In this chapter, we present the Spark software tool in which we have implemented our parallelizing high-level synthesis methodology. We first discuss implementation details and briefly describe how the tool can be used. We then describe how synthesis scripts can be used to guide the transformations and heuristics applied by Spark during synthesis. In particular, we use the synthesis scripts to study the effects on synthesis results of the various speculative code motions, the branch balancing techniques, and different ways of calculating operation priorities.

### 9.1 Implementation of the SPARK PHLS Framework

We implemented the parallelizing high-level synthesis methodology presented in this book consisting of the scheduling heuristics, the pre-synthesis transformations and the synthesis and parallelizing compiler transformations in a prototype framework called Spark. The Spark software tool is over 100,000 lines of C++ code and uses the EDG (Edison Design Group) C/C++ front-end parser [EDG].

Spark takes a behavioral description in ANSI-C as input – currently with the restrictions of no pointers, no function recursion, and no irregular control-flow jumps. In addition to this, the designer has to specify the list of resources allocated to synthesize the design in a hardware resource library along with their timing information and the bit-widths of the data types used in the C input. Thus, resource allocation and module selection are done by the designer and are given as input to the synthesis tool through the hardware resource library. Spark provides the ability to control and thus, experiment with the various code transformations and heuristics employed during synthesis using synthesis scripts and command-line options.

Figure 9.1 presents an overview of the Spark framework. After parsing the design description, we capture it using the 3-layered intermediate representation (IR) described in Chapter 3. Recall that the three layers in this representation are hierarchical task graphs (HTGs), control flow graphs (CFGs), and data flow graphs (DFGs).
Since the HTGs store structural information about the control structures (if-then-else, loops, et cetera) in the code, they are useful for efficiently applying coarse-grain and global code transformations such as loop unrolling, loop pipelining, speculative code motions, *Trailblazing*, chaining across conditional boundaries, et cetera. CFGs are useful for design traversal during scheduling and DFGs capture data dependencies among operations that is useful during scheduling, code motions, and chaining.

The *Spark* tool first applies a range of pre-synthesis transformations as shown in Figure 9.1 and then proceeds to the scheduling phase. We built this phase in a modular style using a transformations toolbox that contains a range of transformations and techniques such as speculative code motions and *Trailblazing* that were discussed earlier in this book. We implemented the scheduling heuristics presented in Chapter 7 in this scheduler framework. These heuristics can be guided using the synthesis scripts as explained in the next section.

After the scheduling phase, *Spark* executes the interconnect minimizing resource binding and control synthesis passes discussed in Chapter 8. This is followed by a