BAE SYSTEMS

ALGORITHM ANALYSIS AND MAPPING ENVIRONMENT FOR ADAPTIVE COMPUTING SYSTEMS

FINAL TECHNICAL REPORT

25 September 2001

Submitted by:	BAE Systems, Information and Electronic Warfare Systems P.O. Box 868 Nashua, NH 03061-0868
Under Contract:	F33615-97-C-1174
As:	CLIN 0001, Data Item A001
For:	**Air Force Research Laboratory/SNAS, Building 23** 2010 Fifth Street Wright Patterson Air Force Base, OH 45433-7301
	Defense Advanced Research Projects Agency Information Technology Office 3701 North Fairfax Drive Arlington, VA 22209-2308

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE April 2, 2001	3. REPORT TYPE AND DATES COVERED Final Report, October 17, 1997 to September 28, 2001

4. TITLE AND SUBTITLE
Algorithm Analysis and Mapping Environment for Adaptive Computing Systems, Final Technical Report

5. FUNDING NUMBERS
Program Funding,
Contract Number:
F33615-97-C-1174

6. AUTHOR(S)
Jphn C. Zaino

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
BAE SYSTEMS (Formerly Sanders, A Lockheed Martin Co)
PO Box 868
Nashua NH
03061-0868

8. PERFORMING ORGANIZATION REPORT NUMBER
CLIN 0001, CDRL A001

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory / SNAS	**Defense Advanced Research Projects Agency**
28 Electronics Parkway	Information Technology Office
Rome NY, 13441-4514	3701 North Fairfax Drive
	Arlington, VA 22209-2308

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
CLIN 0001, CDRL A001

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT
DOD
The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Projects Agency or the U.S. Government.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Under the Algorithm Analysis and Mapping Environment program over a 4 year period, a team at BAE SYSTEMS has developed an integrated algorithm analysis and mapping environment for translating a dataflow representation of a signal processing algorithm into an Adaptive Computing System (ACS) consisting of field programmable gate array (FPGAs) devices. This environment allows designers to transform signal processing algorithms into FPGA-based hardware an order of magnitude faster than is currently possible. Our approach was to focus on three areas of capability critical to the success of adaptive computing: algorithm analysis, algorithm mapping, and smart generators. These capabilities take advantage of the special characteristics of signal processing algorithms to reduce the time to field the ACS implementation, and were implemented as extensions to the Ptolemy design environment (http://ptolemy.eecs.berkeley.edu) developed at the University of California, Berkeley.

14. SUBJECT TERMS
Adaptive Computing, FPGA, Reconfigurable, Algorithm Mapping, Algorithm Analysis, Modeling, Simulation, Ptolemy, ACS Tools, DSP

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	

Table of Contents

List of Figures

List of Tables

Abstract

Over the past four years, a team at BAE SYSTEMS has developed an integrated algorithm analysis and mapping environment for translating a dataflow representation of a signal processing algorithm into an Adaptive Computing System (ACS) consisting of field programmable gate array (FPGAs) devices. This environment allows designers to transform signal processing algorithms into FPGA-based hardware an order of magnitude faster than is currently possible. Our approach has been to focus on three areas of capability critical to the success of adaptive computing: algorithm analysis, algorithm mapping, and smart generators. These capabilities take advantage of the special characteristics of signal processing algorithms to reduce the time to field the ACS implementation, and were implemented as extensions to the Ptolemy design environment (http://ptolemy.eecs.berkeley.edu) developed at the University of California, Berkeley.

Algorithm implementation for ACS requires careful consideration of the appropriate signal representation and the costs of operations. The algorithm analysis capabilities developed on this program reduce the effort required to find good ACS implementation choices for a signal processing algorithm. The environment provides algorithm designers with information about operation counts, including adds, multiplies, and memory accesses, and with analyses of quantization effects related to ACS implementations. For many DSP problems, reduced precision arithmetic will maintain acceptable system performance. A mapping of an algorithm to an FPGA architecture will be successful if the designer can limit wordlength growth without sacrificing algorithm performance. Wordlength reduction introduces noise into the data stream, so the designer must balance the need for an efficient implementation with output quality. Designing optimal wordlength combinations for dataflow graphs can be very difficult, due to the exponential number of wordlength combinations that must be considered. Naive random sampling approaches are ineffective for large flowgraphs, because most of the samples do not correspond to feasible or useful designs.

In our work, we use a Markov chain Monte Carlo (MCMC) sampling approach [1,2] to finding good wordlength combinations for DSP flowgraphs. We define a feasible region of wordlengths that consists of designs near the Pareto-optimal cost/quality boundary which also satisfy any feasibility constraints. The MCMC sampler is used to generate uniform samples from this feasible region by performing a random walk on a specially constructed Markov chain; this approach solves a problem that can be very difficult using only naive sampling approaches. Additionally, the MCMC method offers extreme simplicity in the software coding of the algorithm, in contrast with dynamic programming approaches. With these techniques and capabilities, an algorithm designer can quickly determine the appropriate number of bits for signal representations at all points in the design and quantify the performance of various implementation choices.

Signal processing algorithm mapping for ACS also involves assigning functions to different processing elements. On this program we have developed mapping techniques

for ACS tailored to signal processing. These capabilities include performance analysis, multi-device partitioning assistance, and automatic scheduling. The scheduling and partitioning functions use the coarse-grain nature of signal processing dataflow graphs to help optimize partitioning for ACS. Our tools automatically generate the VHDL code needed to translate data paths across different FPGA devices.

Current methods for logic generation for ACS are either built around libraries of functions or around general-purpose logic synthesis. As part of this effort, we implemented "smart generators" that are extensions of the concept of a parameterized library. These generators are tailored to signal processing functions and include rules that capture specific implementation techniques and trade-offs. For example, a smart generator for a complex multiplier is able to trade between a three-multiplier implementation and a four-multiplier implementation according to area and latency constraints. Additionally, we have provided mechanisms to automatically generate both hardware and software interfaces for the resultant ACS. Our initial target system was a Xilinx XC4062XL-based Wildforce™ board from Annapolis Micro Systems http://www.annapmicro.com. Additional efforts were also expended towards augmenting our tools to support a Xilinx Virtex™ XCV1000-based Wildstar™ board, also from Annapolis Micro Systems.

The algorithm analysis, mapping, and logic generation capabilities have been developed as extensions to Ptolemy. Ptolemy provides a well documented, object-oriented, open software architecture with implementations in C++ and Java. Our extensions to Ptolemy have been captured in a new ACS domain that separates the interface specification from implementation for each signal processing functional block. The algorithms of interest to this project are represented by dataflow graphs comprised of these functional blocks, following a synchronous dataflow model of computation. We have used a Corona/Core architecture, where each block has a common interface known as the Corona, and one or more implementations, known as the Cores. A retargeting mechanism allows the users to change Cores and hence implementation, which moves the dataflow graph between various simulation models (floating point, fixed point) and implementations (C code, VHDL code).

These ACS tools have been used to automatically implement several applications. The first was a Winograd Discrete Fourier Transform (DFT) as part of a channelized frequency shift-keyed (FSK) receiver in ACS. The Winograd algorithmic structure has the minimum number of multiplications for any DFT approach, and is thus ideal for FPGA implementation. The tools have also been used to develop an FPGA implementation of a high speed linear FM detector. We used our tools to also implement a FFT-based complex correlator, requiring the use of two 16-point complex FFT cores in series with a cornerturn to obtain a 256-point spectrum. Most recently, we utilized our tools to accelerate a high resolution radar template-matching application. In all cases, our ACS tools were used to simulate the algorithm, select appropriate fixed point representations, and generate the VHDL implementations. The final FPGA designs were obtained by synthesizing the VHDL and performing place and route with commercial tools. The ACS domain is part of Ptolemy Classic and includes these ACS capabilities and selected demonstrations.

1 The Need

The "Algorithm Analysis and Mapping Environment for Adaptive Computing Systems" is a DARPA ITO/Wright Labs effort to develop tools and a software environment that significantly enhances the ability of a designer to develop adaptive computing systems. Adaptive computing systems, e.g. systems which use reconfigurable logic for computation, have the potential for leap ahead improvement in performance, measured in operations per watt or operations per cubic inch, for a range of signal processing applications. However, the current tools and environments for developing such systems are not sufficient to allow the full power of adaptive computing technology to be developed. Typically, hardware design expertise is required to get any working adaptive computing system implementation. Generation of a good implementation requires many time-consuming iterations between an algorithm designer and a hardware developer. Table 1.0 illustrates both the potential of adaptive computing technology as well as the development time issue.

Application	Performance Benefit	Development Limitations
COTS Midband Algorithm	Reconfigurable implementation performed 25 x more frames per second for same hardware area.	Implementation was twice as long as software implementation. Total development time measured in weeks.
Communications	3 Billion operations per second correlator implemented in a single MCM package. Typical power dissipation is 5 watts.	Implementation of extremely regular structure still took multiple days
Digital Receiver	Sixteen channel filter bank on 320 MHz data implemented on two MCMs.	Weeks to implement. Required knowledge of specialized FFT structure.

Table 1.0. Performance benefits and development limitations of adaptive computing system implementations.

2 Objective

Our objective in this program was to dramatically reduce the time required to implement an adaptive computing system. Our goal was to reduce the time to develop an initial working design from days to minutes and the time to implement an optimized solution from weeks to days. Our goal was to make this capability available to the algorithm designer who lacks significant hardware development background. It was our objective to demonstrate an order of magnitude reduction in the time required to map a military signal processing algorithm onto an adaptive computing system. The objective was to work towards these goals by developing new algorithm analysis and mapping capabilities and by implementing these capabilities in an integrated development environment. These development efforts were to include the following capabilities: a) capabilities that allow a signal processing algorithm designer to analyze their algorithm for characteristics specifically suited, or ill-suited, for implementation in adaptive computing systems; b) capabilities that take advantage of the special structure of many signal processing algorithms to ease mapping algorithms to adaptive computing devices; and c) capabilities that enable device specific and function-specific implementation generators. These capabilities and the environment were demonstrated throughout the program on signal processing algorithms with military applications.

The objective was to implement the capabilities and environment for this program in a form that is readily accessible to the adaptive computing system community. Our work was to make adaptive computing systems more readily accessible to algorithm developers and to speed the development of military signal processing applications. The environment developed was to be an extensible environment to which other researchers would be able to add to.

Specifically, the AAME program objectives (from Statement of Work) were to:

Algorithm Analysis

- Develop capabilities that assist a signal processing algorithm developer in analyzing a signal processing algorithm for implementation in an adaptive computing system. These capabilities shall be implemented as a set of software tools that operate on signal flow graphs. Capabilities shall include at least the following:

    - Develop the capability to propagate derived information about algorithm wordlengths and formats based on wordlengths and formats of input data and coefficients.

    - Develop the capability to track statistical information through an algorithm. This capability shall support comparison of a conventional implementation with reduced complexity implementations.

    - Develop the capability to both simulate and test real and synthetic data sets in complicated algorithms. This capability shall enable algorithm performance analysis and trade-off analyses.

    - Develop the capability to explore alternative algorithmic rearrangements. This capability will allow multiple algorithmic implementations to be tested automatically by the system to find the most efficient rearrangement for a given application and technology.

Algorithm Mapping

- Develop a capability to represent a synchronous data-flow algorithm in a model that captures the mapping of that algorithm to hardware components. Through an iterative process, the user shall be able to find efficient mappings of algorithms to components, both in space and time. These capabilities shall include the following:

- Design and populate a hierarchical library of hardware components. Representations for device-specific components shall be developed and a mechanism for building architectures from these components shall be provided.

- A graphical user interface shall be used to group components of the algorithm and assign them to hardware devices. The designer shall be given feedback on the performance of the partitioning in terms of utilization, throughput, and efficiency, among other metrics. The capability to recommend efficient mappings shall be developed.

- Extend the concepts of multi-processor scheduling to include scheduling of hardware contexts for dynamically reconfigurable devices.

Smart Generators

- Develop the capability to generate device and function-specific implementations. These capabilities shall include the following:

  - Develop and integrate generators for basic digital signal processing (DSP) operations applicable to at least two distinct reconfigurable logic target technologies. Enough basic building blocks shall be provide to produce usable hardware for the demonstrations.

