

Macro-Based Hardware Compilation of Java™ Bytecodes into a Dynamic Reconfigurable Computing System

João M P Cardoso
INESC/University of Algarve

Horácio C Neto
INESC/IST

INESC, Rua Alves Redol n. 9, 3rd. floor
1000 Lisboa, Portugal
Phone: +351 1 3100288 Fax: +351 1 3145843
{Joao.Cardoso, hcn}@inesc.pt

Abstract

This paper presents a new approach to synthesize to reconfigurable hardware (HW) user-specified regions of a program, under the assumption of “virtual HW” support. The automation of this approach is supported by a compiler front-end and by an HW compiler under development. The front-end starts from the Java bytecodes and, therefore, supports any language that can be compiled to the JVM (Java Virtual Machine) model. It extracts from the bytecodes all the dependencies inside and between basic blocks. This information is stored in representation graphs more suitable to efficiently exploit the existent parallelism in the program than those typically used in high-level synthesis. From the intermediate representations the HW compiler exploits the temporal partitions at the behavior level, resolves memory access conflicts, and generates the VHDL descriptions at register-transfer level that will be mapped into the reconfigurable HW devices.

1. Introduction

There is consensus in the research community that the reconfigurable computing (RC) concept needs best HW compiler tools [1] to manage the HW complexity nowadays possible to integrate in a Reconfigurable Processing Unit (RPU) [2]. The RC systems must integrate the best research efforts in HW/SW codesign [3], SW compilation [4], and high-level synthesis (HLS) areas [5].

RC architectures provide the capability for spatial/parallel computation and so can achieve better speed-ups on program execution. However, compilers able to exploit the full potential of the available massive parallelism are needed. Since the proof-of-concept in [6], there have been few research efforts and consequently few advances in architectural compilation targeting RPUs. In [7] HW synthesis, spatial scheduling and partitioning are integrated for compilation of C programs into multi-FPGA devices. In [8] a static-list scheduling approach suitable to dynamic reconfiguration, which starts from data-flow graph representations, is presented. Some authors have adapted traditional HLS frameworks to the architectural synthesis of RC systems [9].

Many research efforts have been tailored to architectures of processing elements implemented in currently available RPUs, and to physical coarse-grain architectures of new RPUs [10]. These efforts try to avoid some well-known problems of the fine-grain approaches, such as the P&R steps and the area/delay prediction.

Typical HLS techniques exploit efficiently the sharing of resources supported by the layout flexibility of ASICs. However, they face some problems with pre-defined architectures such as is the case of the RPUs. The sharing of registers and some functional units can often produce poor results, because in the majority of RPUs the HW size of a MUX is the same as the area of an adder or a register. The re-use of a register, based on the variables' lifetime, needs control overhead and complicates the routing step. Moreover, including registers in all of the interconnections between scheduled steps can degrade the critical path. In the majority of the available architectures of RC systems the HW compilation has to consider the use of the local memory attached to the RPU. The front-end must exploit the distribution of the data based on its lifetime and locality according to the specific characteristics of the target architecture. These synergies are efficiently exploited by the new HLS framework, which is described in this paper, in order to produce the most adequate HW solution to be executed by an RPU.

The concept of “virtual HW” (VW) [11] has been made possible by RPU architectures with short reconfigurable times and partial-reconfiguration support. The HW compilation must integrate temporal partitioning schemes to automatically exploit this new capability. The temporal partitioning resolves the implementation of an HW region that does not fit into an RPU by time-sharing of the device in a way that each partition fits in the HW resources. The new algorithms presented, integrate the temporal partitioning schemes in the HW compilation steps at the behavioral level, in order to efficiently exploit the VW concept.

The efforts of HLS from SW programming languages have been majority tailored to C [12][13]. However, the low level used in the C language to manipulate memory contents, such as the use of pointers to fine-grain physical loca-

tions, can hinder static resolution and incapacitate a significant number of optimizations. The exploitation of data storage at higher levels of abstraction is possible with the Java programming language, which is strongly typed, and uses references with the atomicity of objects or arrays. [14] describes the GALADRIEL front-end compiler that starts from the Java *bytecodes* [15] obtained from a Java program or from other source. The selection of *bytecodes* has the following advantages: the model is platform independent, executable, there are many available tools for free, is object-oriented, retains almost all the source’s information and semantic, uses references with the atomicity of object, and supports multi-SW-languages.

In this paper the integration of HW compilation for RC systems in the GALADRIEL is explained. The HW compilation starts from the intermediate graph models outputted from the front-end and integrates temporal partitioning before the schedule of memory accesses and communication between temporal partitions. It uses graph models that permit the exploitation of inter-block parallelism (execution of independent blocks of the graph at the same time) and the exploitation of intra-block (inter-operation) parallelism (simultaneous execution of multiple operations). These graphs permit the fully exploitation of the freedom to reorder the basic blocks (BBs) due to the lack of data-dependencies and permit that the scheduling phase defines a new order of BBs in the case of resource sharing between conditional paths.

Section 2 explains our approach to HW compilation from the Java *bytecodes*. It describes the representation graphs obtained with the front-end (correspondent to the first level of the complete flow stretched in Figure 1), and the graphs used in the HW compilation phase (second level in Figure 1). Section 3 presents the mapping and scheduling schemes proposed which permit more than one functionality-equivalent component. Section 4 explains the integration of temporal partitioning and the developed algorithms. In section 5 proof-of-concept examples are shown. Finally, conclusions and further work are identified.

2. Hardware Compilation Flow

Previously, we have used an existent HLS tool (the GALADRIEL compiler generates VHDL [16] behavioral processes [14]). However, this approach was found not suitable for RC and therefore it has been decided to develop a new tool called NENYA (with the first achieved results presented here), specially suited to support the RC concept. It computes temporal partitioning from the graphs generated by GALADRIEL and produces VHDL descriptions at the RTL (register-transfer level) for each temporal partition. Figure 1 shows the compilation flow of the *bytecodes* of a given application to SW and HW.

The bounding boxes in Table I summarize the subset of the Java language with correspondent *bytecodes* currently sup-

ported by NENYA. All the supported types can also be used only with the supported operations.

