop of our compiler, in which about 95% of the compilation time is spent.

Data is passed from VLIW mode to array mode whenever a loop is entered that was compiled for the array mode. The VLIW code leaves the live-in data in the shared RF (which is the VLIW mode's main RF), and control is transferred to the array controller, much like control would be transferred from a caller function to a calling function. When the loop finishes executing in the array mode, the live-out data is found in the shared RF, and control goes back to the VLIW mode, much like control is transferred upon a function return. This way, an ADRES processor can run both non-kernel code and kernel code without a lot of switching overhead, and co-code generation in the DRESC compiler is relatively easy as well. The non-kernel code is mapped to the VLIW mode with compiler techniques like graph coloring and list scheduling. These are not considered in this paper. Here we focus on the modulo-scheduled code of the array mode.

6.2 Benchmarks and Target Processors

For our experiments, we have compiled a number of micro-benchmarks existing of one or two kernels, as well as a number of applications containing many kernels, for two ADRES instances[3].

The first ADRES instance is designed for multimedia applications, and for this one we compiled an IDCT micro-benchmark, an MPEG-2 decoder, and a more complex H.264 video decoder, for which we report on the most time-consuming loops. The array mode of this instance is very similar to the architecture depicted in Figure 1. It is a 32-bit architecture that features 16 ALUs, 12 small rotating RFs with 4 registers (8 of which have 1 read and 1 write port, and 4 of which have 2 read ports and 1 write port), and one shared RF of 64 registers, of which 32 registers are rotating. The shared RF has 6 read ports and 3 write ports, as needed by the 3-issue VLIW mode of this machine. To support low latencies in VLIW mode, the shared RF offers forwarding. The 12 local RFs do not. This architecture does not contain the buses (thick lines) depicted in Figure 1. Instead, the interconnect between the ALUs and RFs is based solely on a mesh-like interconnect that connects ALUs and RFs to their immediate neighbors in the array. Besides all usual arithmetic and logic operations. all ALUs can execute specific multimedia operations such as clipping, and two-way SIMD operations (which are programmed by means of intrinsics). 8 ALUs contain multipliers, and 4 can also perform load/store operations.

The other instance targets software-defined radio (SDR) code, and for this one we compiled typical kernels as found in wireless standards: FFTs, data shuffling operations, demapping kernels, etc. This architecture differs significantly from the multimedia instance. The ALUs and local RFs are basically the same, but in this case they are 64-bit wide, and all local registers have 2 read ports. Also, the ALUs now can execute 64-bit SIMD operations in saturated arithmetic, including operations like complex fixed-point multiplications. The interconnect of the SDR instance is based on mesh-plus (which connects all elements to direct neighbors and to neighbors one hop away), and it features both horizontal and vertical busses that connect all components per row and per column. Finally, only 16 of the 64 registers of the shared RF are now rotating.

Because the performance targets of the SDR instance and the multimedia instance are different, they need to be clocked at different speeds: 400 MHz for the SDR instance, and 300 MHz for the multimedia instance. To achieve these speeds in the 90 nm TSMC GP standard cell process technology, pipelining latches needed to be inserted in the interconnect to avoid overly long combinatorial paths. Since the speeds of the two instances are different, and because muxes on the SDR instance have more inputs (there are more

[3] A much wider range of architectures has been targeted to study other aspects of ADRES and DRESC [4, 8]. Those studies show that the compiler is indeed retargetable to a wide range of ADRES instances. We limit this paper to two ADRES instances because these are the ones for which we have spent a lot of effort in optimizing the source C code, for example by means of inserting SIMD intrinsics, and for which we have working, synthesizable VHDL. The SDR ADRES instance is being taped out in Q1 2008 as part of a full SDR MPSoC platform. The multimedia ADRES instance has been verified at the gate-level as well, before and after layout. This illustrates that these ADRES instances are real DSP cores that work, and not only virtual architectures that live in simulators.

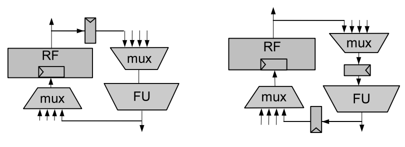


Figure 8. Pipelining latches inserted in the two instances.

	Multi-node RF		Compact RF	
instance	vertices	edges	vertices	edges
multimedia	698	2274	568 (-19%)	1579 (-31%)
SDR	653	2313	552 (-15%)	1620 (-30%)

Table 1. Sizes of the RRGs per cycle.

connections), more latches needed to be inserted in the SDR instance. Conceptually, the latches are inserted in between local RFs and the corresponding ALUs as depicted in Figure 8.