  - Develop and integrate recursive hierarchical generators which allow for multiple algorithmic implementations to be compared for optimality in the targeted technology.

  - Develop and integrate hardware/software interfaces that de-couple the interface from implementation. Large data sets and flow control shall be considered.

  - Incorporate general purpose logic synthesis as a mechanism for extending capabilities to new devices.

Software Architecture and Integration

- Integrate algorithm analysis, algorithm mapping, and smart generators technology into the software environment. All software developed on this program shall be designed to interface standards consistent with the software environment and shall be distributed according to existing procedures, which make the software, as is, available freely to all interested parties. Incremental software releases shall be provided on a regular basis. Software documentation shall be provided.

Demonstrations

- Demonstrate the capabilities and software developed on this program after each software release. Libraries shall be populated and generators developed as appropriate for each demonstration. Capabilities shall be demonstrated in each of the three areas of algorithm analysis, mapping, and smart generators for each demonstration. These demonstrations shall include the following signal processing applications:

  - Demonstrate the ability to support common communication structures including basic mathematics, filtering, and quantization

  - Demonstrate multiple kinds of signal processing operations, including filtering, Constant False Alarm Rate (CFAR) detection, parameter matching and decision processing at different rates. The hierarchical generation capability of the smart generators task shall be used to create implementations of critical operations.

  - Demonstrate processing of large multidimensional data sets using image processing or ATR algorithms developed under the MSTAR program.

  - Demonstrate the retargetability of its approach through a multi-function demonstration of MSTAR modules. The demonstration shall include multiple functions mapped to the same hardware and the demonstration shall include high level symbolic rearrangement, partitioning, and automatic scheduling of algorithms.

Technical Reports/Reviews

- Exercise good management practices to monitor the technical progress of the AAME program by reviewing monthly progress against cost and schedule plans. Prompt communication of any issues affecting progress shall be made. Periodic program reviews shall be conducted. Results of this effort shall be described in papers and conference presentations. Periodic reports summarizing the technical progress as well as a final report shall be delivered.

3 Approach

Our approach to meeting the challenging objectives of this program consisted of four major thrusts:

The first was to focus on a system that starts with a high level language input and restrict ourselves to a limited application domain. In our case we chose the domain of dataflow signal processing algorithms. We used a high level system, the Ptolemy system, from the University of California, Berkeley as our algorithm representation and manipulation environment.

The second thrust was to support algorithm development at the bit level by combining analytical and simulation methods to help the algorithm developer optimize performance. This included development of methods for providing the algorithm designer with feedback about the implications of algorithm implementation decisions. We included in this capability provisions to rearrange algorithms as required by implementation requirements.

The third thrust was to leverage the structure of signal processing algorithms to improve algorithm mapping. We took advantage of the coarse-grain nature of signal processing dataflow graphs to improve partitioning of algorithms in space and time. We also investigated the use of scheduling tools, developed for mapping signal processing dataflows onto programmable processors, to develop hardware schedulers that are used in adaptive computing system implementation.

The fourth thrust included providing a library of signal processing building blocks, with their hardware implementations, and the ability to rapidly compose these blocks as well as the ability to automatically generate the software to drive the hardware.

Figure 1 shows how the capabilities that this program have provided are used to develop an adaptive computing system. In our approach an algorithm is entered into the system. Our approach represents signal processing dataflow graphs within the Ptolemy system. We support direct entry of algorithms within that environment. The system supports floating point simulations to set a performance baseline of an algorithm. Bit width analysis and noise distribution analysis are performed and a fixed point version of the algorithm can be simulated. The results of fixed point simulation can be compared to the floating point simulation and the process iterated until a good tradeoff between precision and performance is realized. This iteration can be augmented by the manual rearrangement of the algorithm to provide alternative implementations. For example, multiple implementations of an FIR filter can be investigated.

Following algorithm analysis the resulting signal processing algorithm dataflow graph is mapped to an adaptive computing resource node. We have considered adaptive computing resource nodes with multiple reconfigurable devices and with the capability to be dynamically reconfigured. The first step of the mapping work is to predict performance of the algorithm on the target hardware. This prediction can flow back up into algorithm analysis and cause alternative algorithms to be explored. Automatic scheduling tools are used in the mapping phase to assist in the partitioning of the signal processing dataflow graph and the mapping of those partitions in space and time over multiple dynamically reconfigurable devices.

The final step in the process is the generation of the detailed programming, both for the reconfigurable devices as well as for the programmable processor that is controlling the adaptive computing resource. The entire process is driven by a set of libraries that provide a common database to present signal processing elements. Included in this approach is the idea that logic generators can be “smart,” e.g. changing their

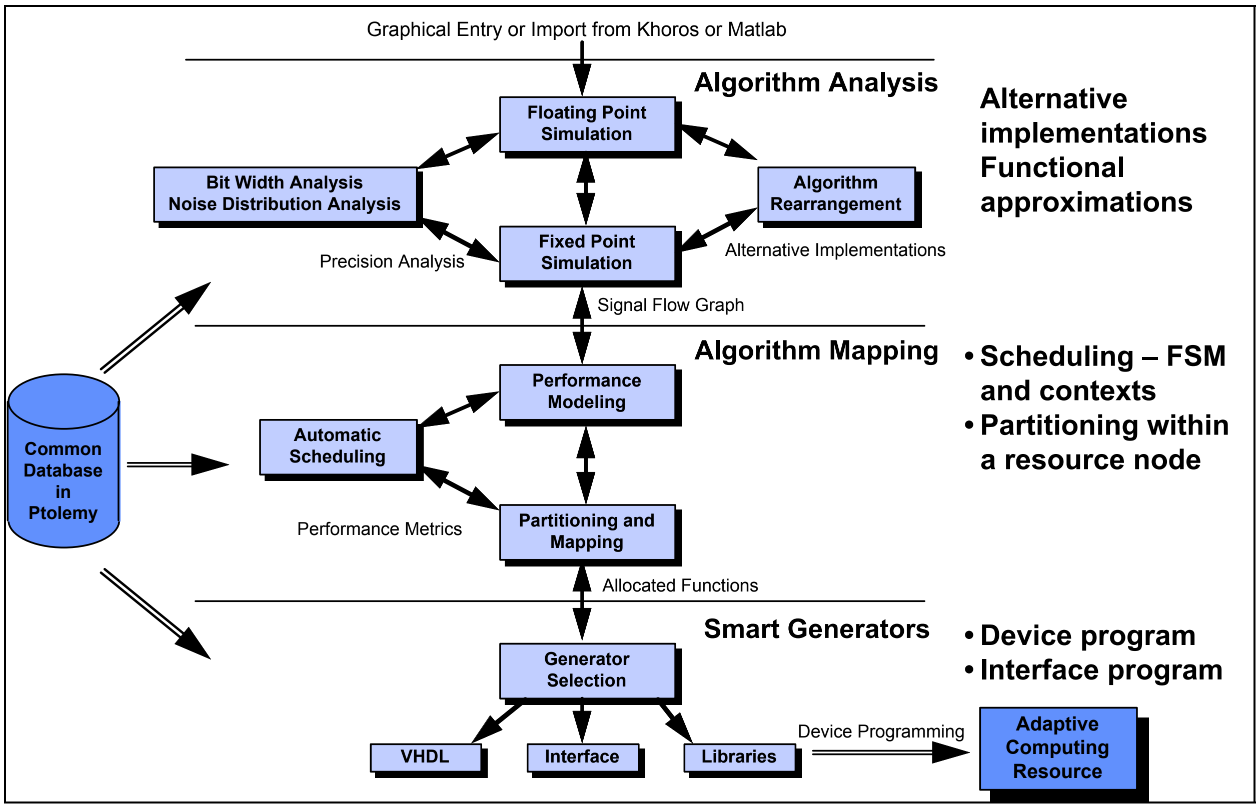


Figure 1. Use of Capabilities to Develop an Adaptive Computing System.

implementation according to desired design characteristics such as maximum throughput or minimum power.

Figure 2 below expands the above design process to include how the designed system flows into run time execution. As can be seen, once the logic generation is complete from the use of the smart generators, commercial tools are used for the place and routing and hardware bitstream configuration file generation. In the run time environment, the hardware configuration is used (loaded to the device) in conjunction with the application software, host operating system and actual device to run the application.

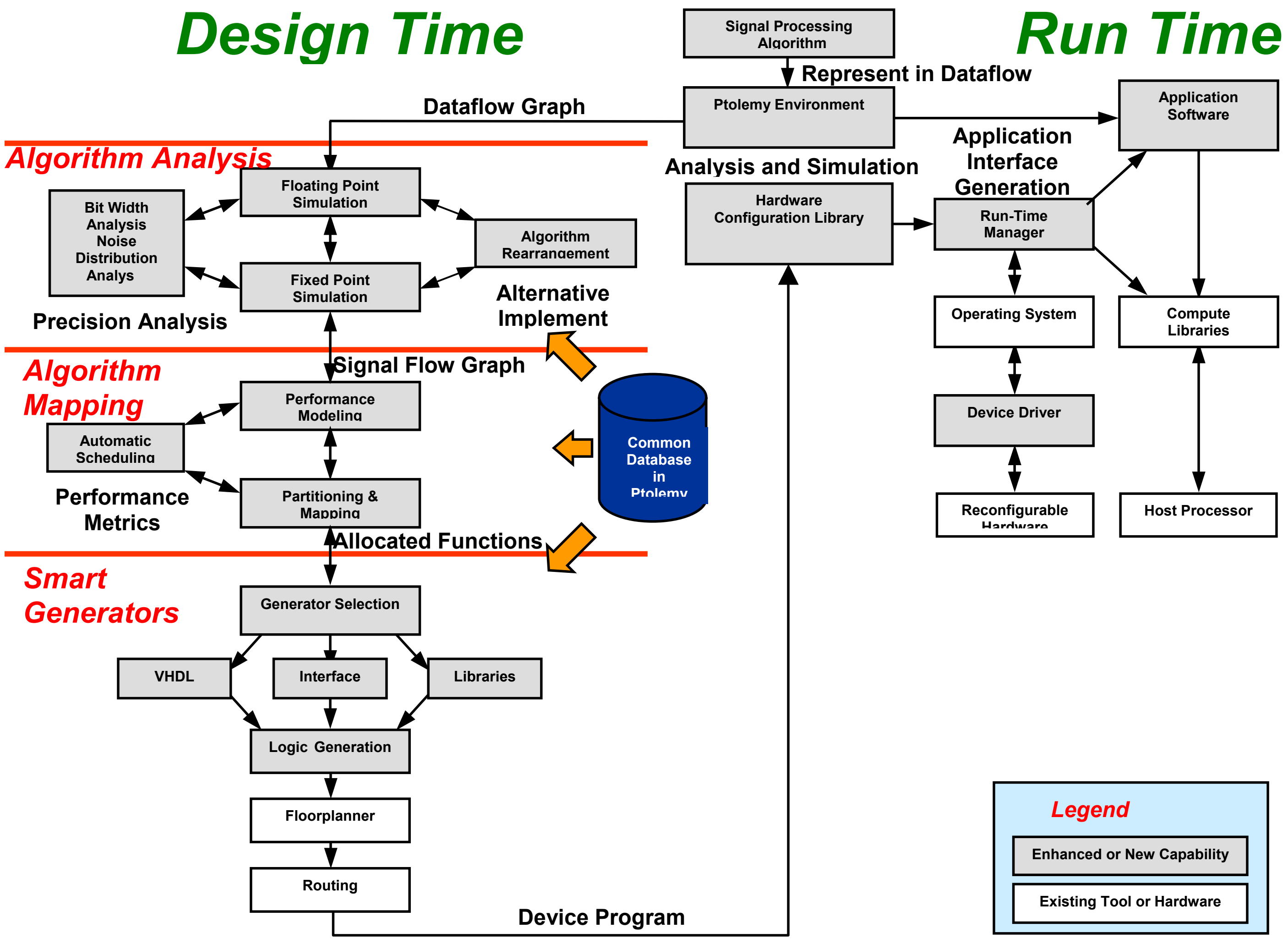


Figure 2. Design Time and Run Time Environments for an Adaptive Computing System Implementation Using the AAME Tools.