Table I. The Java subset which correspondent *bytecodes* are currently supported is shown inside boxes (♣ only uni-dimensional arrays).

Types	Operations	Control & other mechanisms
Boolean	/, %	Exception-handling
Byte	*	Method invocation
Short	++, --, +, -,	Creation of objects
Int	<<, >>, >>>,	Creation of arrays♣
Long	, &&, &,	While, for, do while
Float, double, char	!, ^, ~,	Break, continue
Reference to arrays♣	<, >, <=, >=,	If, ? :, switch
Reference to objects	=, !=	Casting conversion

The NENYA has as inputs the Control Dependencies Graph (CDG), the Data Dependencies Graph (DDG), and a Data Flow Graph (DFG) for each BB. All these graphs are extracted from the Java *bytecodes* of a given application with the GALADRIEL front-end compiler. The CDG and the DDG have been previously used in SW compilation [17] and their use in HW/SW partition has already been proved efficient [18].

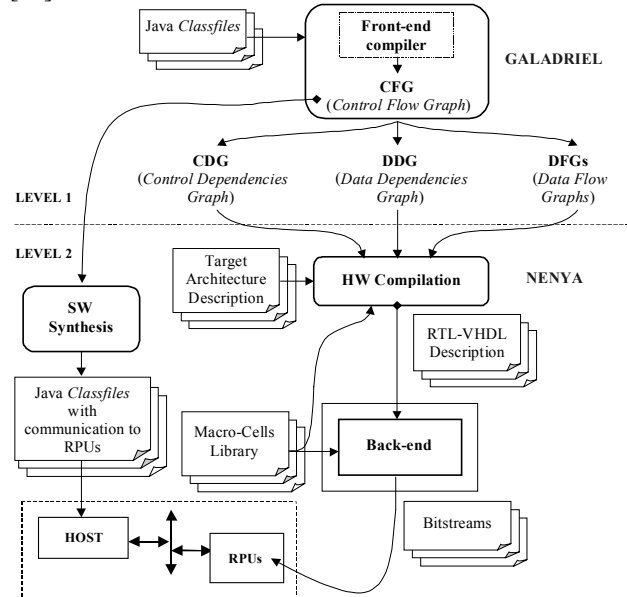


Figure 1. Compilation flow from the Java *classfiles* to reconfigurable computing systems.

At the moment the scheduling scheme does not consider the sharing of resources implementing an arithmetic/logic operation (unless the sharing does not need a control unit). Each arithmetic/logic operation in the DFG is mapped to a macro-cell of the library that implements an equivalent functionality. The mapping is based on the mobilities of the nodes in the DFG as is explained later. The scheduling task resolves the memory accesses, the conflicts that may occur in bus sharing architectures, and the communication be-

tween temporal partitions. Moreover, the compiler generates a description of the control unit of the runtime scheduler of reconfigurations to the same RPU, at the moment implemented with SW.

The structural VHDL generated by NENYA represents the interconnection between units contained in a library of relatively placed macros. This library contains a minimal set of macro-cells to support the subset of JVM instructions. The VHDL representation of these macro-cells includes specific attributes to specify the relative locations and shapes of the corresponding HW, which are taken into account by the P&R tool. The back-end of NENYA is shown in Figure 2. At the moment this back-end is the one used with the XC6000 Xilinx series of RPUs [19].

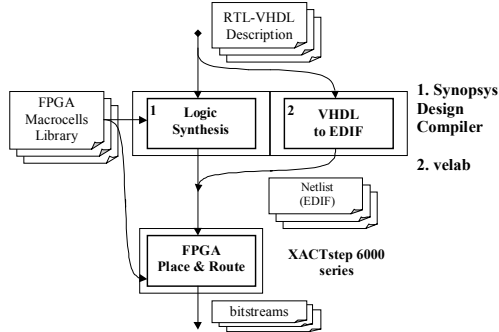


Figure 2. The back-end synthesis flow into an RPU.

1.1 The graph intermediate representations generated by the GALADRIEL front-end

The first step of the GALADRIEL is to disassemble the JVM instructions from the *bytecodes* of the given *classfile*. Then, a Control Flow Graph (CFG) [20] for the selected method is generated. The CFG is a directed graph $G=(N, E)$, with each node N representing a JVM instruction and edges E representing the execution dependency between instructions. Each CFG includes a START and an END node (all other vertices are included in a path between the two vertices). There is an edge between two nodes (N_1, N_2) if and only if the instruction in N_2 can be executed immediately after the instruction in N_1 . Unlike traditional CFGs, this model of CFG includes switch type vertices (to support the *tableswitch* and *lookupswitch* of the JVM [15]) that redirect control flow to one of multiple outgoing edges based on the value of the variable evaluated by the switch, and captures the exception handling routines.

To analyze the data and control dependencies of a method at the BB level, the CFG is transformed in the CFG_{BB} , in which the new vertices are BBs of JVM instructions. A BB integrates a sequence of consecutive JVM instructions in the CFG in which flow of control enters at the beginning and leaves at the end without the possibility of branching or throwing an exception, except at the end, and without the possibility to start except at the first instruction.

From the CFG_{BB} , the Dominators Tree (DT) is constructed and the existent natural loops are identified and extracted using conventional compiler techniques, as described in [20]. After the identification of the related loops information (bounded, unbounded, counter variable, incremental value, initial value, and final condition), each natural loop is grouped in a special node. In the presence of nested loops the set of BBs of an outer loop contains the equivalent set of the inner loop and so on.

The Postdominators Tree (PDT) is constructed with the same algorithm of the DT in the reverse traversing order of the CFG_{BB} . Then, the CDG [17], [21] is obtained from the CFG_{BB} and the PDT. A node N_1 is control dependent of node N_2 , if N_2 determines whether N_1 is executed. A CDG, $G(N, E)$, is a direct graph with each node N representing a BB and edges E representing control-dependencies between BBs. Each CDG includes a START node, and every outermost conditional BB has an edge incoming from the START node. Every innermost conditional BB has an incoming edge from the next outermost one in the hierarchy of control-dependencies.

