

The Tools Have Arrived: Two-Command Compilation and Execution of Scalable, Multi-FPGA Applications

Brian Holland
SRC Computers, LLC
4240 N. Nevada Ave
Colorado Springs, CO 80907
www.srccomputers.com
bholland@srccomputers.com

Abstract—Continual increases in the computational resources of Field-Programmable Gate Arrays (FPGAs) provide greater capabilities for algorithm migration to hardware to capitalize on higher performance at lower power. However, wider adoption of FPGA technologies is often assumed to be limited by development costs associated with lengthy toolflows dissimilar to traditional development of software applications. Despite this perception, an efficient toolflow for FPGA development is currently available and has demonstrated success in cost-effective migration of applications to hardware. The toolflow allows straightforward compilation and execution of applications through GNU make and a standard UNIX runtime environment. This paper describes general requirements and challenges of productive toolflows for FPGA systems. It also presents the SRC¹ MAP processor, Carte development environment, and associated runtime environment for efficient application execution. Lastly, the paper demonstrates the efficiency of the SRC toolflow with a string-matching application.

I. INTRODUCTION

FPGAs are continually evolving beyond the traditional role of replacing application-specific integrated circuits (ASICs). For example, systems from SRC Computers [6] have expanded the usage of FPGAs from glue-logic replacement to peer processors based on the customer's needs for faster and more customized data processing. FPGAs provide increasing capabilities for scientific computation including more fixed resources (e.g., multiply accumulators) and soft macros (e.g., IEEE floating-point operations). They also have grown both in I/O capacity and ability to rapidly integrate new interconnect standards.

FPGAs typically use more discrete tools than other parallel-computing technologies, which has been traditionally considered a key challenge for greater deployment of FPGA-based computing systems. In contrast, multi-core processing uses applications code based on threads (e.g., pthreads), shared memory models (e.g., SHMEM), or message passing libraries (e.g., MPI). Sequential codes are modified using first or third-party libraries plus an associated runtime environment

for relatively straightforward application migration. Similarly, graphics and other vector-processing technologies involve adaptation of sequential codes with technology-specific instructions (e.g., CUDA or OpenCL). However, FPGA systems traditionally require conversion of legacy microprocessor applications to parallelized (e.g., pipelined) circuits using a hardware description language (HDL), integration with a third-party system-level or board-level interfaces, synthesis, and finally place and route using FPGA vendor tools. This manual application development process implies three distinct steps and at least two toolsets from different sources plus further complexity depending on the runtime environment for the FPGA system.

Despite these widespread perceptions, efficient application development and migration for FPGA-based systems exists through effective integration of tools and encapsulation of tedious elements outside the scope of a typical application developer. High-level compilers can greatly assist in conversion of traditional software code into hardware-oriented circuits. These compilers and other third-party tools can also assist in the connection of the application kernel(s) to the requisite system interface. Assuming correct functionality of the kernels and system interfacing, the synthesis and place-and-route (PAR) toolflow can be scripted and therefore largely invisible to the user except for their nontrivial duration. SRC Computers provides the only standard language toolflow, the Carte development environment, that tightly integrates all three levels of tools and includes an efficient runtime management system.

The desire for efficient application development and system usage is a key research emphasis for FPGA computing. Some efforts have focused on identifying strategic challenges [5] and classifying traditional limitations [4] of these systems. Panels such as one at the international conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)[3] described productivity as a major requirement for application migration. However, researchers and panel participants often lack exposure to the capabilities of the available toolflows. Insufficient attention is given to summarizing application developer needs,

¹SRC, MAP, Carte, and CDA are trademarks or registered trademarks of SRC Computers, LLC. All rights reserved. All other trademarks are the property of their respective owners.

identifying toolflows such as the Carte toolflow that efficiently and effectively meet these needs, and quantitatively exploring their capabilities.

The rest of this paper is structured as follows. Section II further details key features and challenges for efficient application kernel migration, system integration, FPGA synthesis and PAR encapsulation, and runtime management. Section III describes the SRC integrated and unique approach to compilation, system integration, encapsulation of synthesis and PAR, and runtime management. Section IV presents a walkthrough of efficient application migration using SRC hardware and the Carte toolflow with a string-matching application. Conclusions are given in Section V.