4 Progress

Over the past four years, the BAE Systems team members involved in this program have successfully developed an integrated algorithm analysis and mapping environment for translating a dataflow representation of a signal processing algorithm into an Adaptive Computing System (ACS) consisting of field programmable gate array (FPGAs) devices. This environment now allows designers to transform signal processing algorithms into FPGA-based hardware an order of magnitude faster than is currently possible. Our approach was to focus on three areas of capability critical to the success of adaptive computing: algorithm analysis, algorithm mapping, and smart generators. These capabilities take advantage of the special characteristics of signal processing algorithms to reduce the time to field the ACS implementation, and are being implemented as extensions to the Ptolemy design environment (http://ptolemy.eecs.berkeley.edu) developed at the University of California, Berkeley.

The algorithm analysis capabilities developed on this program significantly reduce the effort required to find good ACS implementation choices for a signal processing algorithm. The environment provides algorithm designers with information about operation counts, including adds, multiplies, and memory accesses, and with analysis of quantization effects related to ACS implementations. The tools produced under this program help ensure mappings of an algorithm to an FPGA architecture are successful with regards to limiting wordlength growth without sacrificing algorithm performance. Optimal wordlength combinations for dataflow graphs can be determined quickly using the tools developed. In our work, we use a Markov Chain Monte Carlo (MCMC) sampling approach [1,2] to finding good wordlength combinations for DSP flowgraphs. We define a feasible region of wordlengths that consists of designs near the Pareto-optimal cost/quality boundary which also satisfy any feasibility constraints. The MCMC sampler is used to generate uniform samples from this feasible region by performing a random walk on a specially constructed Markov chain; this approach solves a problem that can be very difficult using only naive sampling approaches. Additionally, the MCMC method offers extreme simplicity in the software coding of the algorithm, in contrast with dynamic programming approaches. With these techniques and capabilities, an algorithm designer can quickly determine the appropriate number of bits for signal representations at all points in the design and quantify the performance of various implementation choices.

On this program we have successfully developed mapping techniques for ACS tailored to signal processing. These capabilities include performance analysis, multi-device partitioning assistance, and automatic scheduling. The scheduling and partitioning functions use the coarse-grain nature of signal processing dataflow graphs to help optimize partitioning for ACS. Our tools automatically generate the VHDL code needed to translate data paths across different FPGA devices.

To augment the normal methods of logic generation for ACS, i.e. either built around libraries of functions or around general-purpose logic synthesis, we successfully implemented “smart generators”. The smart-generators exist now as extensions to the concept of a parameterized library. These generators are tailored to signal processing functions and will include rules that capture specific implementation techniques and trade-offs. For example, a smart generator for a complex multiplier is able to trade between a three-multiplier implementation and a four-multiplier implementation according to area and latency constraints. Additionally, we have provided mechanisms to automatically generate both hardware and software interfaces for the resultant ACS. Our initial and primary target system has been a Xilinx XC4062XL-based Wildforce™ board from Annapolis Micro Systems http://www.annapmicro.com. We also augmented our tools to provide limited Virtex technology support, such as that used with a Xilinx Virtex™ XCV1000-based Wildstar™ board, also from Annapolis Micro Systems.

The algorithm analysis, mapping, and logic generation capabilities have been developed as extensions to Ptolemy. Ptolemy provides a well documented, object-oriented, open software architecture with implementations in C++ and Java. Our extensions to Ptolemy have been captured in a new ACS domain that separates the interface specification from implementation for each signal processing functional block. The algorithms of interest to this project are represented by dataflow graphs comprised of these functional blocks, following a synchronous dataflow model of computation. We have used a Corona/Core architecture, where each block has a common interface known as the Corona, and one or more implementations, known as the Cores. A retargeting mechanism allows the users to change Cores and hence implementation, which moves the dataflow graph between various simulation models (floating point, fixed point) and implementations (C code, VHDL code).

The ACS tools have been used to automatically implement several applications along the various stages of development of the tools capabilities. These have included four main demonstrations and several others. The four main demonstrations included:

- A Winograd Discrete Fourier Transform (DFT) as part of a channelized frequency shift-keyed (FSK) receiver in ACS. The Winograd algorithm structure has the minimum number of multiplications for any DFT approach, and was thus ideal for FPGA implementation. This demonstration was shown at the April 1999 DARPA ACS PI meeting.

- A signal detection scheme (template matching) using an FFT-based correlator. This implementation involved an FFT-based complex correlator, requiring the use of two 16-point complex FFT cores in series with a cornerturn to obtain a 256-point spectrum. This demonstration was shown at the April 2000 DARPA ACS PI meeting.

- Initial workings/progress towards accelerating a high resolution radar template-matching application (System-Oriented High Range Resolution Automatic Recognition Program (SHARP)). This demonstration was shown at the October 2000 DARPA ACS PI meeting.

- Full implementation of the accelerated SHARP algorithm on multiple FPGA devices.

Additional demonstrations developed using the ACS tools as the program and tools capabilities progressed included comparison of fixed point vs floating point implementation/results for smaller algorithms (e.g. a “Butterfly” generation algorithm), the first demonstration of the tools automatic implementation of multi-FPGA use (for a magnitude calculation), and the development of an FPGA implementation of a high speed linear FM detector (for the Reconfigurable Algorithms for Adaptive Computing (RAAC) program).

In all cases above, the ACS tools were used to simulate the algorithm, select appropriate fixed-point representations, and generate the VHDL implementations. The final FPGA designs were obtained synthesizing the VHDL and performing place and route with commercial tools.

Along with the above applications/demonstrations, there have been four software releases of the ACS tools that incorporated enhanced tools capabilities developed up to the point of the software release. The first release was in June 1998, the second in April 1999, the third in August 2000 and the fourth in July 2001. Each was released as an update to the ACS domain of the UC Berkeley Ptolemy Classic project (delivered to UCB for incorporation into Ptolemy source code).

The sections that follow describe in detail the progress achieved for each development effort under the AAME program.

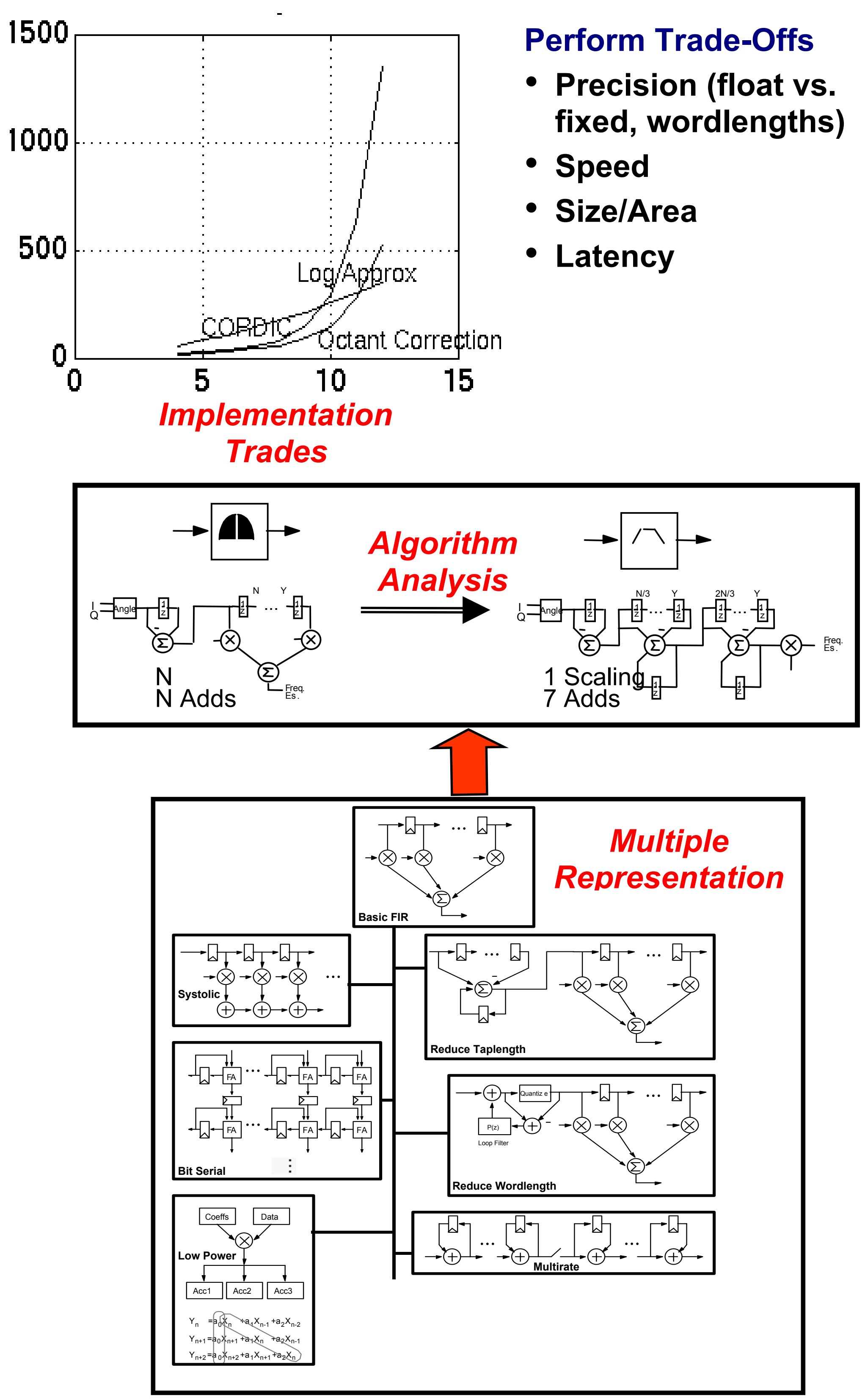
4.1 Algorithm Analysis and Mapping

The two development areas of algorithm analysis and mapping can be distinctive in areas of theoretical usage in the implementation of an algorithm to an ACS system. For example, the areas associated with algorithm analysis can be categorized as SNR analysis, alternative implementations and functional approximations. Algorithm mapping can be considered related to timing and sizing estimation, scheduling, and partitioning of the algorithm pieces across multiple FPGAs. This is generally what was expected during the initial development work. However, as development progressed, it was apparent that the two are much more closely related, one affecting the outcome or iteration of the other. For this reason, this section discusses the development efforts and results of the algorithm analysis portion along with the mapping efforts.

As a result of the efforts in these areas, design space exploration has been automated. In this regard, we have developed and applied bit width optimization theory (Markovian modeling) for algorithm analysis and implemented a bit width optimization tool to trade signal to noise ratio (SNR) versus hardware complexity. In addition, we have implemented pipeline alignment and scheduling algorithms for signal processing dataflow graphs. These latter capabilities automatically generate "algorithm-specific" sequencer and memory control logic, handle uni-rate and multi-rate signal processing, and allow for single and multi-FPGA implementations.

4.1.1 Algorithm Analysis

There are often approximations that can be made in order to improve the efficiency of an algorithm. These could include variable wordlengths and data representation, and approximations to functions in order to reduce system complexity. Figure 3 below shows some of the trade-offs that should be considered, such as precision, speed, size/area and latency.



Perform Trade-Offs

- Precision (float vs. fixed, wordlengths)
- Speed
- Size/Area
- Latency

$Y_n = a_0X_n + a_1X_{n-1} + a_2X_{n-2}$
$Y_{n+1} = a_0X_{n+1} + a_1X_n + a_2X_{n-1}$
$Y_{n+2} = a_0X_{n+2} + a_1X_{n+1} + a_2X_n$

Figure 3. Algorithm Trade-Offs

Although custom wordlengths reduce HW requirements for a given algorithm and reduces unecessary switching and power dissipation. However, a DSP system now becomes an approximation yieldig degraded results. Therefore, the goal was to find good wordlength combinations satisfying the criteria that 1) the design fits in an FPGA and 2) acceptable performance is maintained. The following example in Figure 4 for a complex multiply demonstrates factors that were considered in developing the wordlength analysis tools.