The CDG and the DDG are fundamental to evaluate the implementation of massive parallelism in a true concurrent model (HW). The construction of the DDG considers the utilization of local variables, the use of the operand's stack, the memory alias, the accesses to array elements, and the semantic dependencies (like the dependencies between a creation of an object and the call of its constructor). The data dependencies produced by the utilization of local variables and fields are determined by first solving the reaching definitions data flow problem with a general iterative algorithm [20]. A definition of a variable occurs when there exists a JVM instruction that assigns, or may assign, a value to that variable: *istore*, *astore*, etc. An use of a variable occurs in a JVM instruction that uses the value of the variable: *iload*, *aload*, etc. The use-definition chains for each use of a local variable or field are computed from the reaching definitions. This indicates whose definitions of local variables or fields reach an use.

A BB_i is data-dependent (true-dependency) of BB_j if it uses a local variable or a field last time defined in BB_j . The data-dependencies' types are encapsulated in the DDG in the form of distinct edges. The true-dependencies edges encapsulate the local variable number or field number identification and the number of bits of the variable or field. The output and the anti-dependencies are modeled with edges without labels, because these two types of data-dependencies only constrain the ordering of BBs to be executed and do not have any communication costs.

The stack-dependencies are determined traversing forward the JVM instructions in the CFG of the given program and storing for each one the information about the stack contents (operands, types, instruction that has pushed the oper-

and, etc.). This is possible because the stack model of the JVM is invariant [15], i. e. the stack contains always the same number and type of operands in the execution flow of each instruction of the CFG. After this, for each BB the algorithm only examines the stack-state of each instruction in the BB to determine its dependencies. If all the stack-states that enter in a BB exit it, then this BB does not depend on another BB unless any instruction that uses stack operands without popping them (like the *dup* instruction) is involved. In this case the algorithm examines the states involved. The algorithm also considers the propagation of constants, when examining the states in the stack.

The alias problem [4] requires similar analysis as the "available expressions" problem. We have named it "available object/array assignment" (AOA). An object/array assignment A is available at point P in a program, if for every path in the CFG from START to P there is an assignment A and no definition of the variable assigned in A between the assignment and P. For each AOA, the two variables used in the assignment expression must be treated as an unique variable during the lifetime of the assignment.

The sequence of accesses to the same reference (excluding the *this* reference) that stores an object or array address (message sends, manipulation of the fields, or array elements) are maintained as in the original CFG. The extraction of this type of possible parallelism requires a deeper inter-procedural analysis. The analysis of references currently used does not distinguish elements of the same array or of the same object (uses the atomicity of object or array).

Instead of creating a Program Dependence Graph (PDG) [17], which combines control and data dependencies in a single representation through the union of the DDG and the CDG, the hardware compiler maintains the two graphs in order to permit an efficient exploitation of pre-computation of conditional branches.

A DFG is created for each BB contained in the region to migrate to HW. This DFG has nodes representing operations, variables, constants, load and store operations, conditional nodes, or method invocations, and edges representing the flow of data between nodes. The nodes representing variables only serve as fork nodes for multiple operations using the value of the variable. These nodes of local variables in the DFG can be mapped to nets (wires), registers, or memory locations in the scheduling phases. The computation of the DFG is done using the stack contents computed by the stack-dependencies algorithm cited above.

With the use of the DFG the compiler computes the bit-width for each operation and variable used (see the example in Figure 3). For each JVM instruction of type conversion (*byte2int*, etc.) present in the sequence of JVM instructions in the BB, it sets the variable width of the result and propagates it backwards through the chains of operations in the DFG, and forward to the sink node. From the arguments of

the given Java method (explicitly represented in the *bytecodes* [15]) the algorithm tries to propagate it forwards. More advanced bit-width extraction needs to be done to improve the HW results and to permit whenever possible to extract the *boolean* type which is encapsulated in the *bytecodes* as a byte.

At the moment, the user must identify the code regions that degrade the performance of the application with the help of a profiler tool. A library for inserting tags in the Java source code has been developed, which includes the *Tag* interface with methods to specify code segments (*start*, *end*, etc) and two classes that implement it in order to specify timing constraints (*maxtime* and *mintime*). The compiler uses the calls to these methods to identify the user-defined code segments and the constraints specified. The use of this library permits the analysis of the performance of the code between tags in the host machine, by the execution of the program. The GALADRIEL identifies the region and eliminates all the references to this library in the *bytecodes* of a given method when the CFG construction is performed. To be mapped to HW the user must specify a single-entry, single-exit region of the input Java program.

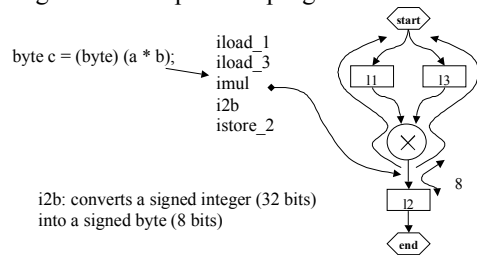


Figure 3. Computation of bit-widths from a DFG.

1.2 Representation Models for HW Compilation

Two types of models with impact in the scheduling have been used by the HLS research community: data-flow, and control-flow. Most common representations use a CFG with fork and join nodes to model control, and a DFG for each BB [22]. Some authors use a DFG for all the region, which is a representation well suited to parallelism but its level of information is too low and the fact that it mixes the control and data dependencies limits the flexibility of the compilation. Some scheduling algorithms, as the one proposed in [23], try to use the best proprieties of both models by switching between them.

The representation model proposed in this paper, maintains the control dependencies separated from the data dependencies and uses a full reordering of the BBs of the input CFG, based on their data-dependencies. Therefore, it is better suited to performance-driven implementations. It permits to explore control-flow and data-flow techniques efficiently, permits to exploit the massive parallelism of the input program, and provides a more efficient schedule by using merge points at a later stage.

