4. Programming of Coarse Grain Reconfigurable Platforms

The programming techniques for reconfigurable architectures are highly dependent on several properties, including the general structure and the granularity. For most coarse grain platforms developed in the past years, an according programming environment has been published. In this chapter, several environments will be presented, examining different approaches based on the architecture requirements.

The frameworks differ in the level of the programming language. For the MorphoSys system, the MATRIX architecture, the PADDI-2 architecture and the REMARC system, the programmer has to specify the application code at assembler level. Some frameworks support the designer by a graphical tool for manual placement and routing. Although the main focus of this chapter is on automatic tools, these approaches are included in this overview for the sake of completeness.

The other published design environments feature an automatic design flow, starting from either a hardware description language, or a high-level language for the design entry. Following the basic design flow for reconfigurable architectures, these environments can be classified by the approach used for the following three steps of the design process:

- Technology mapping
- Placement algorithm
- Routing algorithm

An overview of the presented environments with the according approaches for the design steps is shown in Table 4-1. A special case is the DP-FPGA [ChLe94], for which only a simple script existed for technology mapping according to [Lewi00]. This architecture is considered no further here.

The technology mapping step, which maps the operators in the input netlist to the available operators of the architecture, is mostly simpler for coarse grain architectures than for FPGAs. Thus, one approach for technology mapping is denoted as direct, where the operators from the source file are mapped onto functional units of the architecture in a straight forward way, with one PE for one operator. A similar approach uses an additional library containing functions that cannot directly be implemented by one processing element. This approach only extends the netlist, in contrast to the more sophisticated tree matching algorithm, which is also capable to merge several operators into one functional unit. This method is also widely used in technology mapping for FPGAs. Another architecture uses an undisclosed algorithm, implemented by a special version of a FPGA design toolkit. A general exception to the mapping approaches is the Raw compiler, which does a partitioning step rather than technology mapping. This is possible, as the Raw architecture features general purpose RISC cores as processing elements, thus the mapping takes place on a higher level, assigning basic blocks from the input program to processors.

Name of System	**Programming Language**	**Technology Mapping**	**Placement**	**Routing**
PADDI	High Level (Silage)	Direct	By Scheduling	Direct
PADDI-2	Assembler	Manual	Manual	Manual
DP-FPGA	Unknown	Local	None	None
KressArray	High Level (ALE-X, a C dialect)	Direct	Simulated Annealing	Neighbor (On The Fly)
Colt	Structural Description	Direct	Genetic Algorithm	Greedy Algorithm
RaPiD	High Level (RaPiD-C)	Direct	Simulated Annealing	Pathfinder (On The Fly)
MATRIX	Assembler	Manual	Manual	Manual
Garp	High Level (C)	Tree Matching	Tree Matching	Greedy Algorithm
Raw	High Level (C or FORTRAN)	Partitioning	Greedy Swap Algorithm	None
PipeRench	High Level (DIL)	Library-based	Greedy Linear Algorithm	Greedy Algorithm
REMARC	Assembler	Manual	Manual	Manual
MorphoSys	Assembler	Manual	Manual	Manual
CHESS	HDL (JHDL)	Undisclosed	Simulated Annealing	Pathfinder

Table 4-1: Properties of programming environments for existing coarse grain reconfigurable architectures

For the placement of operators, the global structure of the architecture has a big impact. An approach, which is also found often in synthesis systems for FPGAs is a heuristic placement based on a simulated annealing or a genetic algorithm. A tree matching algorithm can be considered an exception for this purpose, and it is used as the placement for the architecture (Garp) is done together with the technology mapping using this algorithm. The use of greedy algorithms normally leads to unacceptable placement results. The two cases where placement relies on simple approaches are enabled either by the linear arrangement of the processing elements (PipeRench), or by the high level of the communication network (Raw). As an exception to the general approach, the PADDI architecture uses a scheduling algorithm for resource allocation.