Operation: (e+jf)=(a+jb)(c+jd)
Implement as: t = (a-b)d
e = t+(c-d)a
f = t_(c_d)b
Signal Flowgraph:

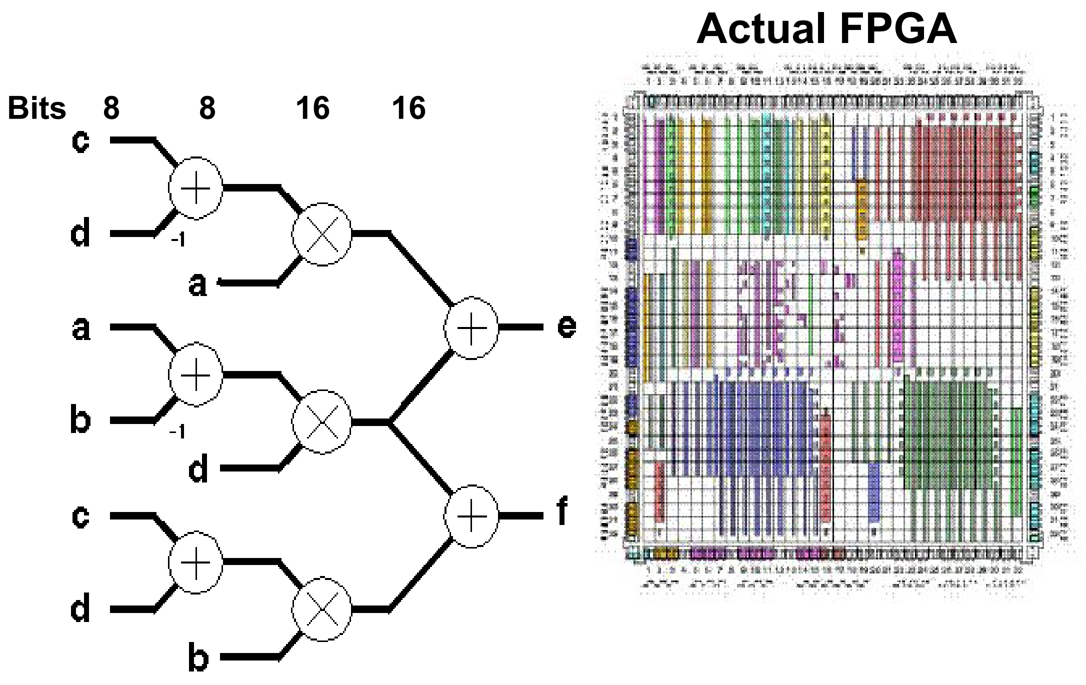


Figure 4. Complex Multiply

For the above example, we would like for both the HW cost and the rounding noise variance to be low. However, since you cannot have both, interesting combinations must be considered as shown in Figure 5 (cost vs. variance).

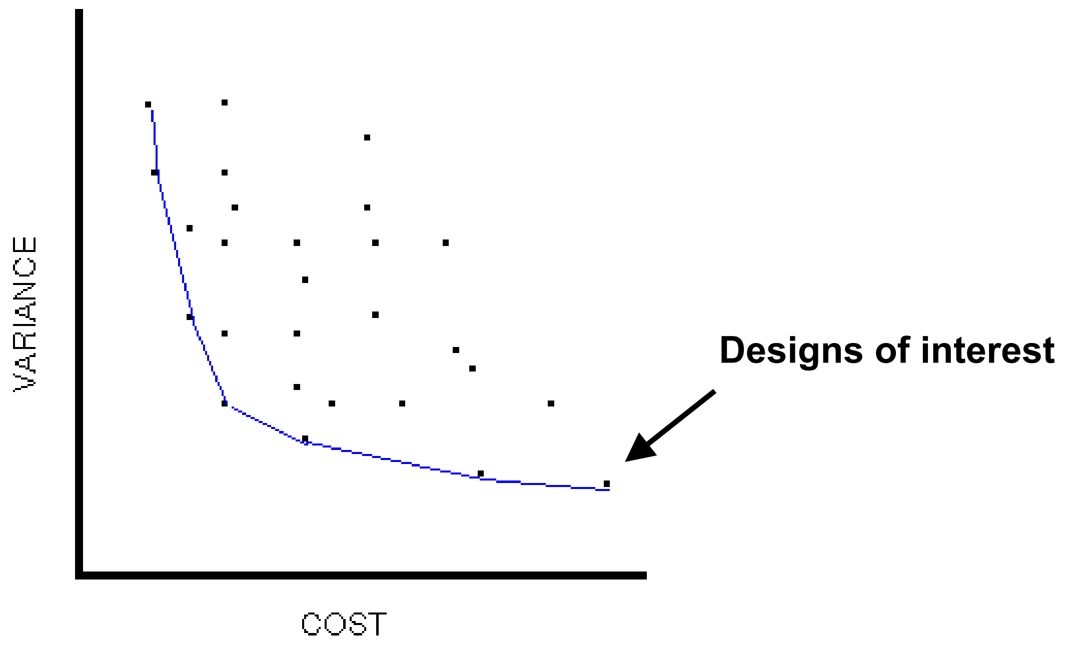


Figure 5. Cost vs. Variance for Complex Multiply Designs

Based on the above, cost and variance estimate functions had to be developed/employed. Figures 6 and 7 below show variance and cost estimates respectively.

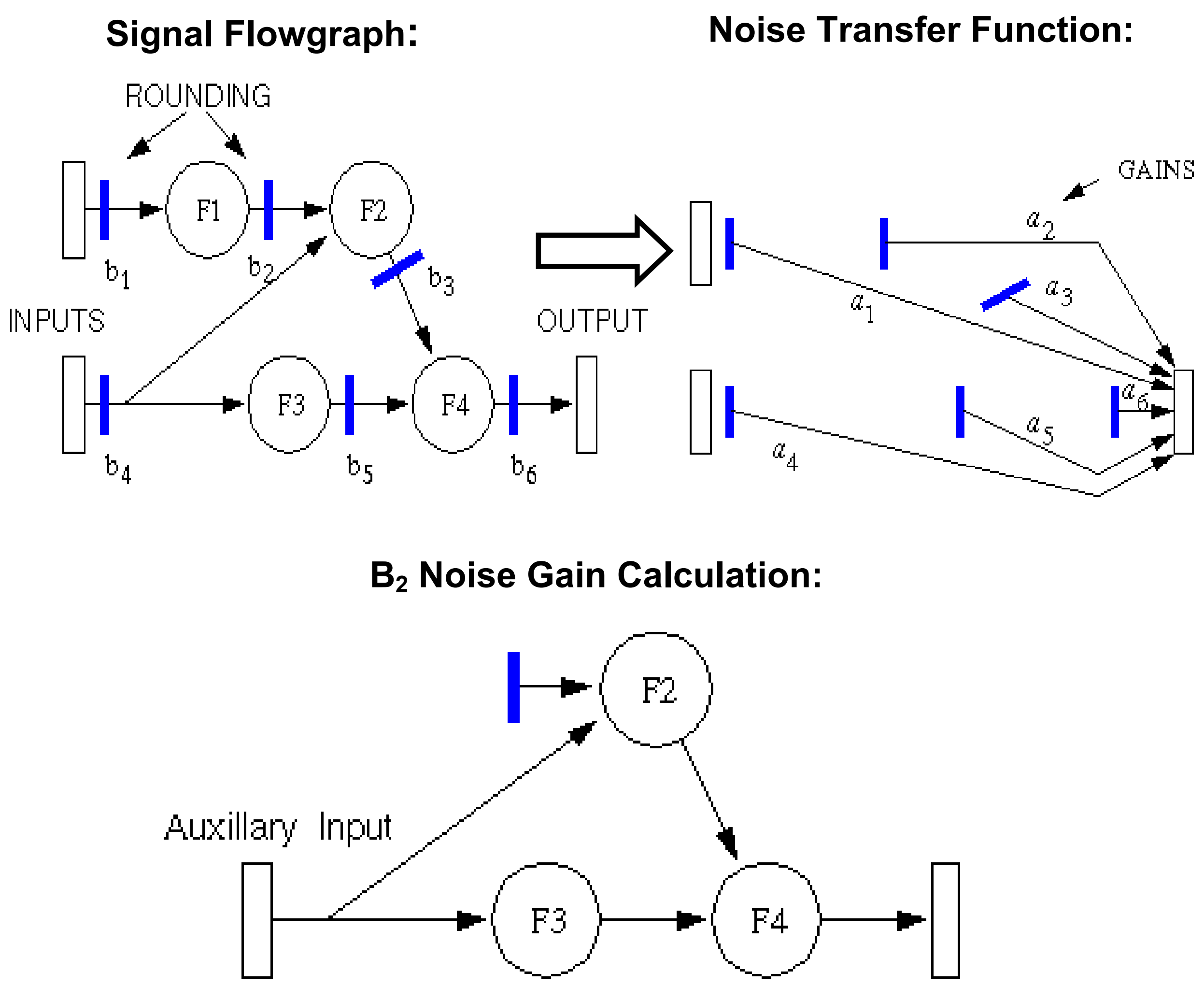
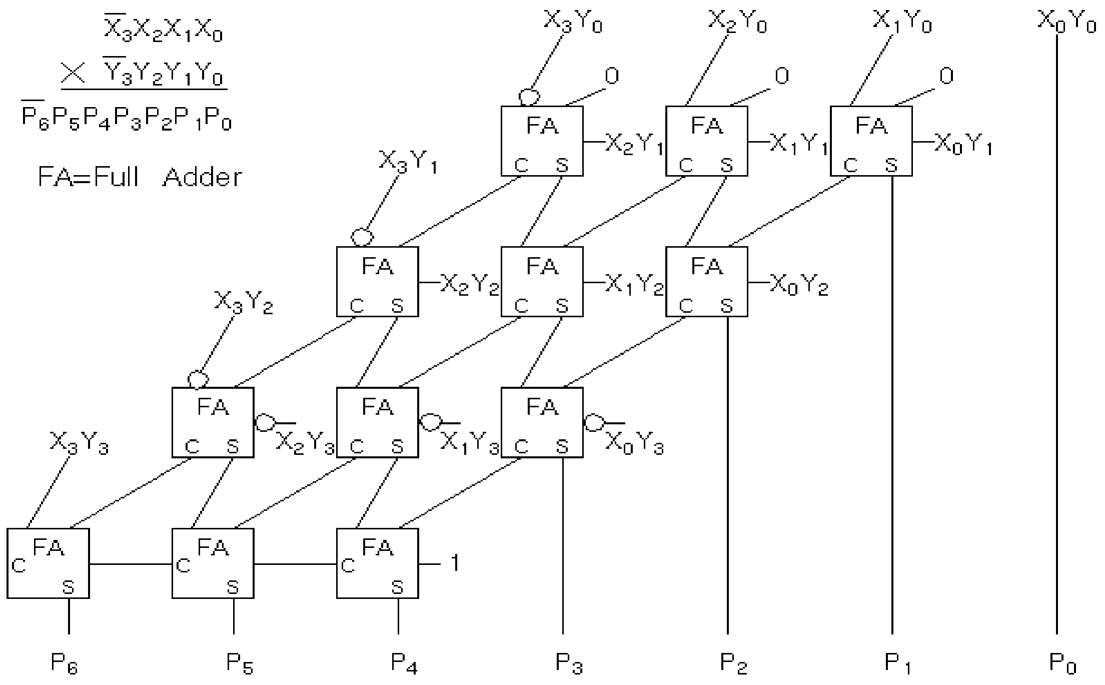


Figure 6. Variance Estimates

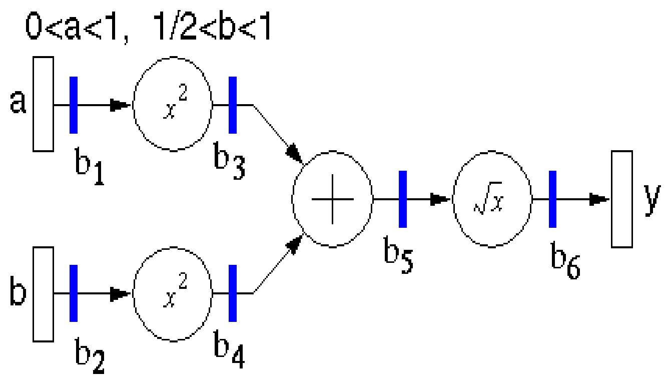
b_1x b_2 Multiplier Array Example



Cost = b_1x(b_2-1)

Figure 7. Cost Estimates

The following example of a magnitude calculation is an example that provided insight into our development of the tools for algorithm analysis. Figure 8 shows the data flow graph and the associated cost and variance calculations.



$$\text{Cost}=b_1^2+b_2^2+\max(b_3,b_4)+1+b_5\,b_6$$

$$\text{Variance}=2^{-2b1}/15+(2^{-2b2}+2^{-2b3}+2^{-2b4})/12$$
$$+(2^{-2b5}+2^{-2b6})/3$$

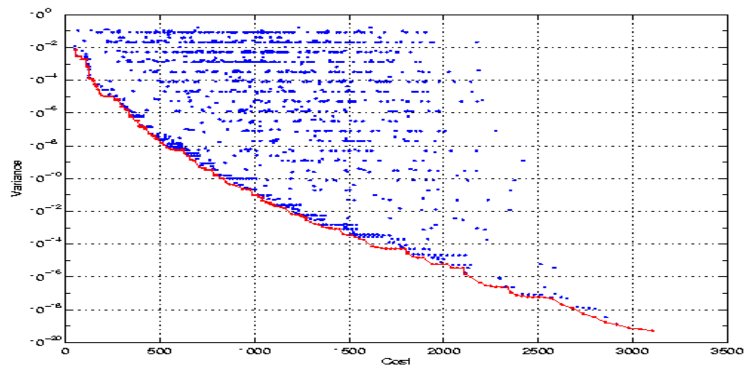
Constraints:

$$b_3 \le 2b_1$$
$$b_4 \le 2b_2$$
$$b_5 \le \max(b_3,\ b_4)+1$$
$$1 \le b_i \le 32$$

Figure 8. Cost and Variance Estimates for a Magnitude Calculation

Figure 9 shows the results if a “naïve” monte carlo calculation were used to explore the cost vs. variance of different designs within the design space and the resultant Monte Carlo wordlengths.