We define a merge point in a different way than standard compiler theory, where a merge point serves as the join of multiple control flow edges in the CFG. In this context, merge-dependencies are the points in which selection must be inserted (in the form of MUXES, for example) because the execution depends on the branch taken in a control point (BB with two or more outgoing edges) prior to the considered merge-point. In a BB there are merge-dependencies of a control-point when the BB is data-dependent of any of the branches of the control structure (with the conditional node as the root). A node that has more than one edge of data-flow dependencies (true-dependencies) correspondent to the same variable location will be merge-dependent of the closest conditional node that is the root of at least one of the sources' nodes of the edge. When the conditional paths have side-effects (for instance the store to memory) merge-dependent points are currently inserted in the nodes that produce side-effects. The insertion of the merge-dependent points is exploited when considering the sharing of resource units between conditional paths.

Example 1 shows the representation graphs and their impact on the compilation to HW. Example 2 shows the sharing of resources in mutually exclusive execution paths and the consequent insertion of new merge points. As described, the control-dependencies, merge-dependencies, and data-dependencies are maintained in disjoint graphs (see Example 1), because in a true concurrent environment the mixture of them can limit the optimizations. The CDG is important to determine if there are operations in distinct conditional paths that can be shared and is a way to represent the control node and the control-dependent paths. In the case of resource sharing in mutually exclusive paths the conditional nodes will also produce merge-dependent points in their successor nodes in the CDG that will share resources (see Example 2).

Example 1. The representation graphs (the correspondent BB number is identified in the source).

The local JVM variable l1 corresponds to h, l2 to e, and l0 to f in the method below. The graphs obtained from the CFG of Figure 4 can be seen in Figure 5.

```

static int ex(int f, int h)
{
    int e = 0;           BB0
    if(f<4)             BB0
        e = f + h;      BB1
    else
        h = 3 * h;      BB2
        e = 3 * h + e;  BB3
    return e;          BB3
}

```

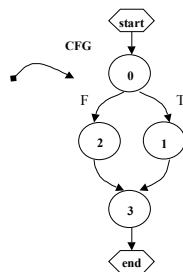


Figure 4. An example and the correspondent CFG.

Using the CFG as the input to the HW mapping, the worst execution time (WET) of the graph can be estimated as:

$T_A = T_0 + \text{MAX}(T_1, T_2) + T_3$ (Figure 4), where each T_i is the WET of the BB_i; With the use of the DDG and the MDG the exploration of maximum parallelism in the region will permit to have for the WET: $T_B = \text{MAX}(T_0 + T_1, T_2) + T_3$ (Figure 5); Considering that the delay of the multiplier is 420ns, for the adder is 107ns, and for the comparison is 110ns, then $T_A = 1,057\text{ns}$ and $T_B = 974\text{ns}$. In examples where the BBs of the conditional branches do not have data-dependencies with their conditional nodes, large improvements may be obtained.

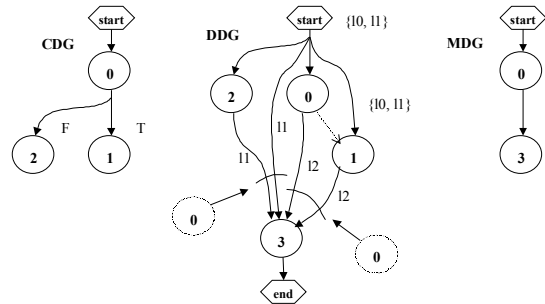


Figure 5. Graph transformations to exploit the true-concurrent hardware technology.

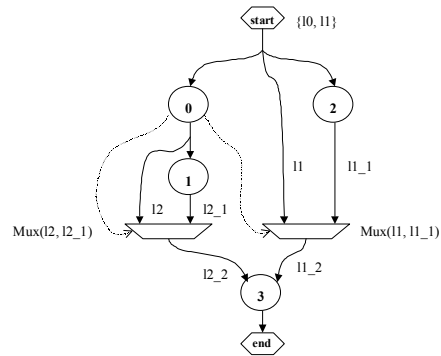


Figure 6. Graph obtained from the DDG and MDG to be mapped to hardware. ♦

Example 2. The sharing of resources in mutually exclusive execution paths.

The example below has the HW graph shown in Figure 7a.

```

...
if (f<10) a = b * c + d * e;
else a = d * c + e;
...

```

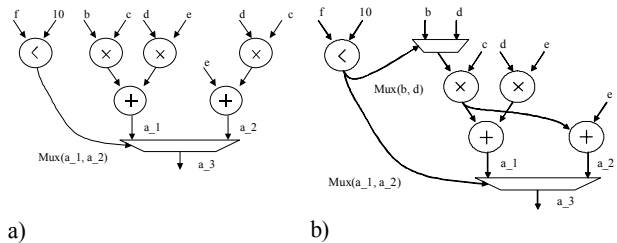


Figure 7. HW graphs: obtained from the MDG+DDG a); obtained with sharing of one multiplier in mutually exclusive execution paths b).

In mutually exclusive paths spatial sharing of each operation can be exploited. With the use of only two multipliers the implementation does not need registers and consequently does not need FSM

control as can be seen in Figure 7b. The resource sharing between mutually exclusive execution paths can reduce the overall area of the graph implementation but may introduce a delay overhead, because it needs to know the result of the comparison earlier. ♦

3. Scheduling and Mapping Techniques

To efficiently generate HW for the selected region it is necessary to map each node in each DFG to a macro-cell of the input description library [24]. Each operation node of a DFG is mapped to a correspondent macro-cell of the HW library with the minimum delay. Then, the node operations of the DFG are scheduled with fine-grained ASAP ("as soon as possible") and ALAP ("as late as possible") schemes (obtaining for each node: $ASAP_{start}$, $ASAP_{end}$, $ALAP_{start}$, and $ALAP_{end}$) without resource constraints. The scheduling is based on the delay of each macro-cell in fractions of the clock period, previously selected by the user, instead of the control steps as used by coarse-grain scheduling schemes [5]. Afterwards, the time-interval that an operation node can use without affecting the delay of the critical path is calculated. This time-interval is calculated by the equation (1), and represents the sum of the mobility and the delay of the operation.

$$TI_i = ALAP_{endi} - ASAP_{starti} \quad (1)$$

Based on the TI of each operation in the DFG, the algorithm selects among the functionally-equivalent macro-cells of the library the one with lower area that does not perturb the critical path delay (the algorithm is sketched in Figure 8).

