Course Grained Reconfigurable Arrays (CGRAs)

Today’s embedded applications demand high compute performance at a tight energy budget, which requires a high compute efficiency. Compute efficiency is upper-bound by the technology node, however in practice programmable devices are orders of magnitude away from achieving this intrinsic compute efficiency. Coarse Grained Reconfigurable Arrays (as shown in Figure 1) are one of the proposed solutions to this problem [1] that is capable of providing the required performance. However, adapting such highly specialized architectures to your application and translating the application into a form that can run on it is an intrinsically complex endeavour. Designing the compiler is particularly challenging because of the explicit data-path of the proposed architecture, and code has to be generated for all possible combinations of IDs and FUs. In addition to the tasks of a regular compiler, the compiler needs to route data between FUs. This is similar to compilers for transport triggered architectures [2].

Processor architecture instantiation

Using the CGRA template it is possible to instantiate customized energy efficient processor architectures. Figure 2 illustrates this with an example VLIW processor architecture instantiation. A key feature here is that the output of the computation unit (ALU) can be provided directly as input to the next operation, thus bypassing the register file completely. This allows for an extremely energy efficient execution of the program but results in difficulties when designing a compiler for the platform.

Code generation for explicit datapath architectures

One approach for scheduling for an explicit data-path is list scheduling using a resource graph (RG). The RG has a node for every FU at every clock cycle in the schedule. Scheduling is done by mapping nodes from the data dependence graph onto the nodes in the RG. However, an infeasible partial schedule may be found when the result of a scheduled operation can not be routed to its destination, because the required pass-through resource became occupied during scheduling [1]. In such cases, either the ordering in which instructions are scheduled should be changed, or a more complex scheduling algorithm is required. Such scheduling algorithms can include backtracking schedulers, capable of revisiting previous scheduling decisions, or will need to use more generic approaches such as constraint programming.

Scheduling operations

The list scheduling algorithm is shown in Figure 4. The operand paths are found by Dijkstra’s shortest path algorithm, similar as done in [3]. Unlike in the previous work, functional unit selection is also performed by the shortest path algorithm. For each operand the shortest path from the producer of the operand to any other functional unit is calculated. The functional unit with the least total cost is selected to execute the operation. The cost of a path is calculated by using the weights of the edges in the resource graph.

Conclusions and future work

A well working compiler is critical for the adaption of your new accelerator architecture while energy efficient architectures can be very problematic for existing compiler frameworks. Using the methods presented here we are now able to map and schedule basic C code onto the platform. However, this is just the first step towards solving the compilation problems for our CGRA architecture.

References


Illustrations

- Figure 1: CGRA template.
- Figure 2: Example VLIW processor instantiation.
- Figure 3: Example scheduling of a simple operation.
- Figure 4: Scheduling algorithm pseudocode.