Results of Naïve Monte Carlo Algorithm



Monte Carlo Wordlengths

b1	B2	B3	B4	B5	B6
4	2	4	2	4	6
5	7	10	10	7	9
11	13	19	17	12	12
15	15	25	30	18	17
18	18	30	19	19	17
20	20	22	23	19	19
24	23	27	32	28	23
27	25	26	32	28	26
28	27	28	27	26	29
31	26	31	31	31	28

Figure 9. Results of Naïve Monte Carlo Algorithm and Monte Carlo Wordlengths

The following charts of Figure 10 show a comparison of predicted versus actual performance.

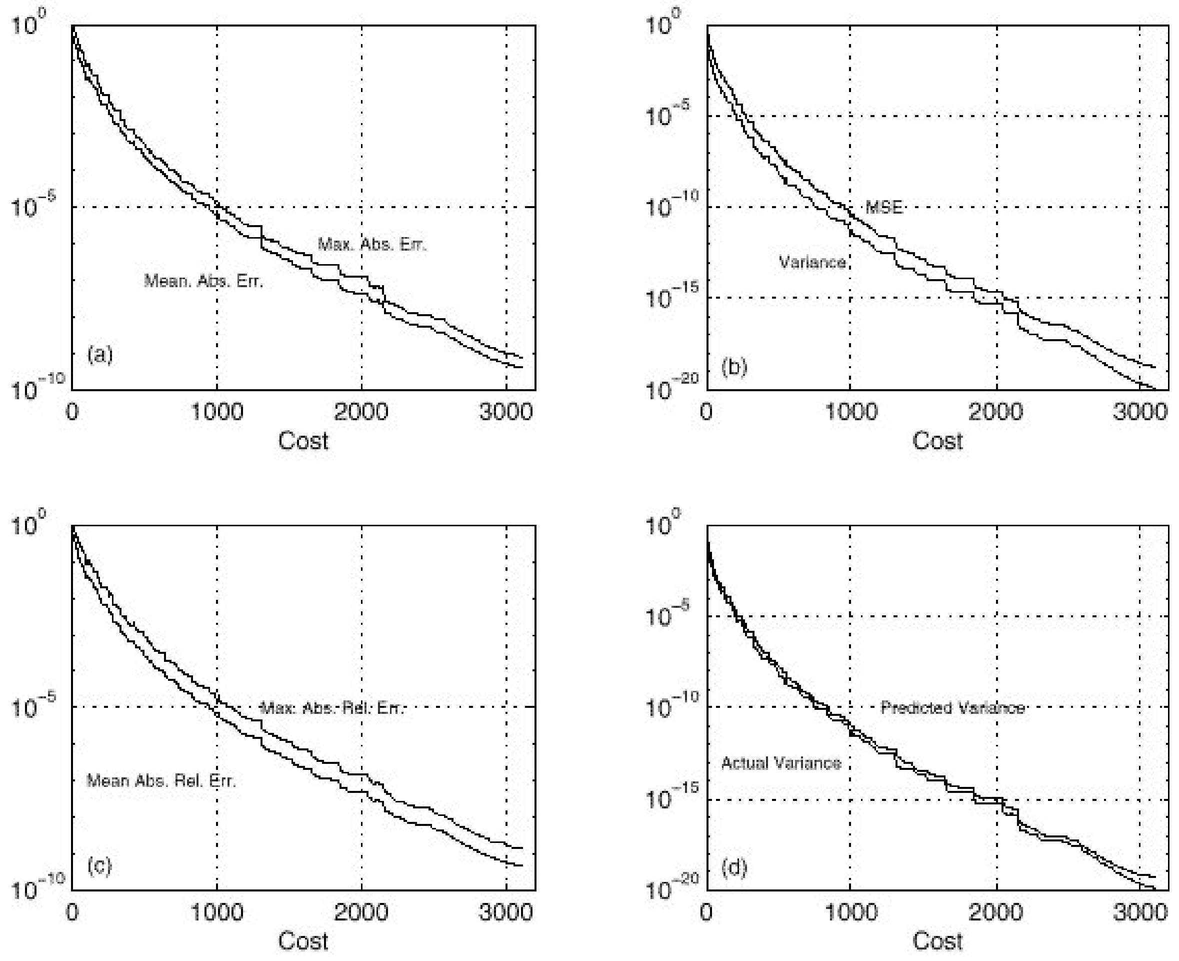


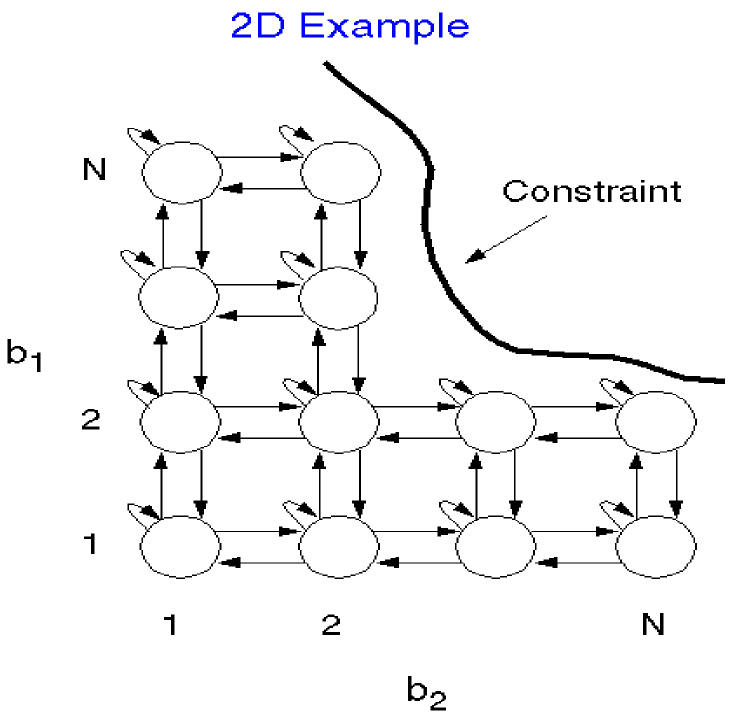
Figure 10. Comparison: Prediction vs. Actual Performance

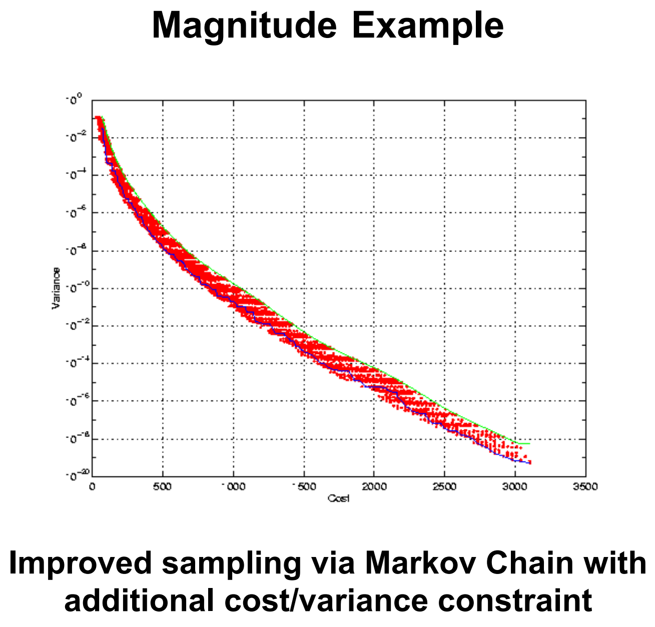
It could be seen that the Naïve sampling approach with the Monte Carlo algorithm wasted samples due to exploration of inefficient regions in the cost/variance plane. Also, when a violation of the constraints occurs, the sample is discarded. Since the design space is exponential in the number of design parameters, it is essentially infeasible to exhaustively test all combinations. This problem only gets worse in multiple dimensions which can include multiple outputs sharing HW building blocks. A better solution was needed.

The improved solution was the use of Markov Chains. A Markov Chain can be described as follows:

- Current state is a vector of wordlengths b
- Transitions involve +/- 1 one of the b elements
- Possible transitions are chosen randomly with equal probability
- If a transition would cross from feasible to infeasible, the current state is maintained

In this manner, a random walk is performed on the Markov Chain, testing b along the way. It can be shown that each feasible stae has equal probability of being sampled (after some number of steps). Figure 11 below shows an example of improved sampling using Markov Chain sampling.





$$steps = \frac{\log \delta / N}{\log(1 - \sqrt{10}N^{-2})}\left(n + \frac{1}{2}\right)$$

Figure 11. Markov Chain Sampling

An additional modified approach that we took provided an iterative approach to the Markov Chain sampling. This was done to overcome the fact that the cost/variance constraints are not usually known. In this fashion, each iteration uses only a small number of samples. The first time through, no constraints are applied. After each iteration, a curve is passed through the best lower bound. As the curve moves up, the curve becomes the next constraint. To determine the lower bound for optimization, and referring to Figure 12 below, the following steps are taken:

- Given C(b) and V(b) (cost and variance as a function of wordlength b)
- Start with a=0
- Minimize C(b) + aV(b) through choice of b
- Slightly increase a, and repeat