II. APPLICATION TOOLFLOWS: FEATURES AND CHALLENGES

The complexity of migrating application kernels to hardware circuits, integrating these kernels with FPGA-based systems, encapsulating the low-level synthesis, place and route details, and using the associated runtime environment are key challenges for efficient usage of such systems. This section describes the requisite features of these four aspects of application development and how increasing their integration can overcome the perceived limitations of FPGA-based computing.

A. Application Kernel Migration

Developing or migrating applications to FPGAs is often related to the classical challenges of algorithm parallelization. Although applications may benefit exclusively from lower power consumption on FPGA systems as compared to microprocessor-based platforms, the performance of many applications increases from the spatial and temporal parallelism inherent in the FPGA's architecture. Approaches to FPGA parallelization can benefit from microprocessor strategies such as loop unrolling and pipelining, but can also suffer from similar challenges of lengthy iterations of implementation, evaluation, and revision. Particularly for FPGA systems, complex structures need to be implemented and subsequently modified simply and quickly. However, these parallel structures must be granular enough to allow for widespread reuse, capitalizing on a library-type approach.

Reusable structures maintain their simple and quick characteristics if they minimize deviation from the original application source code, description, or programming model. For example, textual application specification with C syntax provides a paradigm familiar to programmers. While non-trivial to construct, a specialized compiler can convert this textual code into efficient FPGA designs if the underlying specification includes intuitive representations of parallelism and timing. The compiler would construct full application designs from pre-optimized components. However, even with efficient application specification, revision, and compilation, FPGA designs cannot be developed in isolation of system-level details.

B. System Integration

Efficient application migration requires scoping the parallelization to the constraints of the system. The number and capacity of the FPGAs represents a key constraint, often requiring a different and potentially more complex parallelization strategy than a microprocessor-based system. Tradeoffs include application decomposition among multiple FPGAs and within the resources of a single FPGA (e.g., lower area, lower performance kernels or fewer higher area, higher performance kernels). A lack of integration between kernel-level and system-level tools can require manual selection of parallelization strategies, implementation, and ultimately costly reimplementations if initial choices prove infeasible.

Additionally, management of the system interconnect bandwidth can be a neglected aspect of application migration with disconnected toolsets. Beyond the classical problem of ensuring sufficient overall bandwidth for sustained computation, dividing the available bandwidth to maximize performance and prevent deadlock can become the burden of the application developer. The memory hierarchy further complicates the effective usage of the communication infrastructure. Without system infrastructure, applications must manually connect data transfers between possible combinations of microprocessor memory, FPGA on-chip memory, and the one or more levels of off-chip memory (e.g., SRAM or SDRAM) associated with an FPGA board or module.

C. Runtime Libraries

The runtime environment for an FPGA system also affects the efficiency of the aforementioned FPGA, interconnect, and memory resources. Ideally, an application should only occupy the FPGAs and memories necessary for the computation and this allocation should not prevent the usage of the remaining system resources by other applications. An application cannot determine a priori the resources currently assigned to other applications. Therefore, the application must either wait for its statically assumed resources to become available or be sufficiently adaptive to function with any available resources assigned to it by the runtime environment. The capabilities of the runtime system are related to the organization of the resources in the system architecture.

Systems defined by FPGA cards connected to the microprocessor peripheral bus have traditionally lacked a single unified manager of the global resources. Figure 1 illustrates a common architecture for an FPGA system that organizes a number of FPGAs and memories around a microprocessor and manages this resource node as an atomic unit. Node allocations are controlled by traditional cluster management software while low level drivers manage individual resources and interconnects. The microprocessor-oriented system is relatively easy to manage but wasteful due to coarse-grain resource allocation and has lower performance due to a communication bottleneck through the microprocessor. Additional bottlenecks are also present when coordinating multiple resources through a common peripheral interconnect such as PCI Express. An

Fig. 1. Cluster System

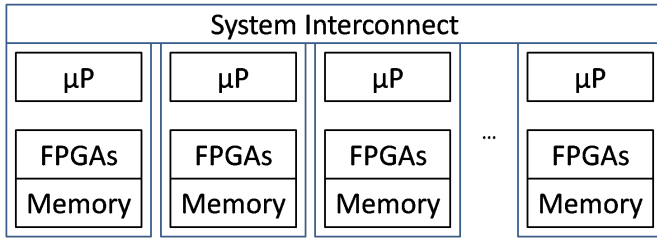
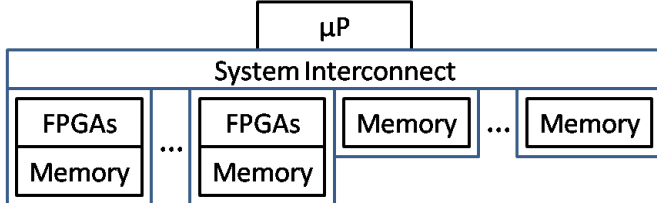


Fig. 2. Direct Connect



alternative organization, Figure 2, allows individually addressable FPGA and shared memory resources albeit with increased system and runtime complexity.