The routing task in the examined environments features also quite different approaches. In two cases, the routing is not done in an extra phase but integrated into the placement and done on the fly. One approach (KressArray) uses a simple algorithm restricted to connects with neighbors and targets with at most the distance of one. The other (RaPiD) employs the pathfinder algorithm, which has been developed for FPGA routing. Greedy routing algorithms are only used in cases where the routing is restricted to one dimension, or the architecture and its programming model allow data transfers without having to care for possible routing congestion. Such an architecture is Colt, which employs wormhole run-time reconfiguration. For other architectures, the results to be expected using greedy routing would be not satisfying. General exceptions to the routing approaches is the Raw architecture, which features only one high-level communication resource, so no selection of routing resources is needed, and the PADDI architecture, which features a crossbar switch having the same effect.

The programming environments are classified by their complexity. In the next subsection, four programming environments with assembler-level input are presented. Then, four systems with mapping approaches similar to FPGA synthesis are shown. In the following subsection, three environments with greedy mapping algorithms are described.

4.1 Approaches with Programming at Assembler Level

Assembler code for coarse grain architectures can be compared to configuration code for FPGAs and is therefore the lowest level of programming. Though, as reconfigurable architectures require structural code, which is represented by the placement and routing of operators, there are graphical user interfaces provided to aid the designer with the programming task. In the case of systems comprising a microprocessor with a reconfigurable coprocessor, only the reconfigurable part is considered for the classification of the programming level.

The architectures, for which the programming is done basically at assembler level are the PADDI-2, the MATRIX, and, as reconfigurable coprocessors, the REMARC and MorphoSys architecture. While the PADDI-2 is based on a crossbar for interconnect, the other three systems are mesh-based reconfigurable architectures.

4.1.1 Programming of thePADDI-2 system

For the programming of the PADDI-2 [YeRa93], [Yeun95], [YeRa95], a software system has been developed comprising software libraries, a graphical interface for signal flow graphs, routing tools, simulation tools, compilation tools and tools for board access and board debugging. Large parts of this process are done manually. The input is a specification of assembly code for each function in the signal flow graph. The programmer then manually partitions the signal flow graph using a graphical tool. This tool aids also in the manual placement and routing. As an alternative to manual placement and routing, an automated tool is provided, which is guaranteed to find a mapping, if one exists by using exhaustive methods. Due to the exhaustive algorithms used, this automatic tool uses much computation time.

4.1.2 Programming of the MATRIX architecture

For the programming of MATRIX [MiDe96], [Mirs96], [Mirs97], a macro language has been developed, which resembles a macro-assembler rather than a high-level tool. According to [Deho00], there was some work on placement and routing, but most of it pointed out weak points of the original MATRIX design.

4.1.3 Programming of the REMARC coprocessor

As the REMARC [MiOl98a], [MiOl98b] is attached to a host processor, the programming environment allows the programming of both the RISC processor and the reconfigurable architecture concurrently. The RISC processor is programmed in C using the GCC compiler. In order to make use of the REMARC architecture, the programmer adds REMARC assembler instructions into the C program at the according places. The compiler then generates assembly code for the RISC processor with the REMARC assembler instructions embedded. This code is then further processed by a dedicated REMARC assembler, which generates binary code for the REMARC instructions. Finally, the GCC compiler is used again to generate an executable with the according RISC processor instructions to invoke REMARC and the REMARC instructions embedded as binary data.

4.1.4 Programming of the MorphoSys architecture

The programming of the MorphoSys [LSLB99], [SLLK98], [LSLB00] architecture consists of a compiler for the main processor, based on the SUIF environment [HAAM96], and development tools for the reconfigurable array. The programmer has to manually partition the input code between the processor and the array by adding a prefix to functions to be mapped onto the reconfigurable part of the system. The compiler then generates code for the TinyRISC processor with instructions to activate the reconfigurable array. For the generation of configuration code, input at assembler level, containing the complete configuration information, must be provided, which is then assembled into configuration data, which can be used to simulate the architecture using a behavioral VHDL model. To generate the assembly code, the programmer can either write it manually or use a graphical user interface, which allows the specification of operations as well as data sources and destinations for each reconfigurable cell. This graphical tool can also be used to visualize simulation of the array.

