Chapter 13. Conclusions and Future Work

Multi-FPGA systems is a growing area of research. They offer the potential to deliver high performance solutions to general computing tasks, especially for the emulation of digital logic. However, to realize this potential requires a flexible, powerful hardware substrate and a complete, high quality and high performance automatic mapping system.

In the last five to ten years there have been a large number of multi-FPGA systems built and proposed. Work has also been done on creating software algorithms to support these machines. However, in general these hardware and software systems have been created in an ad hoc fashion. The hardware systems are grab-bags of appealing features with little investigation into what is truly best for the applications they will support. For software, individual algorithms have been proposed, but with little insight into which approaches are the most important, and how these pieces can be combined into a complete solution.

The primary goal of this thesis has been to offer a disciplined look at the issues and requirements of multi-FPGA systems. This includes an in-depth study of some of the hardware and software issues of multi-FPGA systems, especially logic partitioning and mesh routing topologies, as well as investigations into problems that have largely been ignored previously, including pin assignment and architectural support for logic emulator interfaces. This work points out the challenges of multi-FPGA systems, and presents solutions to many of them.

In Chapter 6 we presented Springbok, a novel rapid-prototyping system for board-level designs that offers many advantages over current systems. Its flexible architecture accommodates a great range of system sizes and topologies. With the ability to solve problems as they occur, Springbok uses its resources more efficiently than do fixed FPGA-based systems, which require a very conservative style. Including arbitrary devices and subsystems into the Springbok structure allows even greater efficiency and accuracy. Finally, the use of FPGAs instead of FPICs for the routing structure reduces overall costs, adds flexibility, and more easily handles the functionality necessary to interface to timing-inflexible components.

To design Springbok, as well as many other multi-FPGA systems, requires a hard look at the hardware structures used to build them. Chapter 7 presented several techniques for decreasing routing costs in mesh interconnection schemes: 1-hop interconnections, Superpins, and Permutations. These topologies reduce I/O and internal routing resource usage, increase bandwidth, and reduce delays in the system.

Including an emulation into its target environment is one of the most important benefits of emulation over simulation. However, with current systems the user is required to develop an application-specific protocol transducer to allow the emulation to communicate with its environment. In Chapter 8 we demonstrated that many logic emulator interfaces can be handled by a simple, generic interface transducer board. While the protocol transducer mappings can be somewhat involved, we have shown that they can be quickly developed in a high-level language, and automatically translated into FPGA logic. Thus, with a small amount of design effort, an emulation can be run in its target environment, greatly increasing its utility.

To be successful and widely used, a multi-FPGA system needs more than just a flexible hardware system. Also necessary is automatic mapping software that maps circuits onto this structure. This software needs to be fast, fully automatic, and capable of producing high-quality results. Chapter 10 presented a survey of bipartitioning techniques, mainly those based on the Kernighan-Lin, Fiduccia-Mattheyses algorithm. We have discussed many optimizations to the algorithm, including several new approaches, which have yielded a fast and efficient bipartitioning algorithm that is significantly better than the current state-of-the-art.

Chapter 11 presented an algorithm for automatically applying bipartitioning to an arbitrary multi-FPGA system. It analyzes the multi-FPGA system topology, and recursively applies bipartitioning, as well as multi-way partitioning and multi-sectioning where appropriate, as part of a fast and automatic method for mapping to a multi-FPGA system.

Pin assignment was covered in Chapter 12. Our algorithm adapts to an arbitrary multi-FPGA system topology. It creates better routing than current systems, which results in faster mapping runtimes, and higher quality mappings.

By combining the algorithms discussed in this thesis with those commercially available, a complete automatic-mapping system can be developed. It offers fast runtimes, with good quality results, and adapts to an arbitrary topology. In this way, a single system can be applied to the numerous different topologies proposed, including topologies with flexible interconnection patterns.

This thesis has provided an in-depth investigation into most of the issues in multi-FPGA hardware and software. As a result, the hardware and software structures of multi-FPGA systems have been optimized, yielding faster and higher-quality systems. Bringing these advantages to bear on board-level prototyping, such as in the Springbok system, gives logic designers a powerful new tool for system-level debugging.

One of the most important aspects of the work contained in this thesis is that the approaches given here are applicable not just to Springbok, but also to multi-FPGA systems in general. The routing topologies can improve many multi-FPGA systems, and the generic interface support is useful for existing logic emulators. The software algorithms provide architecture-adaptive tools to optimize to arbitrary topologies, delivering improved quality, in much less time, than current mapping software.