D. Encapsulation of Build Environments

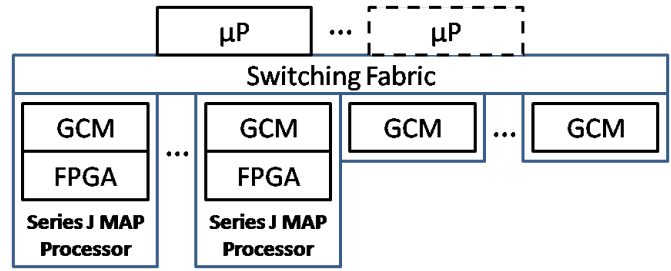
Even with a sufficiently integrated methodology for application kernel migration and integration to an FPGA system, overreliance on developer input during synthesis and PAR can reduce the applicability of FPGA systems for broader use in scientific computing. Assuming the preceding toolflow generated a correct and optimized application code, direct usage of these tools is unnecessary and potentially confusing to traditional application developers. Interaction with synthesis and PAR can provide an experienced developer with fine-grain control over low-level application details, but explicit interaction with these FPGA-specific tools should be optional when streamlined application development is desired.

Additionally, disconnects in toolflows for application development are not limited strictly to the generation of final hardware implementations. Regardless of manual or tool-assisted migration, application functionality is first verified (typically as a software emulation), then implemented or translated to HDL, evaluated in a simulator, and finally built into a full hardware design. Without the ability to progress from software emulation to hardware simulation and final implementation using a single application representation, FPGA systems can become prohibitively expensive in terms of real world code development and maintenance.

III. SRC COMPUTERS' CARTE TOOLFLOW

The SRC Carte toolflow and unified SRC-7 system architecture [7] are specifically designed to address the aforementioned limitations of disconnected methodologies for application migration to SRC FPGA-based MAP processors. The Carte application development toolflow includes a Code Development Assistant (CDA¹) and an architecture-aware compiler for rapid implementation using new or legacy C and Fortran. The

Fig. 3. SRC System Architecture



compiler and subsequent synthesis and PAR tools are tightly integrated with software emulation and hardware simulation for an encapsulated, two-command environment (i.e., “make” and “run”) for application design. The Carte runtime environment manages both the access control to system resources and interconnection bandwidth to help maximize the average communication bandwidth of multiple applications simultaneously.

A. Application Kernel Migration

The Carte toolflow provides several mechanisms for constructing applications with extremely minimal exposure to the underlying hardware specifics. This toolflow was designed to minimize developer effort by focusing on application migration through developer-familiar standard ANSI C and Fortran languages for the MAP processor. (Microprocessor code can remain in other languages.) The CDA tool can automatically replace minimally annotated software segments with highly optimized MAP processor functions. Alternatively, legacy codes can incorporate special functions and semantic mechanisms, such as pipelining, loop fusion, and loop unrolling, to exploit inherent parallelism in the application kernel while maintaining C or Fortran syntax. These optimizations include simple yet powerful functions for effective data movement to and from system resources. The Carte methodology defines MAP processor resources as the primary initiators of data movement through efficient direct memory access (DMA) or streaming between global memory and local resources.

B. System Integration

The macro-based library approach of the Carte toolflow allows effective usage of key architecture features of the system while hiding low-level implementation details. Memory resources exist in a flat global-address space, regardless of locality, with the system interconnect handling appropriate communication routing. MAP processors operate independently, synchronizing only as necessary based on barrier messages handled by the system interconnect. Consequently, application codes can achieve comparable performance whether using different resources in a system or different system configurations.