Together with the differences in the interconnect topology, the different pipelining schemes of both instances ensure that these instances significantly different RRGs. The number of nodes per cycle in the instances RRGs is depicted in table 1. These numbers include all muxes, latches, RFs, ALUs, etc in the array data path. It can be seen that the compact models of rotating RFs reduce the sizes of the RRGs significantly: with about 30% less edges, the router should explore about 30% less routes.

6.3 Compilation Results

To evaluate our modeling of registers and the TSP-based register allocator, we mapped all our benchmark kernels to one of the two instances. The results are presented in Table 2.

The first column contains the name of each kernel. The second and third column contain theoretical lower bounds for the obtainable II [28]. ResMII, the resource-minimal II, is the lower bound caused by resource constraints. Consider, for example, a loop with 68 operations. To execute these on an architecture with 16 FUs, at least 5 cycles are needed, so the II will be at least 5. RecMII, or the recurrence-minimal II is the lower bound on II caused by recurrent (i.e. loop-carried) data dependencies. If an iteration depends on operations in a previous iteration that take RecMII cycles to execute, it cannot start executing before these RecMII cycles have gone. The numbers ResMII and RecMII are included in this table to give an idea of how well our compiler techniques are able to approximate the theoretical optimal performance when the mininam II = max(RecMII,ResMII) is reached.[4]

Next, we present the compilation times, the obtained II, and the obtained instructions-per-cycle (IPC) for the code compiled with the multi-node RF models of Sections 3.1, 3.2, and 3.3. Together with the IIs, we've included the difference with the minimal II. It can be seen that the difference is most often very small, and in a large number of loops, it is even 0 or 1. As a result, very high IPCs are obtained. It is only when the RecMII dominates the ResMII, i.e. when the schedule quality is bound by data dependencies, that low IPCs are obtained. This demonstrates the effectiveness of our compiler techniques.

For the compact models of Section 3.6, we've included the same numbers in Table 2 plus the compilation time ratios (lower is better) and IPC ratios (higher is better) relative to the multi-node models, and the time spent in the TSP post-pass register allocator. It can be seen that the time spent in the TSP allocator is negligible compared to the P&R compilation times. Overall, the compact models are 24% faster, which is due to the reduction in size of the RRGs.

Sometimes however, the compilation times increase. This is due to the nature of the SA algorithm. This algorithm relies on a limited number of random placement trials, and as the cost function used for the two RF models is not identical, the compiler traverses the exploration space in a different order. For that reason, it sometimes takes more SA rounds before a valid schedule is found in the exploration space, and hence more time. As a result of this different traversal, not only the number of SA rounds differs, but so do the found schedules. In this respect, we observed no significant differences in the quality of the obtained schedules: in some cases the multi-node model yields better results, in other cases the compact model finds the highest IPC.

The final 5 columns in Table 2 depicts the number of nets that are routed through 0, 1, 2, 3, or 4 different RFs (using the compact RF models). The 0 RF case corresponds to the router finding direct connections between ALUs to route a net. The other cases correspond to the router storing a value in at least one RF to propagate it from the producing operation to the consuming operation in the schedule. The cases 2, 3 and 4, correspond to the router implementing replication in 2, 3 and 4 RFs, as discussed in Section 3.4. From the numbers, it is clear that replication is indeed performed by the compiler.

7. Related Work

The most popular approach for allocating registers is based on graph coloring [9, 5]. Using an interference graph, which models overlapping live ranges, live ranges are allocated and spill code is inserted according to the heuristics used. Callahan and Koblenz [6] applied this register allocation hierarchically to regions of code such as nested loops rather than to whole procedures. This way, they were able to reduce the number of spill code operations inserted in the frequently executed code, thus reducing the number of dynamic memory accesses. Register coalescing [12, 26] aims at elimination copy operations from schedules by coalescing live ranges. The heuristics used to do so take into account the increased difficulty of allocating the resulting longer live ranges. Because graph coloring techniques can require large computation times (as the size of the interference graph is in the worst case quadratic to the number of live ranges), linear scan allocators [27] have been proposed that traverse the live ranges only once.