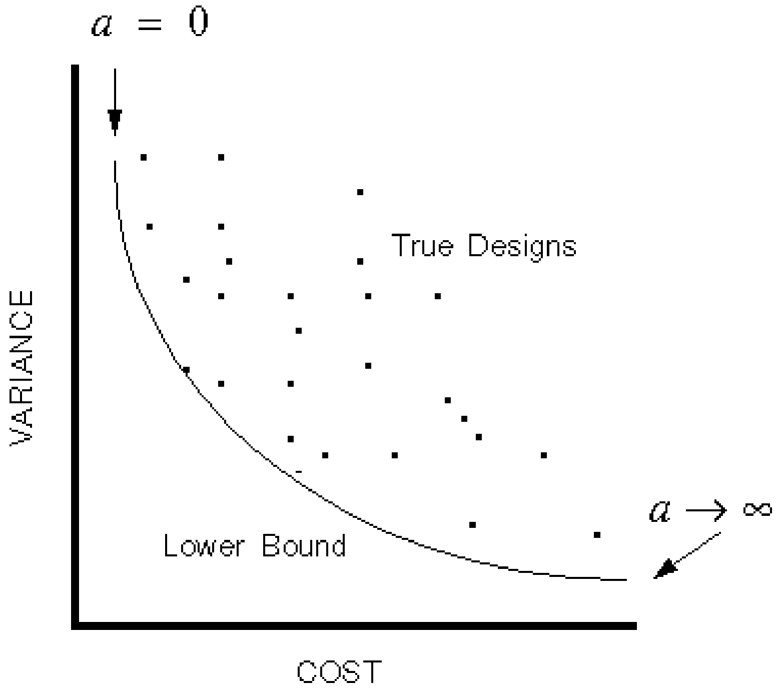
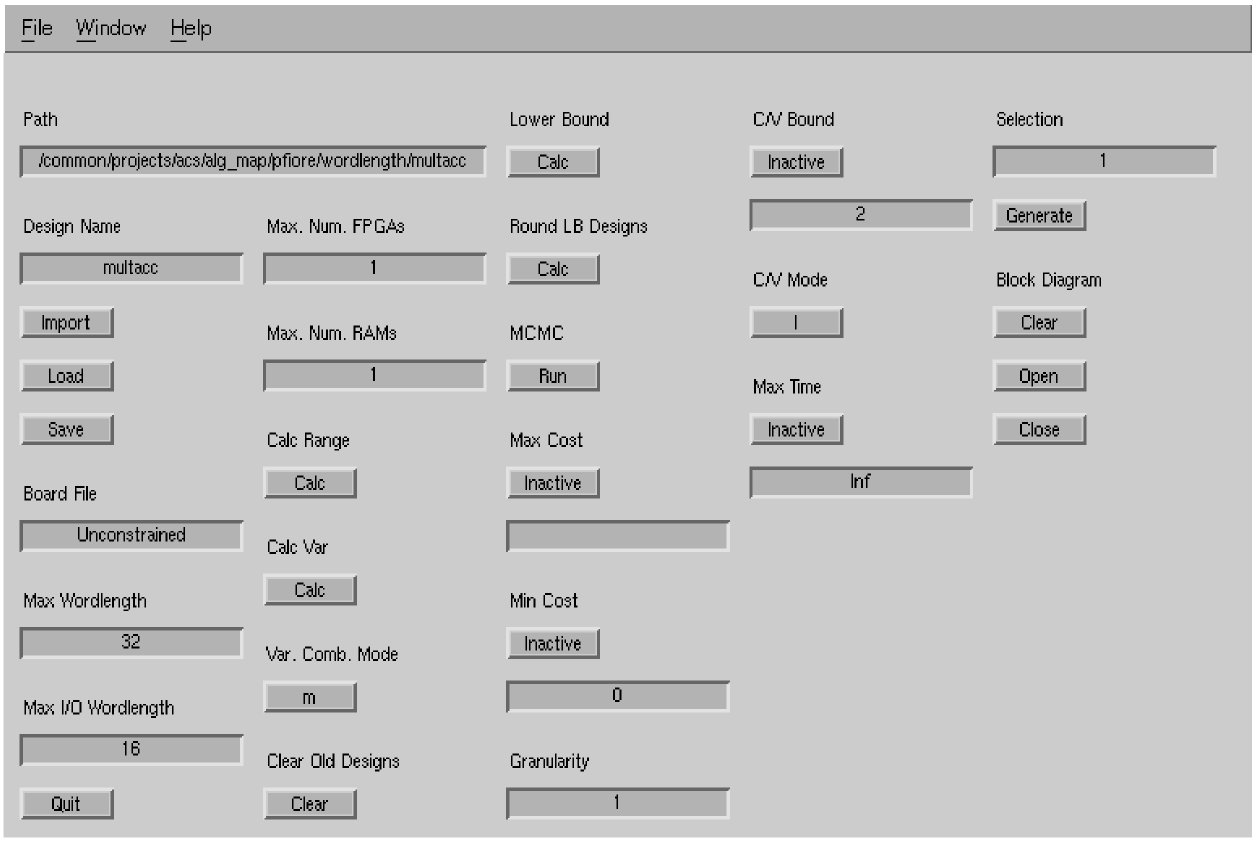


Figure 12. Lower Bound for Optimization

The code uses several optimizations to enhance speed. This procedure can be used for initializing the Monte Carlo method.

The resultant user interface for the analysis tools was developed in Matlab and is shown in Figure 13. As stated earlier, this user interface evolved into a tool deeply integrated in both the algorithm analysis and algorithm mapping phases. Specific details of the tools uses is described in the procedure at the end of this section.



Path: /common/projects/acs/alg_map/pfiore/wordlength/multacc
Design Name: multacc
Board File: Unconstrained
Max Wordlength: 32
Max I/O Wordlength: 16
Max. Num. FPGAs: 1
Max. Num. RAMs: 1
Calc Range: Calc
Calc Var: Calc
Var. Comb. Mode: m
Clear Old Designs: Clear
Lower Bound: Calc
Round LB Designs: Calc
MCMC: Run
Max Cost: Inactive
Min Cost: Inactive 0
Granularity: 1
C/V Bound: Inactive 2
C/V Mode: I
Max Time: Inactive Inf
Selection: 1
Block Diagram: Clear Open Close

Figure 13. ACS Tools User Interface

Based on the development stages of the algorithm analysis capabilities of the tools, we determined the following general conclusions regarding algorithm analysis:

- There is a great demand for custom DSP implementations
- New devices match well with DSP requirements
- Optimizing DSP designs can be difficult
- Random sampling methods fail for large problems
- The Markov Chain for uniform sampling solves this problem
- Lower bounding calculation is useful and easy to calculate

4.1.2 Algorithm Mapping

The objectives of algorithm mapping were three-fold:

- Performance Modeling – provide feedback on utilization, throughput, eficiency, etc… The feedback should be used by the algorithm analysis capabilities.

- Partitioning and Mapping – break large data flow graphs into groups and map those groups across multiple devices and across time.

- Automatic Scheduling - automatically determine the firing sequence, optimal mappings and sequence of configurations.

As a result of the development efforts related to algorithm mapping, the tools capabilities matured to include cost analysis as part of the wordlength analysis, the capability for automated uni-rate and multi-rate pipeline alignment and scheduling through developed algorithms, and support for memory allocation.

For algorithm mapping, it is appropriate to first describe criteria related to the target architecture associated with our tools environment (see Figure 14 below). The number of reprogrammable devices was targeted initially for a single FPGA device. The final target supported evolved to multiple devices on a board. Device interconnections were defined as fixed interconnect with predicable delays. The I/O interfaces included shared memory attached to the reconfigurable devices. The dataflow model used was one of Synchronous dataflow, supporting first uni-rate and then multi-rate dataflow. Modes of operation included uni-rate, pipelined, synchronous operations or multi-rate, pipelined, synchronous operations. Two target architectures were used on this program. These included the Annapolis Micro Systems Wildforce^{TM} (Xilinx 4000 series) and Wildstar^{TM} (Xilinx Virtex^{TM}). The primary support provided under this program was for the Wildforce^{TM} .

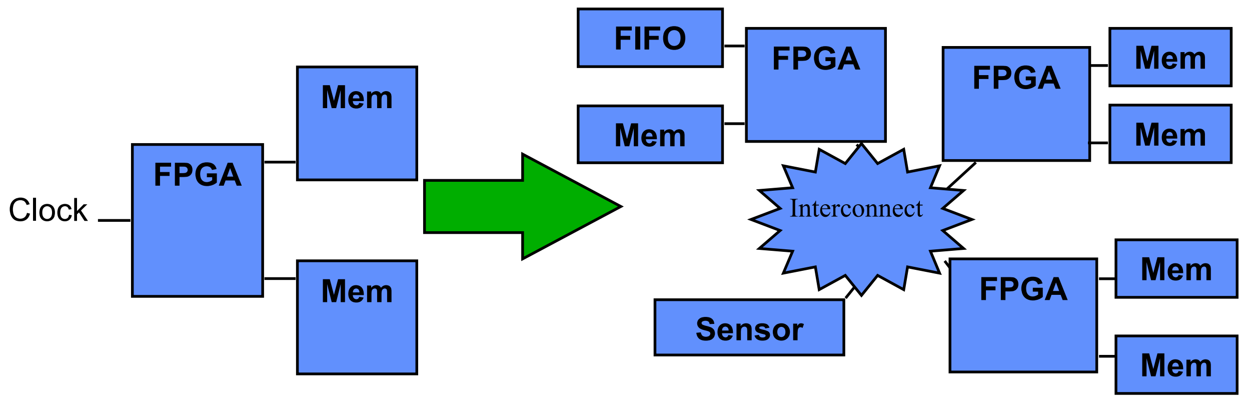


Figure 14. Target Architectures

The processing model used was well-matched to the Ptolemy Synchronous Dataflow (SDF) Domain in the following manner:

- The use of unit and block token produce and consume amounts
- The Netlist structure determines the execution order constraints
- Pipeline delay information is required to determine absolute timing
- Delays are set to align pipelines for the maximum throughput
- Delay can be automatically determined from block parameters

The processing model is a combination of a fully synchronous model and tagged synchronous models in that no handshaking or tags are used but data is not always valid. Data validity is implicit in the timing of latch signals. In addition, Memory access fits the same model in that data from a common memory is demuxed into separate streams running at lower rates and data to the common memory is multiplexed to a single port.

Another characteristic of the processing model is that multiple FPGAs introduce additional pipeline delays. The last feature of the processing model which was successfully developed and integrated into the tools environment was support for multi-rate parameterized execution, allowing for the significant achievement of supporting multiple data streams through the same FPGA device at different rates.

To begin the process of algorithm mapping, a method of creating the appropriate netlist had to be developed. A Macro Level Netlist Generator was developed that translates the algorithm blocks into a high-level netlist. This netlist is different from HDL approaches because the netlist at this level is more abstract than a wire-accurate low-level netlist. In this mode, where the hardware memory is a variable and the correct design is not already in a library, the Macro Level Netlist Generator compiles information about the hardware design.

With a macro netlist generated, additional functionality and coordination is then needed to actually produce a low-level netlist and hardware design. To assist in these functions a function called the Stitcher was developed. The Stitcher uses the data produced by the Macro Level Netlist Generator to coordinate the production of an actual low-level netlist and hardware design. This is done by properly accessing information produced by or contained in the Smart Generators (Smart Generators to be discussed in the next section), information regarding the optimal design choice determined through the use of the wordlength analysis tool and scheduling information, and then augmenting the netlist with this information, commanding the Smart Generators to actually create their associated hardware, and then linking all the hardware designs together.

Scheduling is performed by uni-rate and multi-rate sequencers developed on this project. Scheduling of the algorithm onto the FPGA hardware involved pipeline rearrangemnet and data port conflict resolution. To perform these functions, elements need to be added into the netlist. These include, for example, delays, multiplexers, address generators, tristates, and latches, each of which has its own Smart Generator. At this point, the detailed low-level design can be created.

The scheduling algorithms of the sequencers that were developed can be described through an example. Figure 15 shows a conceptual algorithm data flow graph with three inputs, two outputs and five operations or “instances”. The three input vector streams A,B and C all reside in one bank of RAM (also called a Port). The two output vectors D and E reside in a second port. All data vectors are assumed to be of the same size. All the instances in the design will be created with a pipeline delay of one, except 12, which requires a pipeline delay of two. Because of the unequal amount of pipeline delays, extra delays must be appropriately added to the netlist to correctly align the pipelines.

In addition to this pipeline alignment problem, the usage of the data ports must be correctly scheduled. For example, we cannot schedule D and E to be written at the same time. Our approach is to add multiplexers to the netlist for cases where multiple outputs go to the same port. We also add tristate buffers for cases both reads and writes to the same port can occur. Latches for input data can also be provided. These are not strictly necessary, but can provide for improved timing.

Figure 15 shows the results of this scheduling operation. Note the addition of the input latches, the internal delay, and the output multiplexer into the netlist by the sequencer. The Stitcher also connects all logic ports to the appropriate physical port. The timing chart in the figure shows the activation sequence of the internal nodes of the algorithm as well as the activity of the multiplexer control signal, latch load enable signal, and port activity. Several consecutive launches are shown in the chart. Note that the maximum launch rate of three cycles is attained.