An SRC-7 system (Figure 3) corresponds to the second system architecture described in Section II-C. The system interconnect is comprised of a tiered switching fabric that provides full connectivity between FPGA and memory resources (i.e., a nonblocking crossbar switch). The current MAP

processor, the Series J, contains an Altera EP4SE530 FPGA for user applications. The memory resources, referred to as global common memories (GCMs), physically reside in pairs of banks on either a MAP processor or as an independent unit. However, each GCM bank is independently addressable and accessible by all MAP processors and microprocessors in the system. The Series J MAP processor physically combines the logically separate FPGA and memory resources to allow a mix of resources within a small footprint. The size and configuration of the overall system can be customized depending on application requirements.

C. Runtime Libraries

Application codes are organized into microprocessor functions (primarily for application coordination and other ancillary operations such as file I/O) and MAP functions for FPGA-based scientific computing. The runtime environment allows the main microprocessor function to request memory and system resources, and initiate the MAP processor-based computation with those resources. The environment is multi-user and prevents illegal actions such as memory accesses outside a valid address range by both the microprocessor and the MAP processors associated with an application. The switch-based interconnect further promotes multi-user behavior by packetizing and interleaving data transfers to minimize the average communication delay through the system. Under normal operation, system loading from multiple users cannot prevent the eventual successful completion of an application.

D. Encapsulation of Build Environments

A key feature of the Carte toolflow is the seamless transition of application development through functional emulation in software, simulation of hardware, and full implementation. This flow allows for extremely rapid (i.e., seconds to compile and run) functional testing during code development with standard debugging methods such as gdb and printf(). Once a prospective code is complete, simulation is available for cycle-accurate performance evaluation (i.e., typically minutes to compile and run). Normally, simulation is unnecessary for application development but is available for more detailed algorithm analysis without lengthy synthesis and PAR, or occupying real system resources. Only codes with suitable functionality and performance need hardware compilation and physical system testing.

Progressing through these tools only requires “make debug”, “make simulate”, or “make hw” commands from the user followed by “exec myfile.[dbg/sim/hw]” for running the unified executable. A unified executable is created that contains all necessary emulation binaries, hardware simulation components, or FPGA configuration files for debug emulation, hardware simulation, or MAP processor execution, respectively. The underlying tools are completely encapsulated by the Carte toolflow with any specific tool-related options defined in a singular makefile. This allows not only simplistic application management through the toolflow but also high portability as the underlying tools and/or resources evolve.

Fig. 4. String Matching in Software

```

1: uint64_t data, result, keyword, mask;
2: char *mem = (char *)malloc(numchars);
3: char *res = (char *)malloc(numchars)
4: uint64_t data, result, keyword, mask;
5: keyword = 0x48454C4C4f //ascii HELLO
6: mask = 0x000000FFFFFFFF; //5 char window
7:
8: for(i=0;i<numchars;i++)
9: {
10:     data = (data<<8) | mem[i];
11:     res[i] = (data & mask) == keyword;
12: }
```

IV. APPLICATION DEVELOPMENT

This section presents an example walkthrough of application migration and optimization for the SRC-7 system using the Carte toolflow. The target algorithm is a string-matching kernel used in areas such as packet inspection, data mining, and computational biology [2]. String matching involves locating instances of one or more key strings (also referred to herein as keywords) within a larger sequence of text. Section IV-A outlines the specific implementation of the string-matching algorithm. Section IV-B illustrates how the algorithm can benefit from streaming data for sustained pipelined execution. Section IV-C discusses an additional algorithm optimization which can further increase performance by attempting to match multiple strings in parallel. Quantitative results are presented in Section IV-D.

A. String-Matching Algorithm

This implementation of the string-matching algorithm provides an exhaustive search of the larger text for the keyword. Although precomputation of a substring index [1] for the keyword can increase the performance of microprocessor implementations, the advantages of indexing can be degraded on FPGA systems depending on memory latency. The software implementation presented in Figure 4 provides straightforward execution on microprocessors and FPGAs. The keyword for this example is “HELLO” (Line 5) and a mask (Line 6) isolates a five-character (i.e., the length of “HELLO” excluding the null terminator) window for comparison. For each character in the larger sequence of text, that character is shifted into the window of data (Line 10) and subsequently compared against the “HELLO” keyword (Line 11). The mask ensures that only the relevant characters are compared as the 64-bit datatypes can store upto eight characters. The core algorithm on Lines 10 and 11 involves only simple binary operations, which are highly amenable to FPGAs. The major algorithmic change involves migrating the input and resulting arrays (i.e., mem[] and res[], respectively) to the memory structure and dataflow of the SRC system.