All these techniques try to minimize the (dynamic) amount of spill code inserted into a program. Our post-pass allocator solves a different problem. In our case, a long time has been spent by the compiler in generating a valid (modulo) schedule, for which a register allocation for each RF (or bank or cluster) is guaranteed to exist. The sole task of our post-pass register allocator is to find one such valid schedule without spill code. This follows from the fact that inserting spill code in the (modulo-)scheduled code is impossible without having to regenerate the schedule from scratch. While the latter is a problem in simple VLIW register allocators as well, regenerating a simple VLIW schedule from scratch for a basic block or hyperblock takes much less time. Also the existing work on regional register allocation is not applicable in our context, as the proposed models are used on single hyperblocks of code anyway. Furthermore, register coalescing is not relevant in our case, as the DDG for which we generate a schedule does not contain copy operations. Whether or not a value is stored more than once in a RF (as with copy operations) is decided by the router in the same way as it decides to perform value replication [2] or not.

Earliest work on register allocation for software pipelined loops was done by Rau, *et al.* [30]. This work defines *MAXLIVE* as the maximum number of simultaneously live values in the loop

[4] Please note that max(RecMII,ResMII) is only an upper bound, which is not guaranteed to be achievable.

Kernel			Multi-node RF graph			Compact RF graph + TSP						Net distribution				
	Res MII	Rec MII	time (s)	II	IPC	time (s)	time ratio	II	IPC	IPC ratio	TSP (ms)	0 RFs	1 RF	2 RFs	3 RFs	4 RFs
FIR_shift	3	1	125	3 (+0)	12.67	67	0.54	3 (+0)	12.67	1.00	1.50	25	13		1	
mat_mul	4	1	130	5 (+1)	9.40	208	1.60	4 (+0)	11.75	1.25	1.76	27	26	1		
idct_1	9	1	218	11 (+2)	13.09	186	0.85	11 (+2)	13.09	1.00	2.79	75	59	4		
idct_2	14	1	682	16 (+2)	13.31	273	0.40	16 (+2)	13.31	1.00	3.47	92	96	2		
SDR kernels																
demapQAM64	4	1	66	6 (+2)	9.83	45	0.68	6 (+2)	9.83	1.00	0.57	57	8			
fft	8	2	298	12 (+4)	10.25	156	0.52	12 (+4)	10.25	1.00	1.74	94	24	2		
R8	8	1	58	11 (+3)	11.09	53	0.91	11 (+3)	11.09	1.00	1.53	92	24	3		
R2	6	1	34	10 (+4)	8.30	25	0.74	10 (+4)	8.30	1.00	0.77	74	13			
fft1024	8	1	268	10 (+2)	12.40	156	0.58	11 (+3)	11.27	0.91	2.46	80	45	1		
ifft64	8	2	302	12 (+4)	10.25	156	0.52	12 (+4)	10.25	1.00	1.70	94	24	2		
DataShuffle	14	1	356	16 (+2)	9.56	478	1.34	16 (+2)	9.56	1.00	2.13	118	30			
MPEG2 kernels																
Dequantize_Non_Intra	2	1	37	2 (+0)	9.50	19	0.51	2 (+0)	9.50	1.00	1.07	15	10			
Dequantize_Intra	2	1	28	2 (+0)	8.50	25	0.89	2 (+0)	8.50	1.00	0.94	14	7	2		
Add_Block_1	3	1	24	3 (+0)	11.00	27	1.13	3 (+0)	11.00	1.00	1.18	17	13	2		
Saturate_1	4	2	219	5 (+1)	11.00	153	0.70	5 (+1)	11.00	1.00	1.45	27	17	2	1	
Fast_IDCT_1	5	1	94	9 (+4)	8.78	111	1.18	8 (+3)	9.88	1.13	1.52	44	26	2		
