Chapter 7
Perspectives on Programming Reconfigurable Computing Platforms

Despite tremendous progress in the development and integration of compilation and synthesis techniques, the challenging nature of the compilation and synthesis for reconfigurable architectures has defied the establishment of a de facto standard methodology. In this chapter, we begin by providing an overall perspective of what we believe is missing to make reconfigurable computing an ever increasing reality. We then outline several outstanding issues suggesting a set of research directions in compilation techniques for these architectures. In this context, we have focused on compilation techniques and have deliberately omitted system-level aspects of reconfigurable architectures such as dynamic reconfiguration and operating system-level services. We then describe a vision, albeit speculative, of a compilation flow augmented by the synergetic integration of language description and transformation specification techniques as well as the notion of resource virtualization. Finally, we discuss how reconfigurable technologies can play an important role in future VLSI devices where unreliability is an important issue [57, 130]. In this context, we highlight how compilation techniques for reconfigurable architectures can also play a role in emerging nanotechnology systems.

7.1 How to Make Reconfigurable Computing a Reality?

The last 15 years have witnessed a great enthusiasm for reconfigurable computing as a new and broad computing paradigm with great computational flexibility and performance potential [83, 90, 122]. Despite many research efforts, first in academia and more recently in industry, reconfigurable computing has not been widely adopted as the dominant computing paradigm. We believe there are several key factors that contribute to this, as addressed in the next sections.
7.1.1 Easy of Programming

To make reconfigurable computing approachable to the average programmer, programming tools must offer a set of high-level programming abstractions programmers can easily grasp, and execution models they can easily reason about. As with early compilers for traditional architectures, compilers for reconfigurable architectures must hide the complexity of the low-level programming details while exploiting a wide range of mapping choices in the pursuit of effective computing solutions.

In this context, researchers have proposed various high-level programming language approaches. While imperative languages such as MATLAB offer the clear benefits of sequential semantics, uncovering the underlying concurrency is an extremely complex problem. Some approaches relaxed the sequential semantics of some of its constructs to facilitate the extraction of concurrency from a sequential program (e.g., SA-C [44]), while other efforts have focused on explicitly concurrent languages inspired by the CSP programming model that better matches the spatial nature of reconfigurable architectures [127].

While there is no obvious path that offers the best of both worlds, we envision three main, and possibly complementary, approaches that can ameliorate the programming challenges for reconfigurable architectures, namely:

- **Augmenting Imperative Languages:** In this approach, programmers would use concepts based on aspect-oriented programming [109, 174] to augment an application where information compilers are (currently) unable to derive. For example, aspects can allow the programmer to specify execution modes for specific regions of the code, to indicate data properties such as streaming data rates from input devices, or to add complementary information about an algorithm, not present in the programming model used. The programmer would retain the benefits of an imperative programming language while relying on the richness of aspects to aid the compiler in the mapping of the application to the underlying architecture.

- **Transactional-Based Languages:** In this approach the language allows the programmer to explicitly define regions of the code that execute sequentially, transactions [198], but whose execution order can be arbitrary, provided they execute atomically. Transactions offer a programming model that is concurrent in nature, but isolate the programmer from having to explicitly manage concurrency and data orchestration throughout the execution [247, 336]. In combination with emerging productivity-oriented parallel programming languages (e.g., X10 [77]) that provide mechanisms for high-level synchronization and assignment of data to locus of computations, these approaches might represent promising avenues for programming reconfigurable architectures.

- **Domain-Specific Languages:** In this, nongeneric programming approach, the language would include domain-specific constructs and/or knowledge about the specific target architecture (such as with the DIL [55] and the RaPiD-C [84] languages). Features such as the partitioning among hardware resources with