While the techniques discussed can help many different systems, there are other opportunities to improve multi-FPGA systems, and to achieve the full promise of reconfigurable logic. There are many outstanding issues in both the hardware and software constructs of FPGA-based system.

While this and other works have focused on some portions of the mapping software flow, there is still much yet to be done. One example is the partitioning of digital logic onto multi-FPGA systems. This thesis has included a survey of many bipartitioning techniques, resulting in significant quality and mapping speed improvements over current techniques. However, in many cases multi-way partitioning algorithms make more sense than bipartitioning techniques. Thus, a similar study investigating the many multi-way partitioning algorithms, which integrates the techniques discussed in this thesis, has the potential to provide even better results. Also, while iterative bipartitioning can map to an arbitrary topology, its greedy nature may generate substandard results. Alternatives are possible, including extensions of quadrisection and other techniques that simultaneously perform multiple cuts. However, these techniques must be modified to handle the different possible topologies found in multi-FPGA systems.

Another important issue in the mapping tools for multi-FPGA systems is the time the software tools take to create a mapping. With current software taking up to a full day to complete a mapping, there are many applications that do not take advantage of multi-FPGA systems because of the mapping delays. There are several ways to combat this issue. The fast partitioning and pin assignment tools contained in this thesis are one such step. Another possibility is to examine the other parts of the mapping software and develop new algorithms optimized for this domain. Specifically, current systems use technology-mapping, placement, and routing software originally created for single-chip systems. However, single-chip systems have much different constraints than multi-FPGA systems. The FPGAs in a current multi-FPGA system can expect to have only a small fraction of their logic capacity used, but all of their I/O pins will be consumed. Tools that understand this domain may be able to perform their optimizations much faster than tools optimized for single-chip mapping situations.

Another possibility for improving the performance of multi-FPGA system mapping tools is to adopt and extend the two-phase methodology proposed in this thesis's partitioning chapter. That is, it is not always necessary to achieve the best possible mapping, and a lower quality, but faster, mapping algorithm may be a better choice. Cases where more than ample resources are available, and optimal performance is not critical, are obvious situations for applying poor but fast software. However, even when the highest performance is required, only using high-quality but slow software may not be the best answer. Specifically, when using the best possible mapping software the initial mapping steps (such as partitioning) will usually need to be run on a single processor, and will take a long time. Once these steps are done, multiple independent place and route processes can be run on different machines, speeding up the mapping process. However, there is no reason that these processors must remain idle in the early mapping phases. One possibility is to run the fast, but lower quality, mapping tools concurrently with the high quality, but slow, algorithms. These faster algorithms could quickly generate a mapping, and have the emulation system running the mapping much sooner than the other approach. While the emulation will run more slowly than the higher quality mapping produced by the slower software, the emulator will be making headway as the slower software completes. Once the improved mapping is completed, it is relatively simple to pause the emulator, save the current state of the emulation, and load the improved mapping. By properly setting the stateholding elements in the new mapping, the high-quality mapping can resume where the slower mapping ended. In this way, the emulator can be used more efficiently, with shorter down times, and the user gets answers much more quickly than current systems can achieve. However, there is much research to be done on determining how to quickly create these lower-quality mappings.

One can also speed up the software by minimizing the amount of remapping necessary. Specifically, if a minor upgrade or bug fix is made to an input circuit, it should be possible to reuse much of the previous mapping. Since much of the circuit remains the same, a fast way to create a new mapping is to just remap that portion of the circuit that has changed. While our architecture-adaptive partitioning and routing tools are capable of working on arbitrary subsets of a multi-FPGA system, there are several other pieces necessary to achieve a true incremental update facility. Specifically, the synthesis tools that generate the mapping need to modify only those portions that have changed, or else the modification will alter the entire mapping, and the incremental facilities of subsequent tools will be useless. Also, technology-mapping, placement, and routing tools can also be given an incremental-update capability, yielding even greater performance increases.

While the ability of our algorithms to adapt to an arbitrary topology can be important as part of an incremental update mechanism, this adaptability also opens up another opportunity. Since the tools can optimize for any architecture, the tool suite becomes an ideal system for performing architecture exploration. In this thesis, we examined mesh routing topologies based on totally random routing requirements. While this made it possible to get some idea of the tradeoffs in mesh architectures, it also limited the study's scope. Most obviously, real circuits are not completely random, and although our study took into account some of the non-randomness of circuits (such as buses), better results could be achieved by using actual circuits in the study. Thus, a toolset that can automatically adapt to an arbitrary topology is the key to better understanding how multi-FPGA systems should be constructed. This would not only yield comparisons of different optimizations to a given type of topology (i.e., comparing meshes to meshes, and crossbars to crossbars), but also allow comparisons between different types of topologies (i.e., meshes to crossbars). In this way, we can continue the search for better architectures, resulting in higher quality systems.