Fast_IDCT_2	7	1	600	8 (+1)	13.00	392	0.65	8 (+1)	13.00	1.00	2.56	50	43	3	1	
component_prediction_1	4	2	49	4 (+0)	8.25	57	1.16	4 (+0)	8.25	1.00	1.17	12	15	1		
component_prediction_2	3	2	39	4 (+1)	9.50	19	0.49	3 (+0)	12.67	1.33	1.32	24	12		1	
component_prediction_3	3	2	84	4 (+1)	13.00	18	0.21	4 (+1)	13.00	1.00	1.24	30	12	2		
component_prediction_4	4	2	37	5 (+1)	10.40	39	1.05	4 (+0)	13.00	1.25	1.40	35	16			
component_prediction_5	4	2	44	4 (+0)	12.25	37	0.84	4 (+0)	12.25	1.00	1.28	25	18	1		
component_prediction_6	5	2	439	5 (+0)	13.60	205	0.47	5 (+0)	13.60	1.00	1.64	39	24	1		
component_prediction_7	4	2	138	5 (+1)	12.40	41	0.30	5 (+1)	12.40	1.00	1.91	35	22			
component_prediction_8	5	1	67	6 (+1)	11.33	88	1.31	6 (+1)	11.33	1.00	1.66	37	26	1		
component_prediction_9	5	1	67	6 (+1)	11.33	56	0.84	6 (+1)	11.33	1.00	1.88	37	26	1		
h.264 kernels																
put_h264_horiz_qpel	4	2	30	6 (+2)	8.17	29	0.97	5 (+1)	9.80	1.20	1.23	31	17			
put_h264_qpel_mc02	5	2	38	7 (+2)	10.29	28	0.74	9 (+4)	8.00	0.78	1.33	51	21	1		
put_h264_qpel_mc11_1	5	2	32	5 (+0)	13.60	22	0.69	5 (+0)	13.60	1.00	1.26	46	23	1		
put_h264_qpel_mc11_2	6	2	71	7 (+1)	12.57	82	1.15	8 (+2)	11.00	0.88	2.28	63	26	3		
put_h264_chroma_mc00	3	2	190	4 (+1)	9.25	77	0.41	4 (+1)	9.25	1.00	1.39	17	21	1		
put_h264_chroma_mc	7	2	478	10 (+3)	10.90	305	0.64	10 (+3)	10.90	1.00	2.91	69	41	3		
put_h264_qpel_mc00_4	5	2	71	6 (+1)	12.50	74	1.04	6 (+1)	12.50	1.00	1.48	50	27			
put_h264_qpel_mc20	5	2	39	5 (+0)	13.20	23	0.59	5 (+0)	13.20	1.00	0.94	47	21			
put_h264_qpel_mc01	6	2	54	8 (+2)	10.12	46	0.85	8 (+2)	10.12	1.00	2.07	56	25	1		
put_h264_qpel_mc22	6	2	100	7 (+1)	12.29	26	0.26	7 (+1)	12.29	1.00	1.85	59	29	2		
put_h264_qpel_mc21	8	2	37	10 (+2)	12.40	41	1.11	9 (+1)	13.78	1.11	1.87	81	46	1		
put_h264_qpel_mc12	10	2	153	13 (+3)	11.54	83	0.54	13 (+3)	11.54	1.00	2.69	104	47	5		
filter_mb_1	6	2	837	7 (+1)	12.14	165	0.20	8 (+2)	10.62	0.87	2.65	43	41	2	1	1
filter_mb_2	6	7	490	10 (+3)	8.90	240	0.49	10 (+3)	8.90	1.00	2.43	54	34	2		
filter_mb_3	5	1	188	6 (+1)	12.17	119	0.63	6 (+1)	12.17	1.00	2.27	35	29	4	1	
filter_mb_4	3	2	37	4 (+1)	11.00	19	0.51	4 (+1)	11.00	1.00	1.50	24	20			
filter_mb_5	7	1	507	10 (+3)	11.00	404	0.80	10 (+3)	11.00	1.00	2.77	54	44	4	1	
wrapped_idct_loop_3	4	1	84	5 (+1)	12.00	133	1.58	5 (+1)	12.00	1.00	1.75	37	22	3		1
find_frame_end	4	6	52	7 (+1)	3.86	34	0.65	7 (+1)	3.86	1.00	0.46	21	15			
ff_h264_idct_add_2	3	1	32	4 (+1)	12.00	21	0.66	4 (+1)	12.00	1.00	1.33	28	18	1		
Average							0.76			1.02						

Table 2. Experimental results of compiling numerous loops in several micro-benchmarks and full applications to two ADRES instances.

to approximate the number of registers required for the schedule. They also provide several algorithms to find an allocation using *MAXLIVE* registers. Hendren *et al.* [14] propose a hierarchical graph-based approach to reach a a global allocation (not only loops) in a short amount of time with a number of registers close *MAXLIVE* and that limits the spills that are needed.

