IFPEC: Integrated Framework for Processor Extension Generation and Application Compilation

Kevin Martin, Christophe Wolinski
University of Rennes I, Irisa, Inria
France
{martin,wolinski}@irisa.fr

Krzesztof Kuchcinski
Dept. of Computer Science
Lund University, Sweden
krzysztof.kuchcinski@cs.lth.se

Antoine Floch, François Charot
Inria, Centre Rennes - Bretagne Atlantique
France
{anfloc,charto}@irisa.fr

Abstract—This paper presents our integrated design framework, IFPEC, used for automatic selection and synthesis of processor’s extensions as well as efficient application compilation for these newly generated extensions. Our approach is based on constraint programming (CP) and is able to solve hard design problems. Processor extensions, in our approach, are defined as computational patterns and represented as graphs. This, together with graph representation of applications, provides a way to use subgraph isomorphism and connected component constraints for identification of processor extensions as well as their selection, application scheduling, binding and routing. This is done for architectures composed of a processor and tightly connected run-time reconfigurable processing units, implementing the selected extensions.

I. INTRODUCTION

The IFPEC framework, which is an extended version of the DURASE [3] and the UPAK [6] systems, enables synthesis of application specific processor extensions to speed up applications’ execution. In general, the synthesized extensions are implemented as run-time, functionally reconfigurable processing units, tightly connected to a target processor. The newly created processor instructions are used to control this extension. Our framework is totally based on constraint programming (CP) and is implemented using the JaCoP [2] solver. The solver has been specially extended by graph constraints for subgraph isomorphisms, connected components, and cliques.

Currently, our system supports different models of architectures, as depicted in Figure 1. For instance, Figure 1A depicts an architecture composed of one run-time, functionally reconfigurable processing unit tightly connected to a processor. The newly created processor instructions are used to control this extension. Figure 1B presents another architecture example that is parallel and has a crossbar communication network between processing units. This architecture can also contain local memories. Our IFPEC framework supports also application compiling for the ROMA architecture [5], as shown in Figure 1C. This architecture contains a crossbar communication network enabling data transfers between processing units and distributed memories. These examples illustrate the fact that the IFPEC is an open framework, which can be easily adapted to support different complex architectures.

The design flow of our framework is presented in Figure 1. The input is an application code written in C, a target processor instruction set and an architecture model (supporting sequential or parallel execution modes). The output are processor extensions, application specific instructions for accessing these extensions and a reconfiguration file in the case of the architecture from Figure 1C. The extension’s synthesis can be highly optimized when a FPGA implementation is needed. In this case, the number of logic elements used for the implementation is minimized.

In the IFPEC framework, the C program is compiled into a Hierarchical Conditional Dependency Graph (HCDG) internal representation [1]. This graph is later used for identification of computational patterns, called “pattern generation” in Figure 1. The generated definitively selected pattern set (DIPS) is then used for selection of computational patterns in step “Graph covering, Scheduling”. If an extension can execute many computational patterns, the corresponding run-time, functionally reconfigurable processing unit can be synthesized using a merging tool [4]. This tool merges together all selected computational patterns (“Pattern merging” step in Figure 1). This step uses a specially developed clique constraint. Our system also generates an interface to the processor and the transformed application source code, including application specific instructions.

The abstract generic architecture model of a processor extension from Figures 1A and 1B is depicted in Figure 2. It is composed of a processor interface, a set of processing units U, a set of registers R and two sets of multiplexers MAS and MBS. Figure 2 shows, as an example, the NIOS processor interface. The processing units can be heterogeneous, i.e., each unit can execute a specific set of complex operations. Generally, a particular unit can contain a run-time, functionally reconfigurable data-path. Data transferred from the processor to the extension and data passed directly, without processor intervention, between processing units can be stored for further processing in an extension’s internal register file.

II. EXPERIMENTAL RESULTS

Table I presents results obtained for selected applications from MediaBench, MiBench and Cryptographic Library benchmark sets. These results have been generated for the architecture from Figure 1A. These applications are written in C and compiled using our design flow for the ALTERA Nios2Fast processor running at 150MHz on a Stratix2 Altera...
The patterns have been generated to not exceed four inputs, two outputs and 10 graph nodes. $|V|$, $|P|$ and $|Psel|$ stand for the number of nodes in the application graph, the number of identified and the number of selected patterns, respectively. The application graphs are composed of independent subgraphs and column Optimality specifies for how many of these subgraphs optimality was proven. For instance, JPEG IDCT application graph is composed of six subgraphs and for four of them the optimality was proven.

### REFERENCES