There are many other opportunities for work in multi-FPGA systems and reconfigurable hardware. One possibility is to push forward with the deployment of systems such as Springbok. Although Springbok holds a lot of promise for the prototyping of board-level designs, there is still much work necessary to bring it to fruition. Detailed design of the hardware, especially in the connector technology to enable Springbok's flexibility, as well as communication and debugging support to help the user understand the operation of their prototype, is key to a complete implementation. The software tools will also need to be extended. Most obviously, there is currently no algorithm for assigning daughter cards to specific locations to achieve the highest-quality mapping. That is, while the input specification for a circuit may require a specific mix of daughter cards, there still needs to be an algorithm that decides where on the baseplate these daughter cards should be placed to minimize communication and improve performance. While simulated annealing should be able to handle this task, this would be a very slow answer to the problem. It is likely that alternative placement algorithms, such as force-directed placement, or possibly combining daughter card placement with the partitioning algorithm (assigning daughter cards to different partitions as well as logic nodes), could produce nearly equal quality results in much less time. There is also the issue of how and when to insert extender cards into the array. As each of the software mapping algorithms run, they may encounter resource limitations that cause significant decreases in mapping quality, or even cause the mapping to fail. A way to solve these problems in the Springbok system is to add extender cards with extra resources to ease these bottlenecks. However, how best to integrate this feature into current algorithms is an open question.

Looking beyond Springbok, there are many other opportunities in reconfigurable hardware. One of the most promising is combining together reconfigurable logic and standard processors. While we have discussed many applications where FPGAs can provide the highest performance implementation, most situations fit better onto standard processors. In current systems, there is usually a strict separation of the two resources. If a problem fits well onto FPGAs, it is mapped onto an array of FPGAs. If not, it must be run on a separate, standard workstation. This separation ignores the numerous applications where a mixed approach might be better. Normally, an algorithm goes through a series of phases. Often this might involve some complex initialization protocol, a regular inner loop that consumes most of the computation time, and another complex final cleanup step. The complexity of the starting and ending phases require the flexibility of standard processors, while the inner loop could be best implemented in FPGA logic

The answer to this problem is to integrate together the two types of resources. One way is to construct a mixed FPGA-microprocessor system. By combining several off-the-shelf FPGAs and processors, a single system can provide efficient mappings for many different types of algorithms. The processors in the system provide complex control flow for external interaction, complex mathematical computations, and irregular initialization and cleanup. The FPGAs provide huge amounts of fine-grain parallelism for fast bit-manipulations, large data-parallel operations, and custom datapaths. Thus, merging the two resources together could achieve much greater performance than either type of resource separately can provide.

A more exciting possibility is to add reconfigurable logic into the processors themselves. Specifically, for many applications there is a small set of operations that would greatly improve their performance, but which are not useful widely enough to be included in a general-purpose processor. By including reconfigurable logic into the processors, these instructions can be built on a per-application basis, with different mappings to the reconfigurable logic for different programs (or even different sections of a single program). Thus, we would have a general-purpose method of providing special-purpose processors. This logic may not look anything like current FPGA logic. The reconfigurable logic will need to be optimized to this application domain, opening up a very interesting area for future architectural investigations. Also, the processors themselves may need to be modified to accept this new type of resource. However, the most critical area of research for these types of systems is in determining how compilers can be made to recognize and optimize for the opportunities provided by this added logic. Good compiler support for this application is critical to achieving widespread benefit from the reconfigurable logic.

This logic need not be solely a method of adding extra instructions to the processor's instruction-set. A different model for using these resources is FPGAs used as a generic coprocessor system, where the reconfigurable logic is viewed more as a separate processing element, perhaps with a different control stream, than as just another unit in the processor. In current machines, coprocessors are typically provided for floating-point arithmetic, graphics acceleration, and possibly data encryption. These coprocessors are used not only because they can speed up these applications, but also because these applications are perceived to be important enough to merit special-purpose hardware. Many other algorithms could benefit from special-purpose coprocessors, but are used so infrequently that providing this custom hardware is impractical. A solution to this problem is to add a generic, reconfigurable coprocessor to the system. While it may not be able to achieve the same performance as a custom chip, a reprogrammable coprocessor can economically provide improved performance to many different algorithms. The coprocessor will be shared across many different algorithms, since each algorithm can have its own configuration. Thus, although each algorithm individually is not of wide enough utility to merit a special-purpose coprocessor, by sharing the same generic coprocessor with many different algorithms the hardware becomes practical.