In some cases *MAXLIVE* registers do not suffice for a valid allocation. Eisenbeis *et al.* [11] propose a graph called *meeting graph*. Using this graph they prove *MAXLIVE + 1* registers is always sufficient to find a valid allocation. Itoga, *et al.* [15], claim they have a method using *spiral graphs* to find a valid allocation using the minimum number of registers in polynomial time. However, the paper carrying the proof [13] is in Japanese, which we cannot read.

On architectures without rotating registers, Eisenbeis, *et al.* [11] proposes applying some of the above techniques on unrolled loops. Although that proposal is as efficient as the original techniques in terms of register usage, it suffers from increased code size.

Koes *et al.* [17] proposed a global register allocator that uses an expressive RF time-space representation that resembles our multi-node RF models. Their representation is called a multi-commodity network flow graph, and it is used to represents spill code optimization, register preferences, copy insertion, and constant rematerialization in one model. Just like a P&R-based code generator maps data dependencies on nets through registers, so does their method. They do not apply it in a modulo-scheduler, however, and they only apply it to more traditional architectures like the x86.

The compact RF model was already presented by Mei *et al.* in [23]. Lacking a post-pass register allocator like the TSP allocator presented in this paper, and using only the insufficient condition check $MAXLIVE \leq capacity$ during P&R, Mei's compiler could fail to produce valid schedules for some loops. The multi-node RF model and the compact RF model combined with the TSP allocator presented in this paper offer valid alternatives.

In its current state, our RRG models do not model all RF features found in modern processors. For example, pairs of registers that can be addresses as single wide registers to store SIMD vectors or wider numerical data [1] are not modeled. This is future work.

8. Conclusions and Future Work

In this paper, we have proposed a generic register file model that supports code generation, including register allocation, based on placement and routing techniques that originate from the FPGA synthesis world. Our routing resource graphs can easily be adapted to model rotating and non-rotating register files with different latencies and with different numbers of ports. Register allocation, including cluster (or bank) assignment and value replication, are then performed by the standard router, which does not need to be adapted for different register file properties or when different types of interconnects are used to connect register files to ALUs. We have also presented more compact models that can be used to perform cluster or bank assignment and value replication as part of the routing, but which require a post-pass register allocator to perform the register allocation. We presented such a post-pass allocator that is guaranteed to find a solution following a simple validity check in the router. Both the original and the compact models have been demonstrated to work well on two high ILP ADRES architectures onto which we compiled a large number of modulo-scheduled loops from multimedia and software-defined radio applications.

References

[1] AHN, M., AND PAEK, Y. Fast code generation for embedded processors with aliased heterogeneous registers. *Trans. on HiPEAC 2*, 2 (2007), 40–59.

[2] ALETÀ, A., CODINA, J. M., GONZÁLEZ, A., AND KAELI, D. Removing communications in clustered microarchitectures through instruction replication. *ACM Trans. Archit. Code Optim. 1*, 2 (2004), 127–151.

[3] BETZ, V., ROSE, J., AND MARGUARDT, A. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, 1999.

[4] BOUWENS, F., BEREKOVIC, M., GAYDADJIEV, G., AND DE SUTTER, B. Architecture enhancements for the ADRES coarse-grained reconfigurable array. In *Proc. of HiPEAC Conf.* (2008).

[5] BRIGGS, P., COOPER, K. D., AND TORCZON, L. Improvements to graph coloring register allocation. *ACM Trans. Program. Lang. Syst. 16*, 3 (1994), 428–455.

[6] CALLAHAN, D., AND KOBLENZ, B. Register allocation via hierarchical graph coloring. *SIGPLAN Not. 26*, 6 (1991), 192–203.

[7] CARPANETO, G., DELL'AMICO, M., AND TOTH, P. Exact solution of large-scale, asymmetric traveling salesman problems. *ACM Trans. Math. Softw. 21*, 4 (1995), 394–409.

[8] CERVERO, T. Analysis, implementation and architectural exploration of the H.264/AVC decoder onto a reconfigurable architecture. Master's thesis, Universidad de Los Palmas de Gran Canaria, 2007.

[9] CHAINTIN, G., AUSLANDER, M., CHANDRA, A. K., COCKE, J., HOPKINS, M., AND MARKSTEIN, P. Register allocation via coloring. *Computer Languages 6*, 1 (1981), 47–57.

[10] CHU, M., FAN, K., AND MAHLKE, S. Region-based hierarchical operation partitioning for multicluster processors. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation* (2003), pp. 300–311.