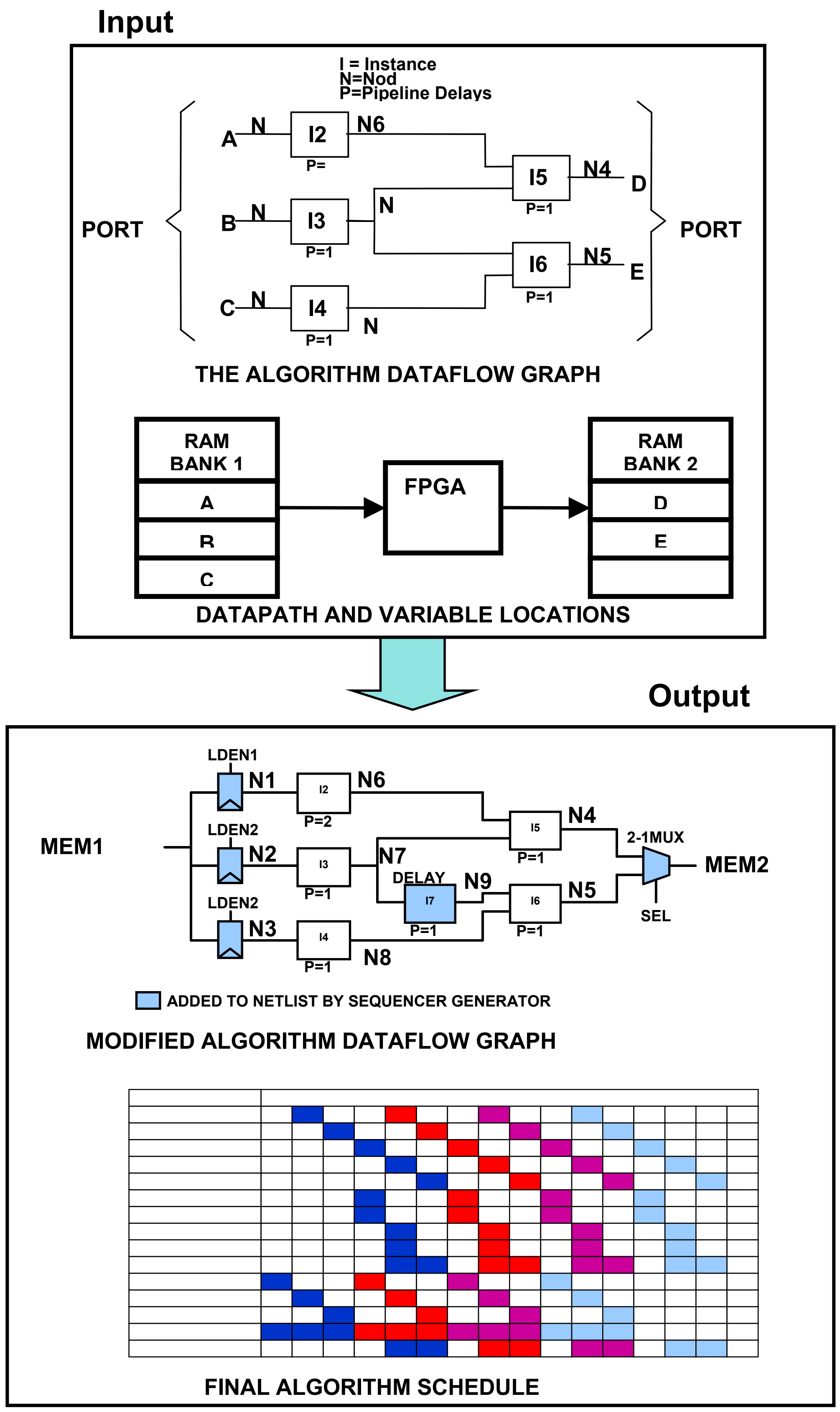
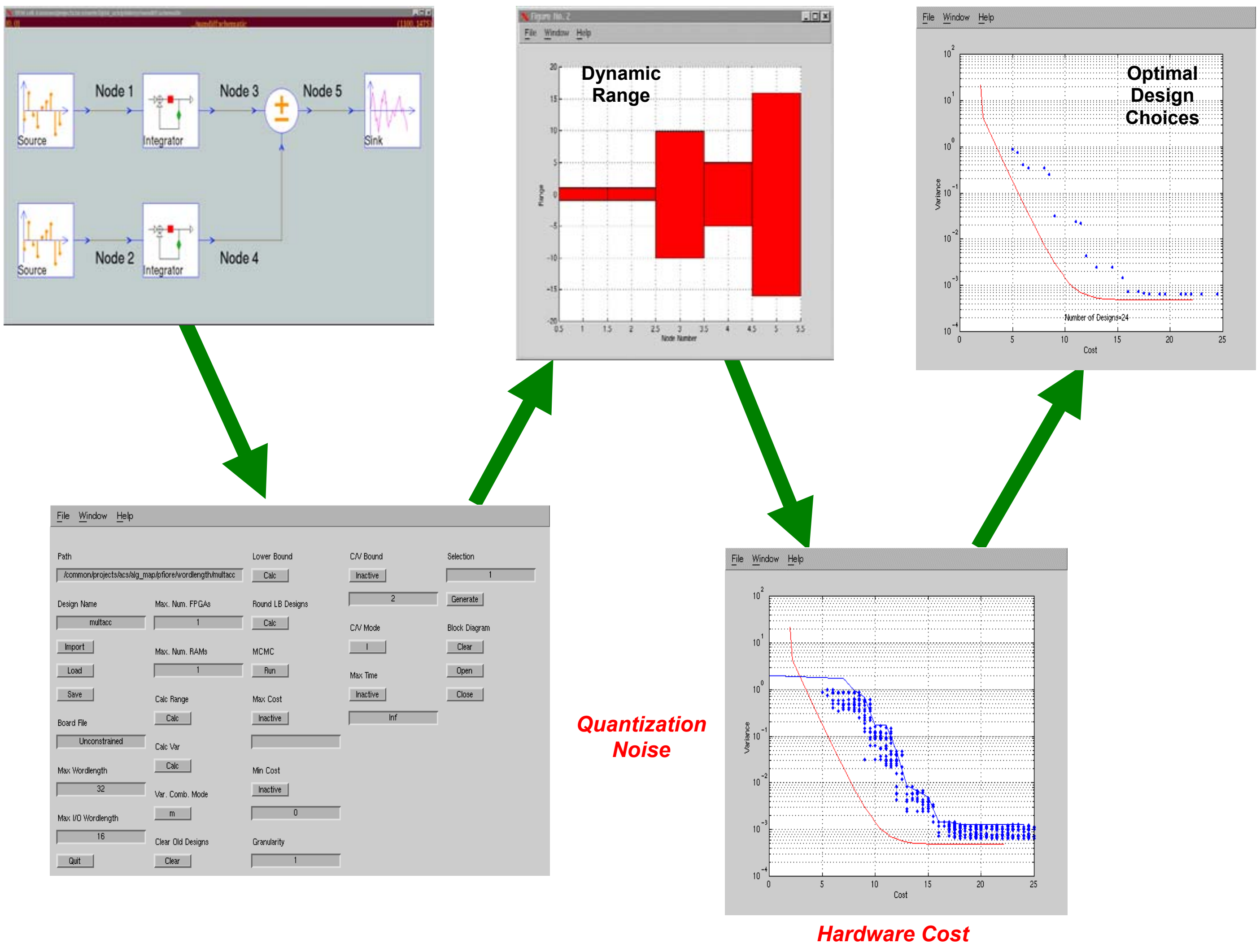


Figure 15. Pipeline Alignment and Schedule Determination Required for Logic Synthesis

In order to provide proper sequencing of data vectors, each bank of hardware memory requires an Address Generator. In our approach, the parameters needed by the Address Generators associated with a computation are automatically derived by the Macro Level Netlist Generator. This information is used by the stitcher to calculate needed address offsets. During the scheduling, the stitcher derives an Address Generator for each data port. This approach frees the algorithm designer from the task of writing code or developing hardware fro this purpose. Also, we found that generating custom address generators for each algorithm substantially reduces the required amount of logic from what would be required for general purpose address generators.

The following, along with Figure 16, depicts the final evolved state of the wordlength analysis tools and the interrelation and capabilities among the algorithm analysis and mapping design spaces:



Path: /common/projects/acs/alg_map/pfiore/wordlength/multacc	Lower Bound: Calc	C/V Bound: Inactive, 2	Selection: 1 (Generate)
Design Name: multacc	Max. Num. FPGAs: 1	Round LB Designs: Calc	C/V Mode: I
Import / Load / Save	Max. Num. RAMs: 1	MCMC: Run	Block Diagram: Clear / Open / Close
Board File: Unconstrained	Calc Range: Calc	Max Cost: Inactive	Max Time: Inactive, Inf
Max Wordlength: 32	Calc Var: Calc	Min Cost: Inactive, 0	
Max I/O Wordlength: 16	Var. Comb. Mode: m	Granularity: 1	
Quit	Clear Old Designs: Clear		

Figure 16. Wordlength Optimization Analysis

- The wordlength analysis tools starts by importing the macro level netlist and associated smart generator files from Ptolemy. These files are the rangecalc, natcon, numsim, cost, and schedgen files, whose functions will be described below.

- The wordlength tool next calculates the numerical range of each arc of the flowgraph. The rangecalc files from each hardware instance in the graph are used for this purpose. Essentially, the numerical range at the inputs of a particular instance is propagated to the outputs. The process is iterated until all numerical ranges are known. The manner in which a particular rangecalc file determines its output range is dependent on the mathematical function implemented by the instance.

- Once the ranges are calculated, we must now calculate the variances scale factors for every arc in the flowgraph. These determine the effect of rounding on the outputs of the graph. The various numsim files from each hardware instance in the graph are used for this purpose. These numsim files simply mimic the numerical properties of their corresponding mathematical function. They are used by exercising the flowgraph with random data, and then sligthly perturbing various graph points, so that an approximation to the gradient of the output of the graph can be obtained. This in turns determines the variance scale factors to be used in the subsequent optimization.

- With the variance scale factors now known, we are ready to find actual good designs, using the Markov Chain Monte Carlo (MCMC) method. The tool performs a random walk, keeping the best designs, as determined by the variance of a particular wordlength combination, as well as its cost. The costs are calculated by using the cost smart generator files.

- The random walk is modified by use of the natcon smart generator files. These files place constraints on allowable wordlength combinations. For example, many of the Coregen library elements nly allow certain input and output wordlength combinations; this knowledge is captured in the natcon files. If the random walk violates a natcon contraint, that particular design is discarded.

- The algorithm scheduling is also performed during the random walk. This is necessary because most hardware instances have schedules and latencies that change as a functions of the wordlengths. The individual schedgen smart generator files provide local information about the behavior of each instance. During the random walk, when a particular wordlength combination is proposed, the individual schedules are fitted together so that an efficient overall schedule results. Note that instances with differing data rates and latencies are easily handled by this approach.

- The user typically performs multiple random walks; the tool remembers the best designs from all the walks. As the design exploration progress, the user has the ability to control the regions of the design space that the tools visits. This is done by setting bounds on the minimum or maximum cost that will be tolerated, the maximum total latency of the design, and most importantly, an upper bound on cost/variance combinations to allow in the random walks.

- This last constraint greatly improves the efficiency of the random walks. As the set of optimal designs improves (approaches the theoretical lower bound), the tool can place a reasonable upper bound on cost/variance combinations to consider. When the random walk would violate this constraint, the step is simply not taken. It can be shown that the resulting feasible region will be uniformly covered by a random walk (in the limit). By gradually moving the upper bound closer to the set of optimal designs so far discovered, we shrink the feasible region so that it approaches only the truly optimal set of feasible designs.

- Once the user has a set of designs that are good enough, the tool allows the user to check the schedule of various designs, and to then select one to actually build. The selected design is translated into a macro level netlist, with the actual wordlengths and schedule information. This data is then used by the sticher to construct the VHDL for the design.

4.2 Smart Generators

Smart generators provide a systematic way to capture design information about specialized arithmetic structures. Smart generators allow for rapid population of libraries with new components by capturing algorithm-specific implementations, from the bit-level up to the coarse grain function level, and by making information about the implementation available for the hardware design and implementation process and tools. An algorithm designer specifying a mathematical operation may not be aware of many of the ingenious implementations that have been developed. Even simple structures such as adders, multipliers, and filters have an enormous array of implementation alternatives, many of which have been finely tuned by computer architects and VLSI designers. A Smart Generator can choose the correct implementation considering the user's specification and the target technology. This approach is different and more powerful than a simple parameterized building block.