B. Application Migration

As described in Section III-A, streaming provides a mechanism for the resources on a MAP processor to access the

Fig. 5. String Matching in Carte

```

1: Stream_64 S64_in, S64_out;
2: Stream_8 S8_in, S8_out;
3:
4: #pragma src parallel sections
5: {
6:   #pragma src section
7:   {
8:     //GLOBAL MEMORY TO FPGA INPUT STREAM
9:     streamed_dma_gcm(&S64_in, ...);
10:  }
11:
12: //INPUT STREAM FORMATTING
13: //QWORDS (64 bit) TO CHARS (8 bit)
14: ...
15:
16: #pragma src section
17: {
18:   for (i=0;i<numchars;i++)
19:   {
20:     get_stream(&S8_in, &mem);
21:     data = (data<<8) | mem;
22:     res = (data&mask) == keyword
23:     put_stream(&S8_out, res, 1);
24:   }
25: }
26:
27: //OUTPUT STREAM FORMATTING
28: //CHARS (8 bit) TO QWORDS (64 bit)
29: ...
30:
31: #pragma src section
32: {
33:   //FPGA OUTPUT STREAM TO GLOBAL MEMORY
34:   streamed_dma_gcm(&S64_out, ...);
35: }
36: }

```

system’s global memory. Streaming provides a low overhead mechanism for sequential data input and output consistent with the requirements of the string-matching algorithm. The Carte implementation, Figure 5, streams the larger sequence of text to a pipelined implementation of the data windowing and comparison algorithm. The specification of the core algorithm, Lines 21 and 22, remains nearly unchanged from the original software code. The Carte compiler can pipeline this algorithm, automatically stalling or proceeding based on the availability of data on the input stream and capacity of the output stream. The additional Carte code connects the algorithm streams to the global common memory with additional formatting between the native width (and endianness) of the memory, 64 bits, and the character datatype, 8 bits. The “src section” pragmas define the concurrent behavior of the streaming I/O and the algorithm execution. The CDA tool can assist in constructing the formatted character streams based on simple annotations of the relevant arrays in the original software code, thereby making common hardware optimizations even more accessible to application developers.

C. Additional Optimization

The initial hardware migration focused primarily on temporal parallelism where the input stream loads the next character

Fig. 6. Optimization - Multiple String Matches

```

1: #pragma src parallel sections
2: {
3:   //GLOBAL MEMORY TO INPUT STREAM + FORMATTING
4:   ...
5:
6:   //STREAM FANOUT: S8_in to S8_in1 and S8_in2
7:   ...
8:
9:   //KEYWORD1: S8_in1
10:  #pragma src section
11:  {
12:    ...
13:    get_stream(&S8_in1, &mem);
14:    ...
15:  }
16:  //KEYWORD2: S8_in1 ...
17:  //KEYWORD3: S8_in1 ...
18:  //KEYWORD4: S8_in1 ...
19:
20:  //KEYWORD5: S8_in2
21:  #pragma src section
22:  {
23:    ...
24:    get_stream(&S8_in2, &mem);
25:    ...
26:  }
27:  //KEYWORD6: S8_in2 ...
28:  //KEYWORD6: S8_in2 ...
29:  //KEYWORD6: S8_in2 ...
30:
31:  //MERGE RESULTS
32:  ...
33:
34:  //FORMATTING + OUTPUT STREAM TO GLOBAL MEMORY
35:  ...
36: }

```

while the string-matching kernel operates on the current character. The algorithm can also benefit from spatial parallelism with multiple distinct keywords compared against the larger sequence of text. Streams allows for efficient duplication of the data to support a number of parallel kernels with minimal overhead. Figure 6 describes simultaneous string matching with eight keywords. To help minimize overhead delays such as fanout, the original input stream, S8_in, is replicated into streams S8_in1 and S8_in2. Because the actual output of each kernel is a singular bit representing a match or no match for each position in the larger sequence of text, the individual bit from each of the eight kernels can be merged into one character stream for more compact output. Again, the core algorithm is mostly unchanged and with proper annotation, the CDA tool can also assist in stream duplication and merging.