[11] EISENBEIS, C., LELAIT, S., AND MARMOL, B. Circular-arc graph coloring and unrolling. In *Proceedings of the 5^th Twente Workshop on Graphs and Combinatorial Optimization* (Twente, Netherlands, May 1997), U. Faigle and C. Hoede, Eds., pp. 71–74.

[12] GEORGE, L., AND APPEL, A. W. Iterated register coalescing. *ACM Trans. Program. Lang. Syst. 18*, 3 (1996), 300–324.

[13] HARAIKAWA, T., SOENO, M., YAMASHITA, Y., AND NAKATA, I. Register allocation frameworks for slide-window architecture. *Transactions of Information Processing Society of Japan 39*, 9 (1998), 2684–2694. (in Japanese).

[14] HENDREN, L. J., GAO, G. R., ALTMAN, E. R., AND MUKERJI, C. A register allocation framework based on hierarchical cyclic interval graphs. In *Compiler Construction* (1992), pp. 176–191.

[15] ITOGA, H., HARAIKAWA, T., YAMASHITA, Y., AND TANAKA, J. Register allocation for software pipelining with predication using spiral graph. In *Proceedings of the International Symposium on Future Software Technology (ISFST2001)* (2001), pp. 58–65.

[16] KIM, S., AND MOON, S.-M. Rotating register allocation for enhanced pipeline scheduling. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)* (2007), pp. 60–72.

[17] KOES, D. R., AND GOLDSTEIN, S. A global progressive register allocator. In *Proc. PLDI* (2006), pp. 204–215.

[18] LAM, M. S. Software pipelining: an effecive scheduling technique for VLIW machines. In *Proc. PLDI* (1988), pp. 318–327.

[19] LAPINSKII, V. S., JACOME, M. F., AND VECIANA, G. A. D. Cluster assignment for high-performance embedded vliw processors. *ACM Trans. Des. Autom. Electron. Syst. 7*, 3 (2002), 430–454.

[20] MAHLKE, S., LIN, D., W.Y., C., HANK, R., AND BRINGMANN, R. Effective compiler support for predicated execution using the hyperblock. In *MICRO 25: Proceedings of the 25th annual international symposium on Microarchitecture* (1992), pp. 45–54.

[21] MARX, D. Eulerian disjoint paths problem in grid graphs is NP-complete. *Discrete Appl. Math. 143*, 1-3 (2004), 336–341.

[22] MEI, B., VERNALDE, S., VERKEST, D., MAN, H. D., AND LAUWEREINS, R. ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix. In *Proc. of Field-Programmable Logic and Applications* (2003), pp. 61–70.

[23] MEI, B., VERNALDE, S., VERKEST, D., MAN, H. D., AND LAUWEREINS, R. Exploiting loop-level parallelism for coarse-grained reconfigurable architecture using modulo scheduling. *IEE Proceedings: Computer and Digital Techniques 150*, 5 (2003).

[24] PARK, H., FAN, K., KUDLUR, M., AND MAHLKE, S. Modulo graph embedding: Mapping applications onto coarse-grained reconfigurable architectures. In *Proc. CASES* (2006).

[25] PARK, I., POWELL, M. D., AND VIJAYKUMAR, T. N. Reducing register ports for higher speed and lower energy. In *MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture* (2002), pp. 171–182.

[26] PARK, J., AND MOON, S.-M. Optimistic register coalescing. *ACM Trans. Program. Lang. Syst. 26*, 4 (2004), 735–765.

[27] POLETTO, M., AND SARKAR, V. Linear scan register allocation. *ACM Trans. Program. Lang. Syst. 21*, 5 (1999), 895–913.

[28] RAU, B. R. Iterative modulo scheduling. Tech. rep., Hewlett-Packard Lab: HPL-94-115, 1995.

[29] RAU, B. R., AND GLASER, C. D. Scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Proc. 20th Annual Workshop on Microprogramming and Microarchitecture* (1981), pp. 183–198.

[30] RAU, B. R., LEE, M., TIRUMALAI, P. P., AND SCHLANSKER, M. S. Register allocation for software pipelined loops. In *Proc. PLDI* (1992), pp. 283–299.

[31] TERECHKO, A. S., AND CORPORAAL, H. Inter-cluster communication in vliw architectures. *ACM Trans. Archit. Code Optim. 4*, 2 (2007), 11.

[32] TOUATI, S.-A.-A., AND EISENBEIS, C. Cyclic register pressure and allocation for modulo scheduled loops. Tech. Rep. 4442, INRIA, April 2002.