4.2 Frameworks with FPGA-Style Mapping

As FPGAs were the first reconfigurable architectures, the synthesis algorithms for these devices are well-known and can often be used directly for coarse grain architectures. Thus, all three examples presented here use simulated annealing for placement, and two architectures are routed using the pathfinder algorithm [EMHB95]. The FPGA synthesis algorithms are typically computation intensive. Though, the problem complexity for these algorithms is reduced by the coarse granularity of the architectures, resulting in a bearable compilation time.

The KressArray uses the Datapath Synthesis System (DPSS) for application mapping, with a C-like input language. The compilation framework for the RaPiD system works similar. Here, it must be noted, that RaPiD relies on relatively complex algorithms, although resembling a linear array of processing elements. The Colt architecture uses a structural description of the algorithm dataflow. The CHESS array is programmed using a Hardware description language (JHDL) as input and employing standard FPGA synthesis techniques.

4.2.1 The Datapath Synthesis System for the KressArray

For the programming of the KressArray-I [Kres96], the DataPath Synthesis System (DPSS) has been developed [HaKr95], [Kres96]. The framework generates configuration code for a given datapath description from a high-level language. Due to the original target of the KressArray-I as a part of the MoM-3 Xputer machine, the DPSS is also integrated into the design environment for the MoM-3. A compiler for a subset of C [Schm94] partitions the application into sequential code, describing the data access sequences, and structural code, resembling the data path. While the sequential code goes to the MoM-3, the structural code is mapped onto the KressArray. The more general CoDe-X approach [Beck97] employing the MoM compiler allows the input of applications in a superset of the C language. This system partitions the application onto the host and one or several Xputer-based accelerators.

The main components of the DPSS are shown in figure 4-1. The input language is the high-level language ALE-X (Arithmetic and Logic Expressions for Xputers). This language allows the specification of datapaths with assignments, local variables and loops with few variables. The next processing steps include a front end, logic optimization, technology mapping, placement and routing, and I/O scheduling. The output is a mapping of the application onto the KressArray. An example mapping of eight assignment statements as produced by the DPSS is shown in figure 4-2. In a post-processing step, this mapping has to be run through an assembler to generate the final configuration binary. Also, a schedule for the data I/O is generated. The operator repertory of the rDPUs is kept flexible by storing them in a separate operator library. In the following, the process steps of the DPSS are sketched.

After the front end has analyzed and parsed the input file, it generates an expression tree of the application, which is used in all further steps. Next, some logic optimizations are performed, including the removal of dead code and common subexpressions, constant folding, or vectorization. In the technology mapping step, the operators of the input file are matched with the available operators in the operator library, resulting in a netlist representation of the application. The following placement and routing assigns the operators to individual rDPUs. Unlike other approaches, placement and routing is done concurrently in the same step using a simulated annealing algorithm. The routing is restricted to direct neighbor connections and neighbor connections with one intermediate rDPU. All other connections are routed using the global bus. The resulting mapping is the base for the I/O scheduling step, which generates an optimal sequence for the data words over both the serial global bus and the external data bus. While the global bus schedule is used by the KressArray-I controller unit, the external bus is controlled by the MoM, which uses the according schedule for address generation. The I/O scheduling performs also