After the intra-mapping and intra-scheduling at the level of operations in each DFG, the HW compiler must compute the scheduling at the BB level. It traverses all the nodes of the graph obtained by the union of the DDG and the MDG and applies the following pre-processing steps:

- a) Assign a level to each node based on the dependencies among nodes. A node whose inputs are primary inputs has level 1. A node whose inputs are created by nodes will have level equal to the maximum level of the correspondent levels of the source nodes plus one.
- b) Determine the $ASAP_{start}$ and $ASAP_{end}$ of each node based on the dependencies among nodes. A node whose inputs are primary inputs has $ASAP_{start} = 0$ and $ASAP_{end} = delay_{node}$. A node whose inputs are created by other nodes will have $ASAP_{start}$ equal to the maximum of the $ASAP_{end}$ of each of the source nodes, and $ASAP_{end}$ equal to $ASAP_{start} + delay_{node}$.
- c) Assign a level to each node based on the dependencies among nodes. A node whose outputs are primary outputs has level = MAXLEVEL (determined in a)). A node whose outputs are other nodes will have a level equal to the minimum of the levels of each sink node minus one.

- d) Determine the $ALAP_{start}$ and $ALAP_{end}$ of each node based on the dependencies among nodes. A node whose outputs are primary outputs has $ALAP_{end} = MAXTIME$ (calculated in b)) and $ALAP_{start} = ALAP_{end} - delay_{node}$. A node whose outputs are created by other nodes will have $ALAP_{end}$ equal to the minimum of the $ALAP_{start}$ of the source nodes, and $ALAP_{start}$ equal to $ALAP_{end} - delay_{node}$.
- e) Sort the nodes by increasing order of their levels.
- f) Determine the fine-grain mobilities for each BB. The mobility of a node is the difference between its $ALAP_{start}$ and $ASAP_{start}$ or its $ALAP_{end}$ and $ASAP_{end}$.
- g) For each level, sort the nodes by increasing order of the mobilities.

If the region does not fit into the RPU, temporal partitioning is performed (as described in the next section), and for each temporal partition, the memory accesses and the data-transfers are scheduled.

```

Mapping {
LD = LoadLibrary();
Foreach DFG  $\in$  {set of DFGs to migrate to HW} {
  Map each Operation Nodei of DFG to a
  Functional equivalent Macro-cell  $\in$  LD with
  Minimum delay;
  DelayMAX = Compute fine-grain ASAP( $ASAP_{start}$ ,
   $ASAP_{end}$ );
  Compute fine-grain ALAP( $ASAP_{start}$ ,  $ASAP_{end}$ ,
  DelayMAX);
  Compute TI( $ASAP$ ,  $ALAP$ );
  Refine Mapping(TI, DFG);
}
}
Refine Mapping(TI, DFG) {
Foreach Operation Nodei of DFG with TI  $\neq$  0 {
  S1 = Search in LD for the set of functional
  Equivalent Macro-cells of the operation
  in Nodei;
  If ((S1  $\neq$   $\emptyset$ ) && (Search Macroj  $\in$  S1 that
  is lower area and delayj  $\leq$  TI(Nodei))) {
    Nodei.SetMacro(Macroj);
    Compute the new fine-grain ASAP for the
    path which contains Nodei;
    Compute the new fine-grain ALAP for the
    path which contains Nodei;
    Compute the new TIs for the path which
    Contains Nodei;
  }
}
}

```

Figure 8. Algorithm to schedule and map nodes for each DFG.

When dealing with arrays, in this first implementation, we consider that the RPU has access to a local memory. The binding of memory addresses to store the array elements is done statically and if the array dimension cannot be found by the intra-procedural analysis the designer must help the tool. The DFG obtained from the *bytecodes* uses a variable's node that represents the array address in the memory (see Figure 9). For all the DFGs in the HW region these nodes of variables are replaced by the real addresses of the start memory location which is statically allocated for each array. The $ASAP_{start}$ and $ALAP_{start}$ of the load/store nodes cor-

respond to the beginning of a clock cycle when doing the scheduling and a time interval (the number of clock cycles needed for the macro-cell to access memory) between this clock cycle and the load into the register is added.

A scheduler scheme based on static-list scheduling, with the priority function based on the mobilities, is used to avoid conflicts in the use of the RAM macro-cell between concurrent accesses in each temporal partition and to furnish the control steps needed to interface to a synchronous component. The description of the control unit used is outputted in a tabular format to be translated to a VHDL entity and synthesized with a logic synthesis tool.

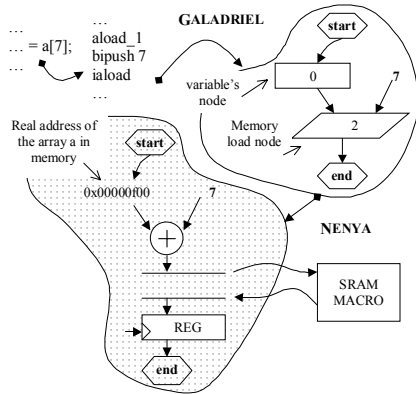


Figure 9. The transformations from the Java program to the HW graph in the presence of arrays.

The control unit selects between the addresses and between the data sources, and in the case of load operations, controls the storing of the output of the SRAM-macro-cell in the specific register. This control unit is also used to load/store data between temporal partitions.

4. Temporal Partitioning

The development of algorithms to enable the automation of the process to compute partitions of a given graph to be executed by time-sharing of the RPU was formulated in [25] and [26]. As described in [26], this problem has similarities to the scheduling problem in HLS [5][22]. The nodes of a given graph have to be scheduled in time slots to be executed in each partition. Temporal partitioning must preserve the dependencies between nodes (that are already temporal dependencies) such that a node A dependent on node B cannot be mapped to a partition executed before the partition where node B was mapped. The similarities of both problems allow the use of common HLS scheduling schemes for temporal partitioning and the integration of them in the same algorithm.