Further optimization to this implementation could involve the source and destination of the data streams. The current streaming I/O targets global common memory either on or external to the MAP processor, with currently available bank sizes ranging from 1GB to 16GB depending on location and configuration. This implementation assumes the larger sequence of text in the global common memory is first populated from some other source in the system. If the text originates from a file, the MAP processor implementation could stream

TABLE I
TOOLFLOW PERFORMANCE - 1MB TEXT

	Compilation Time (s)	Execution Time (s)
Software Emulation	2.6E+0	1.0E+2
Physical Hardware	4.2E+3	3.2E-1

TABLE II
STREAMING AND PIPELINING EFFICIENCY

Size (B)	Clock Cycles	Efficiency (%)
1K	1396	73.3
10K	10617	96.4
100K	102772	99.6
1M	1048953	99.9+

directly from microprocessor memory instead with no change to the streaming dataflow of the algorithm. Additionally, the text could originate from a network interface connected to the MAP processor via the general purpose input/output (GPIO) interface, which can support an infinite stream of text and results without intervention or input from the microprocessor.

D. Quantitative Results

Table I summarizes the computation and executions times for software emulation and the physical hardware implementation of the string-matching algorithm on a 1MB sequence of text streaming through global common memory. At 2.6 seconds, the compilation time for software emulation is analogous to compilation of conventional microprocessor codes and therefore rapid enough for frequent evaluation of revisions to an algorithm design. An execution time of 100 seconds is not trivial but still sufficiently short to allow for iterative functional testing. In contrast to emulation, the actual implementation requires over 4000 seconds for compilation, but, as expected, requires considerably less execution time.

Table II describes the efficiency of the streaming and pipelining in the string-matching algorithm. This implementation processes one character of the larger sequence of text per clock cycle, excluding system latency. Longer streams of characters better amortize this latency leading to higher efficiency. Consequently, 1KB of text (i.e., 1024 characters) is likely an insufficient volume of text for maximizing performance as only 73.3% of clock cycles perform actual computation. In contrast, sequences of text of 100KB and above have over 99% efficiency and therefore minimize execution time. This predictability of performance for sufficiently large data streams can allow for more accurate analysis and evaluation of algorithm designs prior to implementation.

V. CONCLUSIONS

Perceived limitations of FPGA systems have been associated with lengthy and disconnected toolflows for application development. Efficient migration to FPGA systems historically requires straightforward algorithm specification, compilation, and execution. The SRC Carte toolflow and unified SRC-7 system architecture provide C and Fortran syntax for kernel specification, abstract yet efficient integration with system

resources, robust runtime support, and complete encapsulation of low-level FPGA tools. Resulting applications require only two simple, Unix-style commands for compilation and execution. The string-matching case study demonstrated that a high-performance applications can maintain specifications analogous to legacy software while capitalizing on a streaming dataflow for FPGA systems. The compilation and execution times allow for iterative development cycles of functional refinement through software emulation followed by hardware simulation and generation of the final design implementation. Sufficiently large streams of 100KB or more demonstrated that over 99% of their clock cycles perform actual computation.

REFERENCES

- [1] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Comm. ACM*, 20(10):762–772, 1977.
- [2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. MIT Press and McGraw-Hill, 2001.
- [3] H. Lam, G. Stitt, et. al. Reconfigurable supercomputing: Performance, productivity, and sustainability (panel session). In *Proc. Engineering of Reconfigurable Systems and Algorithms (ERSA)*, July 13 2010.
- [4] I. Gonzalez and E. El-Araby and P. Saha and T. El-Ghazawi and H. Simmler and S. Merchant and B. Holland and C. Reardon and A. George and H. Lam and G. Stitt and N. Alam and M. Smith. Classification of application development for fpga-based systems. In *Proc. National Aerospace Electronics Conference (NAECON)*, July 16–18 2008.
- [5] S. Merchant and B. Holland and C. Reardon and A. George and H. Lam and G. Stitt and M. Smith and N. Alam and I. Gonzalez and E. El-Araby and P. Saha and T. El-Ghazawi and H. Simmler. Strategic challenges for application development productivity in reconfigurable computing. In *Proc. National Aerospace Electronics Conference (NAECON)*, July 16–18 2008.
- [6] SRC Computers, LLC. www.srccomputers.com.
- [7] SRC Computers, LLC. Introduction to the SRC-7 MAPstation system, 2009.