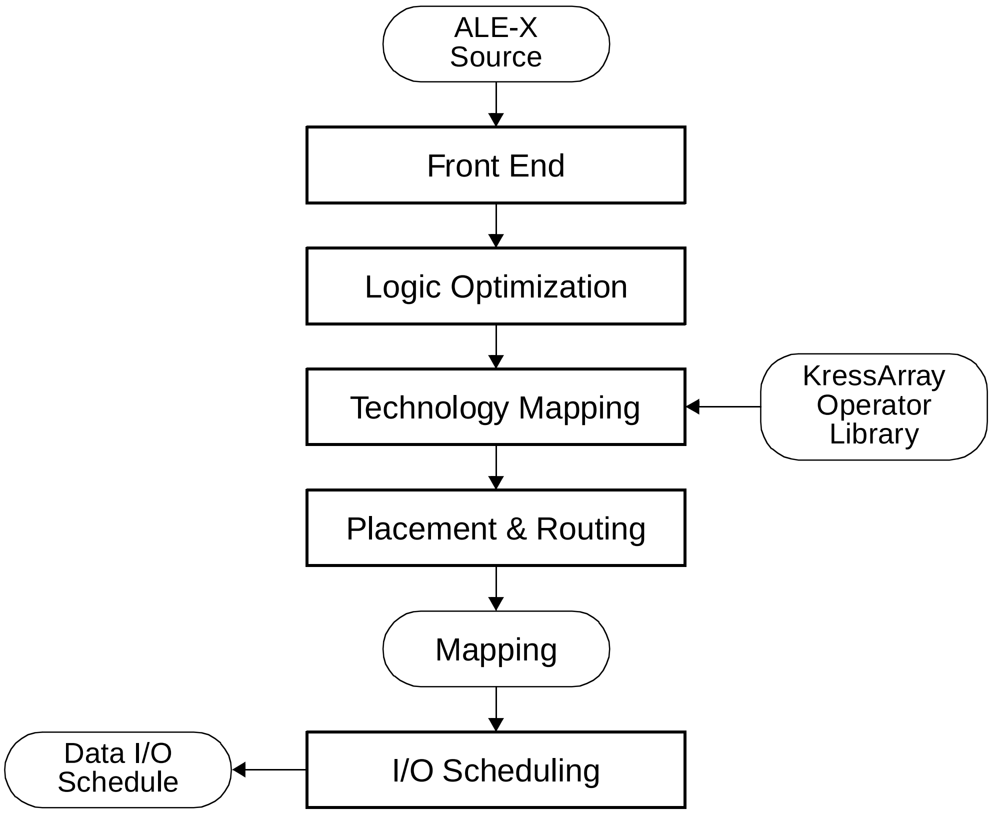
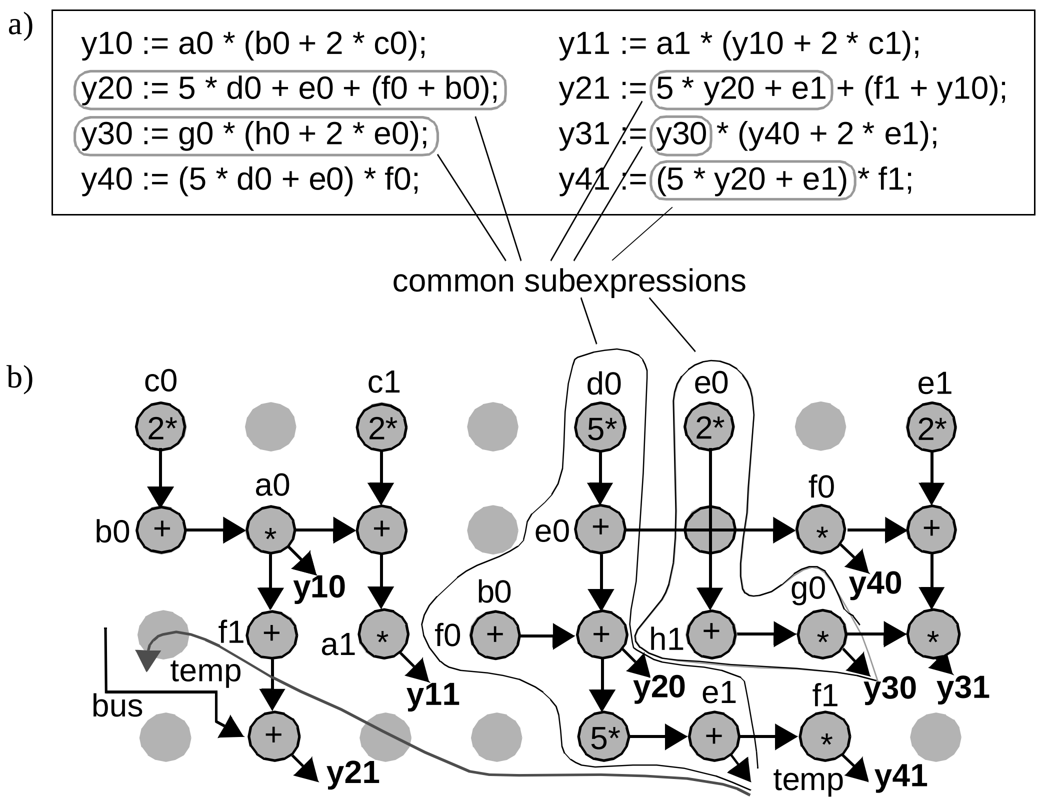


Figure 4-1: Main components of the KressArray-I DataPath Synthesis System

several optimizations like loop folding and memory cycle optimization. The latter tries to save memory cycles at run time by storing data in the register file of the rALU controller.