In [25] the temporal partitioning problem is modeled in a specified 0-1 non-linear programming (NLP) model. This formulation and spatial partitioning are integrated in the SPARCS HLS system [9]. However, even for small input graphs, the formulation is not always solved in reasonable

execution time. These algorithms are NP-complete and heuristic methods must be developed to permit feasible execution of large input examples. In [26] a partitioning algorithm based on the levels of nodes obtained by the ASAP scheduling algorithm is used. The algorithm fills the available area of the RPU in the increasing order of the ASAP levels. The selection of nodes in the same level is arbitrary.

Our approach integrates some temporal partitioning schemes to permit the exploration of their results and the user can select among them. In Example 3 the orientation of a partitioning algorithm by ALAP levels produces best results over the ASAP levels, because of the temporal proximity of dependencies between nodes in the ALAP levels.

Example 3. Temporal partitioning with ASAP or ALAP levels.

Suppose that all the edges have the same communication cost, each node has unit area and the maximum area of the HW device is 4 units. The temporal partitioning results of applying a version of the algorithm in [26] with ASAP and with ALAP levels are respectively shown in Figure 10.a) and Figure 10.b). For this example the ALAP levels produce better results. The partitions obtained from ALAP levels have fewer communication edges between themselves.

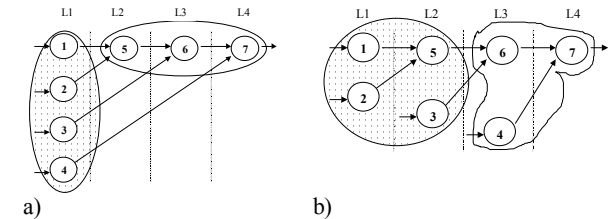


Figure 10. Two schemes of temporal partitioning: by ASAP levels a); by ALAP levels b). ♦

Two temporal partitioning algorithms have been implemented. Algorithm 1 extends the algorithm in [26] with the selection of a node, in the same ASAP or ALAP levels of the scheduling, by a local priority function. This function is applied to all the nodes of the current level, and permits to orient the selection by increasing order of the mobilities. Therefore, the critical path of the graph will have high priority on mapping the nodes to the current partition.

These algorithms switch to the next partition when a node does not fit in the current partition, without trying to map the remaining unscheduled nodes in the level. Algorithm 2 (shown in Figure 11) tries to resolve this issue, by recursively searching in the list of mobilities so that if a node cannot be mapped to the current partition, other nodes can be considered. This algorithm can also be based on the ASAP or ALAP levels and with the same priority function as algorithm 1.

To address different architectures and RPU technology, three schemes of communication mechanisms between temporal partitions will be considered:

- The use of registers and a SW control task to load and store the communication between partitions. This scheme is well suited to boards of RPU's with a processor or a micro-controller that can also control the HW reconfiguration of partitions and their communications, or to boards without local memory to store data. With the advent of the integration of RPU's in processor cores this can also be used because of the decreased communication overhead of these types of systems.
- The use of a set of registers in the RPU when the device can be partially reconfigured. The registers will be configured once in a region of the RPU and shared between temporal partitions (this can include the advent of new RPU's tailored to time-sharing).
- The use of a macro-cell to access memory locations controlled by an HW control unit. This is well suited to types of RPU's where there is no processor (only local memory) and the communication between the host and the RPU's is time-consuming.

```

TempPart(Schedul, SL, ScheduleNum, CurrentLevel,
IsALAP, NodesLevel, SameLevel) {
  If(CurrentLevel <= getMaxLevels()) {
    SameLevel = SortMobility(SL, CurrentLevel,
IsALAP);
    If(NodesLevel == null) {
      NodesLevel = new Stack();
      Push the Nodes of the current level onto
      The stack NodesLevel;
    }
  }
  Foreach Unscheduled Node of CurrentLevel
  Sorted by mobility {
    NodesLevel.peek(Nodei);
    If(Area(Nodei) + Sched.Area() < AMAX) {
      Put(Nodei) in current schedule;
      Update Sched.Area;
      NodesLevel.Pop(Nodei)
    }
  }
  If(NodesLevel not empty) {
    ScheduleNum++;
  } else {
    NodesLevel = null;
    CurrentLevel++;
  }
  TempPart(Schedul, SL, ScheduleNum, CurrentLevel,
IsALAP, NodesLevel, SameLevel);
}

```

Figure 11. Algorithm for temporal partitioning oriented by the mobility of the nodes and with search within the same level of ASAP or ALAP.

5. Experimental Proof-of-concept

At the moment the whole framework has about 20,000 lines of code in the Java programming language.

The NENYA generates VHDL hierarchical RTL descriptions. At the top level each entity related to a temporal partition is defined. Each temporal partition entity has a level describing each BB (a DFG) in terms of the interconnections be-

tween macro-cells of the HW library. This hierarchical description has direct correspondence to the hierarchic graph representation used during the HW compilation and can therefore be a user's guide in the manual refinement of the P&R tasks, and to permit the back-annotation into NENYA of more realistic area and time proprieties for each entity. The use of the DOTTY [27] tool to visualize the graph representations with the same labels between the graphs and the VHDL signals and instances can also be an important help mechanism in the P&R phases.

A library of macro-cells (actually circuit-generators) [24] has been implemented which permits NENYA to target the Xilinx XC6000 series of FPGAs [19]. The library includes integer operators (arithmetic, logical, comparison, shifters by constants, barrel shifters, etc.), a macro-cell for memory access, MUXES, etc. Each component is declared and characterized (such as the one shown in Figure 12) in a specific file. The NENYA supports the parsing of the file and the loading of the components into an intermediate format suitable to be processed during the mapping phase.

```

...
Component mult_op {
  Area=(a+b)**2; //a and b are the number
                //of bits of each operand.
  Latency=14*a-20ns;//delay of the macro-cell
                //as a function of a.
  Operation=imul; //Type of the function
  ...           //imul=integer multiplier
}
...

```

Figure 12. Description of a macro-cell in the library.