Such processor-FPGA hybrids, even if they prove practical, are still many years away. In that time there are several changes in the reconfigurable logic domain that are important to anticipate. First, the primary bottleneck of current systems may (at least partially) be alleviated. While today's systems suffer from a significant I/O bottleneck, this need not always be so extreme. There currently are much higher-I/O packages available, such as those used by FPICs to achieve 1,000 pin chips. Although these technologies are quite expensive now, with relatively few applications taking advantage of the extra connectivity, as more applications migrate to these technologies the cost should greatly decrease. Thus, it is quite likely in the near future that FPGAs may see a nearly fourfold increase in the available I/O pins. Also, there are techniques such as Virtual Wires that can also help alleviate the I/O bottleneck. Time-division or voltage-division multiplexing can greatly increase the available bandwidth even in today's I/O-bound chips. Thus, while the I/O bottleneck may never completely disappear, there is great potential to ease the current I/O crunch.

If the potential for increasing the external bandwidth of current FPGAs actually is realized, it may require a complete re-evaluation of our current systems. While today's software and hardware constructs are primarily built to minimize the required inter-chip routing, with more I/O bandwidth other issues become more critical. Primarily, the performance of the mapping becomes an even greater concern, and algorithms such as min-cut partitioning may give way to more critical path oriented approaches. Investigating these issues will become an even greater concern in the future.

Another change in the current multi-FPGA system equation is in the types of circuits handled, or at least in the perception of what these circuits look like. Most current systems either are hand-mapped, and thus highly aware of the peculiarities of a given mapping, or automatically mapped, where the input circuit is assumed to be purely random logic. However, even in our current mappings, the circuits are not truly random. Much of the current logic is regularly structured datapaths. By treating these datapaths as random logic, the automatic mapping software throws away many potential optimizations. While optimizing for random logic may have been the right answer for initial system construction, since if you can map random logic you can handle anything, future systems may need to be more circuit-aware. Specifically, the automatic-mapping software should be able to recognize highly regular datapaths - or require that the user highlight these components - and take advantage of the regularity to provide much better mapping quality. Techniques such as bit-slicing or pipelining at chip boundaries can both improve the performance while decreasing the inter-chip bandwidth. Memory elements in input circuits can also require special processing. While many multi-FPGA systems include separate memory chips, and thus memory structures in the circuit can be handled by these elements, the circuit may need to be restructured to take advantage of these facilities. Since the memories in the hardware may differ from those specified in the circuit, resynthesis and repacking into the available memories will be important to achieve the highest quality mapping. Such circuit-sensitive optimizations should become an ever more important part of a complete automatic mapping solution.

Although there is much work to be done on the hardware and software aspects of multi-FPGA systems, and reprogrammable logic in general, the potential payoffs are significant. With ever faster mapping tools, as well as hardware systems that simplify the job of the mapping algorithms, we may be able to create mappings in the same time it takes us now to compile programs. With such a system, it may be as easy to implement an algorithm in hardware as it is now to run them on a processor. Thus, software programmers will then have access to the power and performance to which the hardware designers are accustomed.

These advanced software processes, as well as improved hardware systems, will push logic emulation further and further into the currently exclusive domain of software simulation. Where today a designer will only use emulation when no other solution will suffice, with fast mapping software logic emulation becomes much more competitive. While very small, short tests will always be the domain of software simulation, testing of large subsystems can much more quickly be performed on a logic emulator if the mapping tools are fast enough. With board-level emulation systems similar benefits will also be achieved for multi-chip designs.

Higher quality software may also push FPGA-based custom-computing machines into the mainstream. Where large supercomputers currently hold dominance, multi-FPGA systems may become an equal partner. While there are some algorithms for which a supercomputer will always be the best answer, likewise there are algorithms for which only the fine-grain parallelism of a multi-FPGA system will suffice. However, the key to making FPGA-based supercomputers generally useful is automatic mapping software that can deliver high enough quality to rival software compilers for traditional programming languages.

Finally, we may see reconfigurable logic making inroads into the workstation and PC workplaces. While this logic may bear only passing similarity to current FPGAs, it will have many of the software and hardware issues of FPGAs. With the ability to add custom instructions to standard processors, as well as truly generic coprocessors for accelerating many types of algorithms, reconfigurable logic may provide a significant performance boost for many situations.