4.2.2 Programming of the Colt architecture

Unfortunately, details of the programming of Colt system [BAM96], [BiAt97] are not published. According to [Atha00], there exist tools accepting a dataflow description of an application and perform a placement and routing onto the architecture. The description is below C level, describing the algorithm to be mapped structurally. As the configuration pathway of a data stream is deterministic and has to be known before the execution, a placement and routing step is necessary to ensure correct execution. The placement is done by a genetic algorithm, followed by a routing phase employing a greedy algorithm.



a)

```
y10 := a0 * (b0 + 2 * c0);          y11 := a1 * (y10 + 2 * c1);
y20 := 5 * d0 + e0 + (f0 + b0);     y21 := 5 * y20 + e1 + (f1 + y10);
y30 := g0 * (h0 + 2 * e0);          y31 := y30 * (y40 + 2 * e1);
y40 := (5 * d0 + e0) * f0;          y41 := (5 * y20 + e1) * f1;
```

Figure 4-2: Example DPSS mapping of eight assignments; the shaded regions denote common subexpressions:
a) Assignment statements in Pascal notation
b) Resulting mapping

4.2.3 Programming of the RaPiD architecture

The programming of the RaPiD [ECFF96], [ECF96], [CFFF99] architecture is done in RaPiD-C, a C-like language which requires the programmer to explicitly specify the parallelism, data movement and partitioning using special extensions, like a signal-wait mechanism for synchronization and conditionals to identify the first or last iteration of a loop. RaPiD-C programs may consist of several nested loops, which describe pipelined applications to be mapped onto the architecture. The outer loops are transformed into sequential code for the address generators, while the innermost loops are transformed into structural code for the RaPiD architecture. The compilation process takes place in four steps: generation of the netlist for the structural code, extraction of dynamic control, generation of instruction streams for the programmed controller, which form the dynamic control and generation of the I/O configuration data for the stream units. While the I/O configuration data and the instruction stream can directly be used, the netlist is mapped onto the RaPiD architecture using techniques including pipelining, retiming, and place and route. According to [Ebel00], the placement is done using a simulated annealing algorithm, with the routing performed on the fly to measure the quality of the placement. For the routing, the congestion-driven pathfinder algorithm [EMHB95] is employed.

4.2.4 Programming of the CHESS array

According to [Mars00] the CHESS [MVS99] architecture is programmed using a version of the Java-based hardware description language JHDL [BeHu98], [HuBe99] and its according CAD framework to generate CHESS netlists. JHDL allows to specify hardware objects using the syntax of the Java programming language. The version employed for the CHESS array is not available for the public as of the time of this writing. The placement process is done using simulated annealing with a force-directed algorithm for local optimizations. The routing is performed using the Pathfinder negotiated congestion algorithm [EMHB95].

4.3 Other mapping approaches

Greedy algorithms for the mapping of applications are uncommon for FPGAs, as with simple algorithms, a poor mapping result could be expected. In fact, the four architectures, which employ non-standard mapping techniques in their programming environments have architectural features supporting a simple approach for placement and routing.

The Garp architecture has been classified as a mesh-based architecture before. However, for the mapping of applications, the reconfigurable mesh is treated more like a linear array. This allows the concurrent performance of technology mapping and placement in one step and a simple greedy routing algorithm. The Raw architecture features only one communication resource, so the problem of wire selection, which is normally done in the routing step, does not appear at all. Instead, the compiler has to generate a transfer schedule for the high-level time multiplexed neighbor connections. The second advantage of Raw, which allows simple mapping algorithms, is the use of general purpose processor cores in the processing elements, thus allowing to map whole blocks of the source program onto one PE. The PipeRench architecture resembles a linear array and is meant to implement application pipelines, thus keeping the placement process simple. Due to the restricted communication resources, which do not allow backward connections over more than one stage, the routing can be done using a greedy algorithm. As an exception to the general mapping approach, the programming environment for the PADDI architecture has also been included in this section, as it does not use the standard placement and routing approach.