Several test cases have been explored. To validate the complete HW/SW approaches, boards of RPU's connected to the PCI have been used. A Hamming decoder, a Binary Pattern Image Coding (BPIC), and the HAL example [22] are considered here. The Hamming example has 121 JVM instructions grouped in 6 BBs and is used to compare our approach to logic synthesis and to "traditional" HLS (T-HLS). Both have been applied directly to the behavioral VHDL automatically obtained from the *bytecodes* of the example [14]. Table II shows the obtained results. Table III shows the macro-cells used by NENYA. The use of more specialized macro-cells (operations with constants) gives an area of 237 cells and the delay of the critical path decreases to near 140 ns. The logic synthesis with the Synopsys™ Design Compiler [28] results in an area of 41 cells and a delay of 52 ns (see LS in Table II) due to the bit-optimizations performed by the tool. Although logic synthesis can be applied to this example, it cannot be used in larger examples because it is too inefficient to generate a flatten-structure. Moreover, when the behavior includes memory accesses and loops a true-logic synthesis approach is not suitable or even feasible.

The use of an academic HLS tool results in an estimated HW area of 1,080 cells, instead of the 321 cells obtained

with the NENYA. This result is explained by the utilization of numerous registers and MUXES. Its P&R is too time-consuming and was not considered. These results do not consider any logic optimizations of the RTL structure obtained from either the HLS or the NENYA tools.

Table II. HW Results for the Hamming decoder. (♣ without the routing overhead; ♠ measured with the XACTstep tool [29]; REC-reconfiguration time).

Approach	Area (cells)	Delay (ns)		address-data pairs	X × Y cells	REC (us)
		♣	♠			
NENYA	321	50	160	946	24×32	28.67
LS	41	19.6	52	127	7×8	3.85
T-HLS	1,080	-	-	-	-	-

Table III. Macro-cells used by NENYA for the Hamming decoder (8 bit's data types).

Macro-cell	Description	Quantity
AND_op	Logical AND	8
CONST_op	Constant	16
OR_op	Logical OR	2
XOR_op	Logical XOR	13
SHR_c_op	Logical shift right by a constant	6
SHL_c_op	Logical shift left by a constant	2
MUX_5_1	Multiplexer with 5 entries	1
IF_EQ_op	Comparison (equal)	4

The example has an SW execution time of 5,21μs and 0,5μs with the JDK1.1 interpreter and a JIT respectively (executed in a PC with WinNT, 64MB RAM and Pentium @133MHz). The results show speed-ups of 32.5 and 3,1 of the HW solution obtained from NENYA over the interpreter and JIT respectively. The HW/SW version does not improve the performance of the SW version, due to the communication costs between the host and the board attached to the PCI.

The BPIC is an algorithm that decomposes an image in blocks of 4x4 pixels and encodes it in a single word. Here we consider part of the algorithm that computes the encoding based on the luminance and the mean of the luminance of the pixels in the block. Figure 13 shows the part of the algorithm considered and the code segment (which is the innermost loop unrolled) identified with “tags” to be compiled to HW. The segment has 58 JVM instructions grouped in 12 BBs. The NENYA assumes the mapping of the array `lum` in the local memory of the board and generates a control unit to schedule the memory accesses. Table IV shows the results obtained after the P&R step of the HW graph generated by NENYA. The maximum clock frequency of the example is limited by the impossibility to do a timing-driven P&R. Moreover, the centralized control unit furnishes control signals that drive extremities of the FPGA with long wires, which degrades their delays.

Table IV. HW results for the BPIC example.

Area (cells)	Cycles	Delay (ns) @16 MHz	Address-data pairs	REC (us)
905	7	437.5	2,692	81.6

```

... // lum is an array of ints
int bitstring = 0;
for(byte xp=0; xp<4; xp++){
    MAXTIME c1 = new MAXTIME();
    c1.start();
    bitstring <<= 1;
    if(lum[xp] > mean_lum) bitstring |= 1;
    bitstring <<= 1;
    if(lum[xp+4] > mean_lum) bitstring |= 1;
    bitstring <<= 1;
    if(lum[xp+8] > mean_lum) bitstring |= 1;
    bitstring <<= 1;
    if(lum[xp+12] > mean_lum) bitstring |= 1;
    c1.end();
} ...

```

Figure 13. Part of the BPIC algorithm.

The third example is the HAL with 16-bit operands. The loop body has been considered to be implemented in HW. The segment only fits in the XC6264 and when we consider the use of an XC6216 the DFG is partitioned by NENYA in two time partitions to be executed with time-sharing of the FPGA (see the results in Table V). The time-sharing implementation uses SW to schedule the reconfigurations and to load/store the data. We specify the maximum area of the FPGA to be 60% (2,458 cells) of the overall FPGA area (to make sure that the routing is feasible). The results were obtained with the algorithm of Figure 11 oriented by ASAP levels. Better performance can be achieved with the use of the local memory of the board to store the data. The reconfiguration times do not consider the partial-reconfiguration capability of the FPGA.

Table V. Results for the HAL example.

FPGA	Area (cells)	Delay (ns) ♣	Address-data pairs	X × Y cells	REC (us)	Stores/loads #
XC6264	4,420	564	7,263	108×72	220.1	4/3
XC6216 #1	2,328	255	3,873	64×64	117.4	4/4
#2	2,156	309	3,346	52×64	101.4	4/1

6. Conclusions and Future Work

This paper presents a new approach to hardware compilation from a software programming language, more suitable to fully exploit the special characteristics of reconfigurable computing architectures. Intermediate representations best-tailored to performance-driven hardware synthesis of large examples have been explained. These representations are more adequate than those traditionally used by high-level synthesis algorithms. They permit to exploit the ordering of basic blocks in mutually exclusive execution paths by storing the control and merge dependencies in two distinct graphs. The paper also describes how the intermediate representation graphs from the Java *bytecodes* are extracted.

The approach includes the temporal partitioning technique in order to exploit the “virtual hardware” concept. The paper presents extended heuristics and the integration of temporal partitioning in hardware compilation at the behavior

level. However, the selection of the node to be scheduled in the current partition can be based on more complex cost functions. Further efforts must integrate the communication cost of each node to produce best results, and must consider at least the schemes of communication between temporal partitions identified in this paper.