4.3.1 Programming of the Garp architecture

Garp [HaWa97] has an advanced programming environment using C as input language and generates code for the MIPS main processor with embedded configuration code for the reconfigurable array. The reconfigurable hardware is used to accelerate loops in the C input program. Hereby, only non-nested loops can be handled. The compilation process starts with the parsing of the input program using the SUIF compiler [HAAM96] front end. In the next step, a basic block representation of the code is generated. The basic blocks are then formed into so-called hyperblocks, which contain a contiguous group of basic blocks, usually including basic blocks from different alternative control paths. Any control flow between the blocks inside a hyperblock is converted to a form of predicated execution. Then, certain blocks, which cannot be mapped onto the architecture, are removed from the hyperblocks. The resulting reduced hyperblock is then the base for mapping onto the hardware. In the next step, interfacing instructions for the main processor to invoke the mapped hyperblock are generated, and the hyperblock itself is transformed into a data flow graph.

In order to map the data flow graph onto the Garp architecture, the proprietary Gama tool [CaWa98] is used. This tool executes both the technology mapping and placement step using a tree covering algorithm which runs in linear time and is implemented by a modified tool of the lcc compiler [FrHa95]. This approach preserves the datapath structure, allowing to exploit specialized features in the Garp architecture like the carry chains. The algorithm first splits up the dataflow graph into several trees. In the following step, the trees are covered with module patterns, which resemble the possible modules that can be implemented in one Garp row. The result of this step is a number of modules, each of which can be typically mapped onto one row of Garp and may contain several nodes of the original data flow graph. During the tree covering, the modules are also placed in the array. This is done in a linear style by simply abutting the current module with the best cover of the fan-in nodes of this module, thus producing a linear layout. After some optimizations trying to correct inefficiencies caused by the separation of the input trees, the configuration code is generated. This code is assembled into binary form and linked together with the C object code for the main processor to form the final application.

The generation of the configuration code includes also the routing process, which is sketched briefly in the following, according to [Haus00]. Only the vertical wires are routed, as a row of processing elements is viewed as the unit for reconfiguration. For vertical routing, each column of processing elements is routed independently. Wire pressure is noted at every position in the column, and then a greedy algorithm assigns logical connections to physical wires, starting with connections where the wire pressure is greatest.

4.3.2 The Raw compiler

Due to the simplicity of the Raw [WTSS97], [AABF97], [Tayl99] architecture, many mechanisms implemented in hardware in current microprocessors are realized in software, putting a high responsibility to the software environment [BLAA99], [LBFS98], [MFLA99]. The Raw software environment features both a compiler and a run-time system. The run-time system manages dynamic mechanisms like branch prediction, data caching, speculative execution, or dynamic code scheduling, which are necessary for an efficient execution. The compiler handles issues like resource allocation, exploitation of parallelism, communication scheduling, and code generation. Thus the code loaded finally into the architecture consists of both the run-time system and the code for the application itself.

The global approach consists in dividing the program execution into coarse-grain parallel regions, with each region executed on multiple tiles. The regions communicate using dynamic messages, while inter-region communication is based mostly on the static network. The scheduling of this static network has to be free of deadlocks as well as efficient in a way that network stalls should be avoided. The compiler generates code for both the main processors and the switch processors in each tile. The compilation of applications for the Raw system takes place in several phases. A simplified overview on the compilation process is shown in. The phases are sketched briefly in the following.

The compiler is based on the SUIF compilation framework [HAAM96]. The input program can be specified in C or Fortran. In the first phase, traditional compiler optimizations are performed like memory disambiguation, loop unrolling or array reshaping. The pointer analysis phase [BLAA99] partitions the application data onto the distributed memory formed by the data memories in the tiles. This step includes an analysis of potential data dependencies and the generation of memory accesses, which are know at compile-time and do not need to be taken care of during run-time. This process is supported by code transformations and advanced data partitioning. The next step performs analysis for the realization of data caching [MFLA99], which is also done completely in software. The following space-time scheduling [LBFS98] parallelizes the computation across the processors, using data distribution, dependency and serialization information gained in the previous steps. As this phase contains also the mapping of the application onto the architecture, it will be shown in more detail.

The sub-steps of the space-time scheduling work on single basic blocks of the program. In the first step, code transformations are performed to bring the basic block into a form suitable for subsequent analysis steps. Also, a dependence graph for the basic block is created. The instruction partitioning step partitions the original instruction streams into multiple streams for each tile without binding it to specific tiles. The data partitioning step does the same for the data,

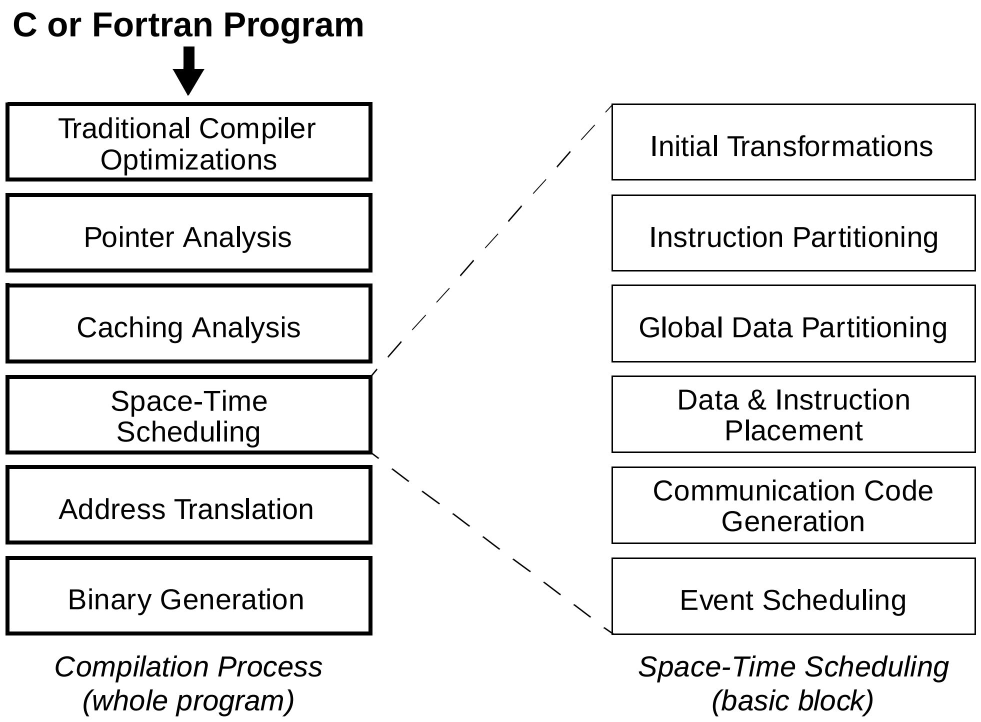


Figure 4-3: Overview on the Raw compilation process

grouping data elements into sets, each of which is to be mapped onto the same processor. In the following data and instruction placement, the data sets and instruction streams are mapped onto physical processors. The placement of data is driven by those data elements corresponding to fixed memory accesses. For the instruction placement, a swap-based greedy algorithm to minimize the communication bandwidth is used. This algorithm first arbitrarily assigns clusters of instructions to arbitrary tiles and looks for pairs of mappings that can be swapped to reduce necessary communication hops. Using the resulting mapping and the dependence graph, the next phase called communication code generator creates instructions for routing of data between tiles, where this is needed. Finally, an event scheduling of all computation and communication events is performed to produce the minimal estimated execution time.

After the space-time scheduling, an address translation step determines the necessary address translations for the software-based caching mechanism. Finally, the Raw binary is generated using the MIPS compiler backend.