Further work will focus on the support of methods' invocation (it can be used by the programmer to guide temporal partitioning) and on the development of scheduling and temporal partitioning schemes able to support cyclic regions. The possibility to share large hardware resources will also be considered (it can alleviate the execution overheads by merging more operations in each temporal partition).

Acknowledgments

The authors would like to acknowledge the support of the PRAXIS XXI Program under the scope of Project PRAXIS /2/2.1/TIT/1643/95, and of the Portuguese PhD program of the PRODEP 5.2 action.

References

- [1] W. H. Mangione-Smith, et al., "Seeking Solutions in Configurable Computing," IEEE Computer, December 1997, pp. 38-43.
- [2] J. Becker, R. Hartenstein, M. Herz, U. Nageldinger, "Parallelization in Co-Compilation for Configurable Accelerators," In *Proc. of the Asia South Pacific Design Automation Conference*, Yokohama, Japan, February 10-13, 1998.
- [3] G. De Micheli, R. Gupta, "Hardware/Software Co-Design," *Proc. of the IEEE*, vol. 85, no.3, March 1997, pp.349-365.
- [4] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, Inc., San Francisco, California, USA, 1997.
- [5] G. De Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw Hill, 1994.
- [6] P. Athanas, H. Silverman, "Processor Reconfiguration through Instruction-Set Metamorphosis: Architecture and Compiler," IEEE Computer, vol. 26, n. 3, March 1993, pp. 11-18.
- [7] J. Peterson, R. O'Connor, P. Athanas, "Scheduling and Partitioning ANSI-C Programs onto Multi-FPGA CCM Architectures," In *Proc. of the 4th IEEE Symposium on Field Programmable Custom Computing Machines*, Napa Valley, California, USA, April, 1996, pp. 178-179.
- [8] M. Vasilko, D. Ait-Boudaoud, "Architectural Synthesis Techniques for Dynamically Reconfigurable Logic," In *Proc. of the 6th Int. Workshop on Field-Programmable Logic and Applications*, Darmstadt, Germany, Sept. 23-25, 1996, LNCS, vol. 1142, Springer-Verlag, 1996, pp. 290-296.
- [9] I. Ouais, et al., "An Integrated Partitioning and Synthesis System for Dynamically Reconfigurable Multi-FPGA Architectures," In *Proc. of the Reconfigurable Architectures Workshop*, Orlando, Florida, USA, March 30, 1998.
- [10] B. Radunovic, V. Milutinovic, "A Survey of Reconfigurable Computing Architectures," Tutorial in the *8th Int. Workshop on Field Programmable Logic and Applications*, Talin, Estonia, 30 August - 2 September, 1998.
- [11] X.-P. Long, H. Amano, "WASMII: a Data Driven Computer on a Virtual Hardware," In *Proc. of the 1st IEEE Workshop on Field Programmable Custom Computing Machines*, Napa Valley, California, USA, April 5-7, 1993, pp. 33-42.
- [12] D. Soderman, Y. Panchul, "Implementing C Designs in Hardware: ANSI C to RTL Verilog Design & Test-Bench Compiler," Compilogic Corporation, March 1998. See <<http://www.compilogic.com/pubs.htm>>.
- [13] L. Semeria, G. De Micheli, "Synthesis of Pointers in C: Application of Pointer Analysis to the Behavioral Synthesis from C," In *Proc. of ICCAD 1998*, November 98.
- [14] J. M. P. Cardoso, H. C. Neto, "Towards an Automatic Path from Java™ Bytecodes to Hardware Through High-Level Synthesis," In *Proc. of the 5th IEEE International Conference on Electronics, Circuits and Systems*, Lisbon, Portugal, September 7-10, 1998, pp. 85-88.
- [15] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, Massachusetts, 1996.
- [16] ____, *IEEE Standard VHDL Language Reference Manual*, IEEE Std 1076-1993.
- [17] J. Ferrante, K. J. Ottenstein, J. D. Warren, "The Program Dependency Graph and its uses in optimization", *ACM Transactions on Programming Languages and Systems*, 9(3): 319-349, June 1987.
- [18] A. V. Chichkov, "HW/SW Co-design Methodology for Custom Computing Machines using Partitioning Based on the Implicit Parallelism," Ph.D. Thesis (in Portuguese), IST, Lisbon, January 1998.
- [19] Xilinx Inc., *XC6000 Field Programmable Gate Arrays*, version 1.10, April 24, 1997. See <<http://www.xilinx.com>>.
- [20] A. V. Aho, R. Sethi, J. D. Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley, Reading, Massachusetts, 1986.
- [21] B. Steensgaard, "CDGs and their mutations," *Microsoft Research*, October 12, 1995.
- [22] D. D. Gajski, N. D. Dutt, A. C.-H. Wu, S. Y.-L. Lin, *High-Level Synthesis, Introduction to Chip and System Design*, Kluwer Academic Publishers, 1992.
- [23] R. A. Bergamaschi, S. Raje, L. Trevillyan, "Control-Flow versus Data-Flow-Based Scheduling: Combining Both Approaches in an Adaptive Scheduling System," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 5, no. 1, March 1997, pp. 82-100.
- [24] J. M. P. Cardoso, H. C. Neto, "The Library of macro-cells used by NENYA: Circuit generators for the Xilinx XC6000 FPGA series," INESC Technical Report, December 1998.
- [25] M. Kaul, R. Vemuri, "Optimal Temporal Partitioning and Synthesis for Reconfigurable Architectures," In *Proc. of the Design, Automation & Test in Europe*, Paris, France, Feb. 23-26, 1998.
- [26] K. GajjalaPurna, D. Bhatia, "Temporal Partitioning and Scheduling for reconfigurable Computing," In *Proc. of the 6th IEEE Symposium on Field Programmable Custom Computing Machines*, Napa Valley, California, USA, April 15-17, 1998.
- [27] AT&T Inc., *doty: Graphviz*, version 1.3, 1998. See <<http://www.research.att.com/sw/tools/graphviz/>>.
- [28] Synopsys Inc., *Design Compiler v3.3a*, April 8, 1995.
- [29] Xilinx Inc., *XACTstep Series 6000 User Guide*, 1997.