4.3.3 Programming of the PipeRench architecture

For the programming of the PipeRench [CWGS98], [GSMB99], [BuGo99], the DIL language has been developed. DIL is a single-assignment language, which is intended to serve both as an input language to be used by programmers and as an intermediate language for other high-level language compilers. Although DIL has been developed for the PipeRench architecture, it is hardware independent. In the following, the main steps of the DIL compiler are described briefly. First, the architecture description of the target hardware and the source program are read and parsed. Then, the compiler inlines all modules, unrolls all loops and generates a straight-line single-assignment program, which is converted into a graph for further processing. After several optimizations like dead code removal, common sub-expression elimination, algebraic simplifications, etc., a placement and routing step is performed. For the placement and routing, a greedy algorithm has been developed, which allows a fast mapping of an application onto the architecture. The algorithm first breaks up the graph into pieces fitting on one stripe. Then, it is assured that the graph is routable by eventually inserting no-operation nodes. After the preliminary steps, the actual placement and routing is performed by placing the nodes one by one, using a list scheduling of the topologically sorted graph. Hereby, the algorithm tries to add nodes to stripes if possible. Once a node is placed, it is routed using the crossbar switch in each row. A placed node is never moved again.

The placement and routing for the PipeRench can be accomplished by this fast greedy algorithm, as the architecture has several features simplifying this task compared to other architectures. First, as the hardware resources are virtualized, no care has to be taken to make the application fit onto the architecture. Second, the crossbar switch as interconnect network simplifies routing, as no path has to be found to connect a source to a destination. Third, the high granularity allows to place whole operators in one piece. Fourth, the architecture is restricted to pipelined applications with unidirectional data flow.

4.3.4 The CADDI system for the PADDI architecture

In [CGNP92], Chen et al. proposed a compilation framework for the PADDI architecture called CADDI (Compiler for PADDI). The base of the framework are two low level programming tools being an assembler and a simulator. Upon these tools, a system is proposed allowing the specification of algorithms in the Silage language [Hilf85], a high level language for digital signal processing applications. The CADDI environment is part of a unified rapid prototyping framework, which includes also two other systems for semi-custom and multiple programmable processor target architectures. Thus, a part of the compilation process is done by tools from those systems. Generally, the compilation system allows two design goals: Finding the minimum execution time implementation of an algorithm with given hardware constraints, or finding the minimum hardware implementation with a given timing constraint. The whole system allows user interaction.

As a first step, the silage specification is compiled into a control / data flow graph (CDFG), which is used by the subsequent steps. The graph consists of nodes representing operations as well as data and control edges representing a corresponding precedence between nodes. As design decisions are also taken during the process, estimations of the critical path, minimum and maximum bounds for hardware for a given time allocation and minimum bounds for the execution time for a given hardware are determined. Also, transformations to the CDFG can be applied, including pipelining, retiming, algebraic transformations, loop unrolling and operation chaining, the latter merging several operators of the CDFG into one single operator. Besides this transformation, the technology mapping in CADDI is directly done from the CDFG to the architecture.

The further compilation path includes a partitioning step, if several PADDI clusters are involved, then a resource allocation, assignment, and scheduling phase. In the resource allocation phase, hardware resources and the number of clock cycles are determined and allocated. If hardware is minimized with a time constraint, hardware resources are added until the schedule is feasible. If time is minimized with a hardware constraint, the allocator increases clock cycles until a schedule is possible. In the assignment phase, operations are mapped to EXUs of the PADDI chip(s). This is done based on the rejectionless antivoter assignment algorithm [PoRa89], which tries to distribute the operators of the CDFG onto the available EXUs. Once an assignment is accepted, a scheduling is done to optimize speed. After those compilation phases, the CDFG is mapped onto the PADDI chip(s) using the assignment and the scheduling informations. The mapping is straight forward, as all components are fixed and the routing does not underlie limitations due to the crossbar switch. For the allocation and assignment phase, another approach besides the CADDI compiler has been proposed in [Chen92]. This approach uses a two-level hierarchical clustering algorithm, coupled with simulated annealing. However, also this approach does only an assignment of the operators to EXUs. There is no traditional spatial placement or routing.