A novel feature of the Smart Generator is the ability to utilize other Smart Generators in constructing its hardware implementation. Very useful higher-order functions can be constructed in this manner using hierarchical and normal Smart Generators, with a minimum of effort. For example, filters, FFTs, and linear algebraic operations such as matrix multiplication can be constructed out of simpler building blocks such as additions, subtractions, multiplications, and divisions. A Smart Generator may actually choose among multiple implementation strategies that are known to the designer, and may even call itself with different design parameters.

To actually write a Smart Generator, we had to invoke basic element building blocks which were appropriate to the FPGA technology we used. The basic building blocks for the Xilinx 4000 series FPGA logic family, the primary FPGA technology used on our program, is described as:

- AND, NAND, OR, NOR – Simple combinatorial logic

- NOT, XOR, XNOR - Simple combinatorial logic
- 16ROM – Generic 4-input 1-output combinatorial logic element
- FD – Positive edge triggered flip-flop
- FDCE – Positive edge triggered flip-flop with clock enable and clear
- VCC,GND – Used to tie off an input to a fixed high or low voltage
- FA – Full adder
- FA2 – Two bits of a fast ripple carry adder string
- CONNECT – Connect two signals together

The objectives associated with the development and use of Smart Generators included:

- Parameterized libraries – generate node implementations for specified bit widths and parameter values

- Hierarchical representations – provide generators that can recursively call other generators

- Interface generation – automatically generate software to move data between general-purpose processor and the reconfigurable platform and to manage sequences of configurations

- General synthesis – provide device independent representation of implementation

The above objectives were accomplished. We were successful in implementing a portable logic synthesis methodology for VHDL. We integrated Xilinx Core Generators (4000 Series) capability within the VHDL code generation. In addition, we implemented smart generators for state machine and memory control requirements. Lastly, Smart Generators that can call other Smart Generators (hierarchical generation) were developed and implemented.

During development of the multirate sequencer for scheduling, it was discovered that for multirate operation, each input and each output of each instance Smart Generator must have a corresponding "enable" control point. This represented a change from the unirate approach, where one enable control for the entire star (function) was sufficient. It is acceptable for the enable control to be internally unconnected, if the Smart Generator does not depend on its operation. In this case, the synthesizer will prune any control logic associated with this unconnected input. If the enable is connected internally, the understanding is that any storage elements driven by that enable signal only change state when the enable signal is active. This mechanism is necessary so that state can be saved between initializations.

4.3 Ptolemy Integration and the ACS Domain

Other than the Matlab specific wordlength analysis tools, the new capabilities developed under this program were developed and integrated with the Ptolemy signal processing algorithm and implementation environment from the University of California (UC), Berkeley (Figure 17). Ptolemy supports heterogeneous system simulation and design using several different models of computation, each implemented in a separate domain. The class of application problems addressed by our work falls into Ptolemy's synchronous dataflow (SDF) domain, where the flow of data is predictable and does not change. In addition to its well-defined models of computation, Ptolemy provides an open software architectue with standard interfaces among its functional areas. Our choice of Ptolemy allowed both rapid insertions of our new capabilities as well as the ability for additional researchers to build their own extensions and libraries, an approach that has been successful in the past.

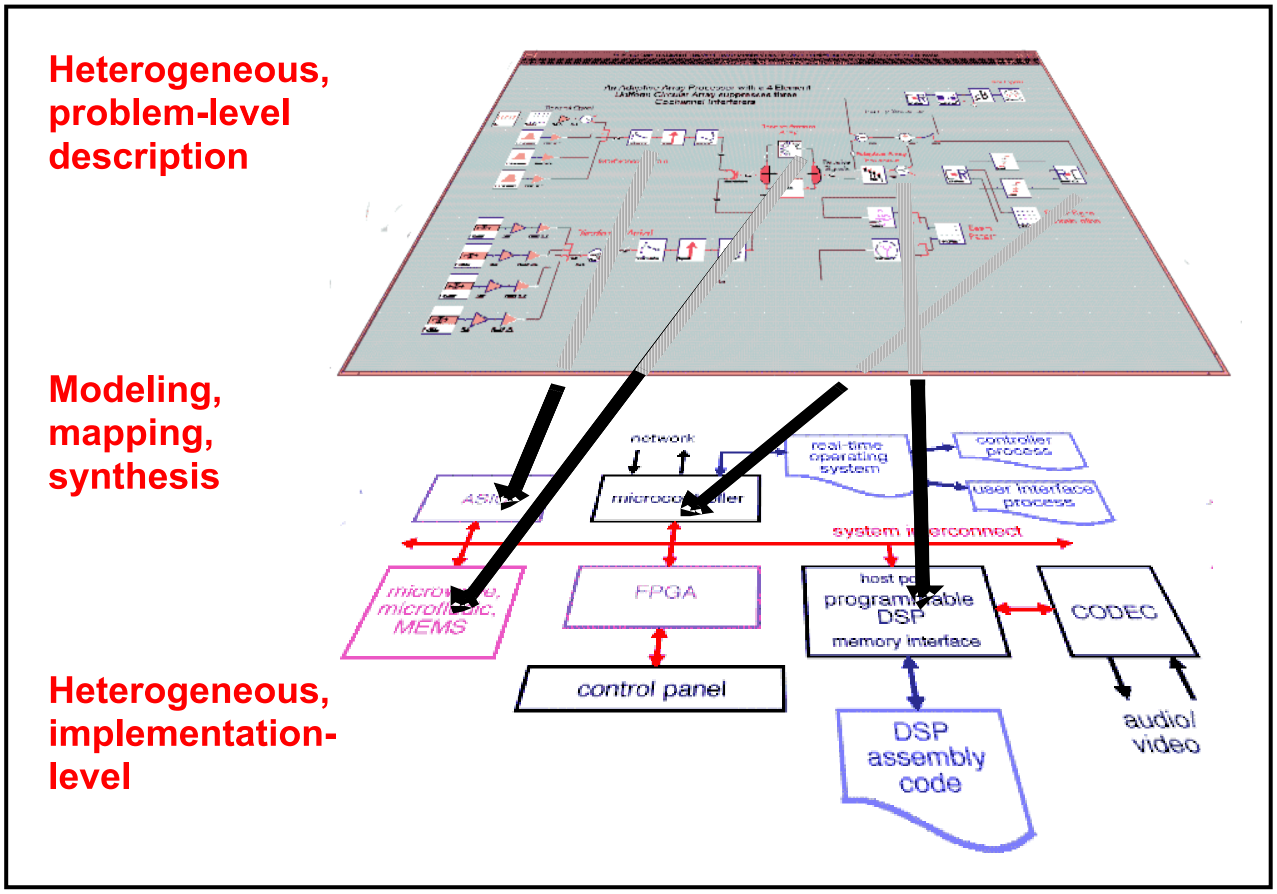
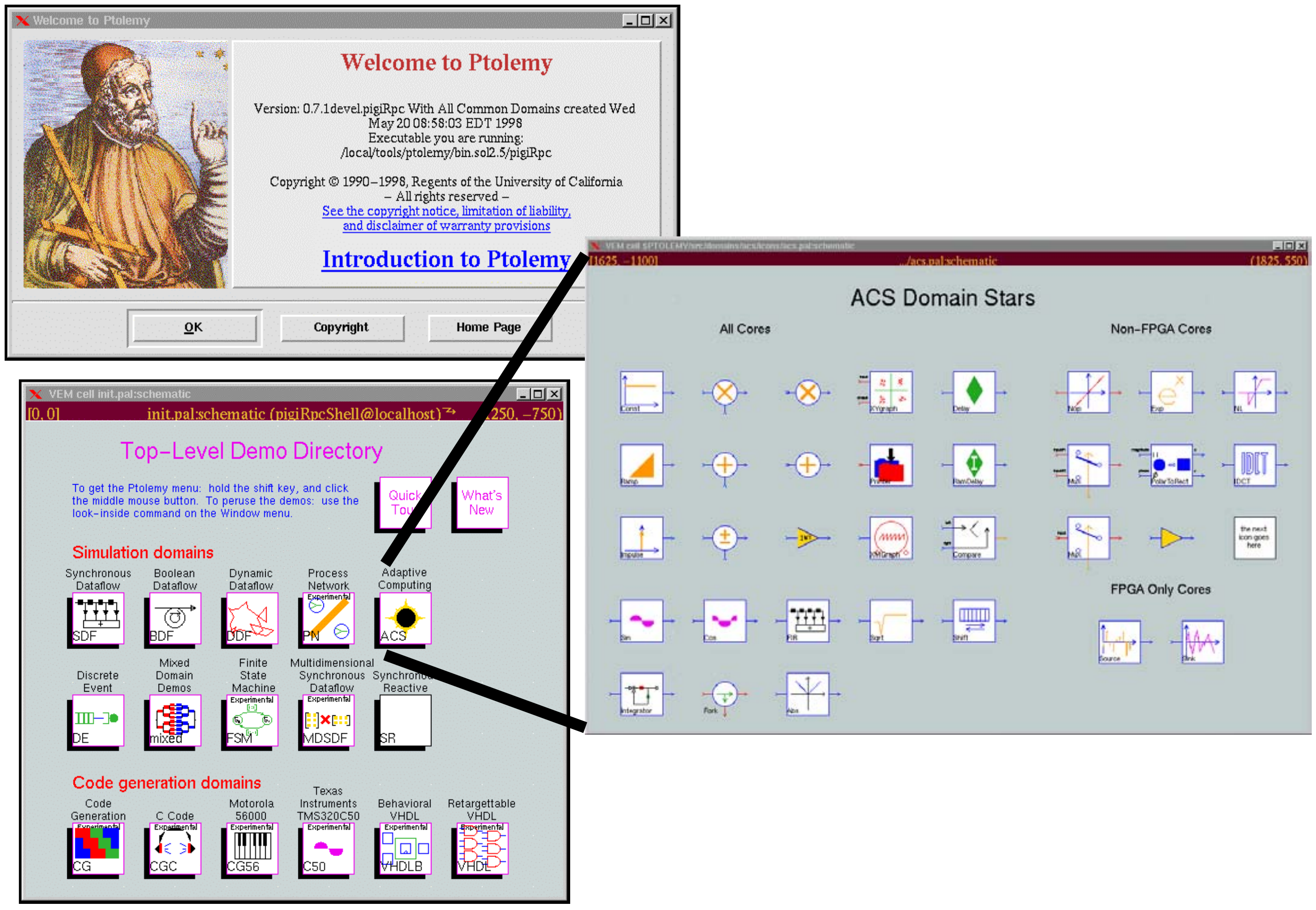


Figure 17. Ptolemy Project, University of California, Berkeley<http://ptolemy.eecs.berkeley.edu>

Within Ptolemy, there has traditionally been a distinction between simulation and implementation (code generation), and separate domains were created for each purpose. No implementations (code or hardware designs) could be generated from the simulation domains and simulation was not possible from the code generation domains. Also, it was not possible to move among different types of simulation (e.g. floating-point to fixed-point) or among code generation (e.g. C code, assembly code, hardware designs) without recreating graphs with different models.

Together with UC Berkeley, we extended Ptolemy by developing a new Adaptive Computing System (ACS) domain (Figure 18) that separates the interface specification from the implementation for each signal processing functional block, allowing simulation and design/code generation to exist in the same domain.


Welcome to Ptolemy

Welcome to Ptolemy

Version: 0.7.1devel.pigiRpc With All Common Domains created Wed May 20 08:58:03 EDT 1998
Executable you are running:
/local/tools/ptolemy/bin.sol2.5/pigiRpc

Copyright © 1990–1998, Regents of the University of California
– All rights reserved –
See the copyright notice, limitation of liability, and disclaimer of warranty provisions

Introduction to Ptolemy

VEM cell init.pal:schematic
[0, 0] init.pal:schematic (pigiRpcShell@localhost) (250, –750)

Top–Level Demo Directory

To get the Ptolemy menu: hold the shift key, and click the middle mouse button. To peruse the demos: use the look–inside command on the Window menu.

Figure 18. ACS Domain, Ptolemy Project

In the ACS Domain, algorithms are still represented by SDF graphs. We have used a "Corona/Core" architecture (Figure 19), where each block has a common interface known as the Corona, and one or more implementations, known as the Cores. A retargeting mechanism allows the user to change Cores and hence implementations. The Implementation of a data flowgraph between various simulation models (floating-point, fixed-point) and implementations (C code, VHDL code) is accomplished by simply selecting a different target.

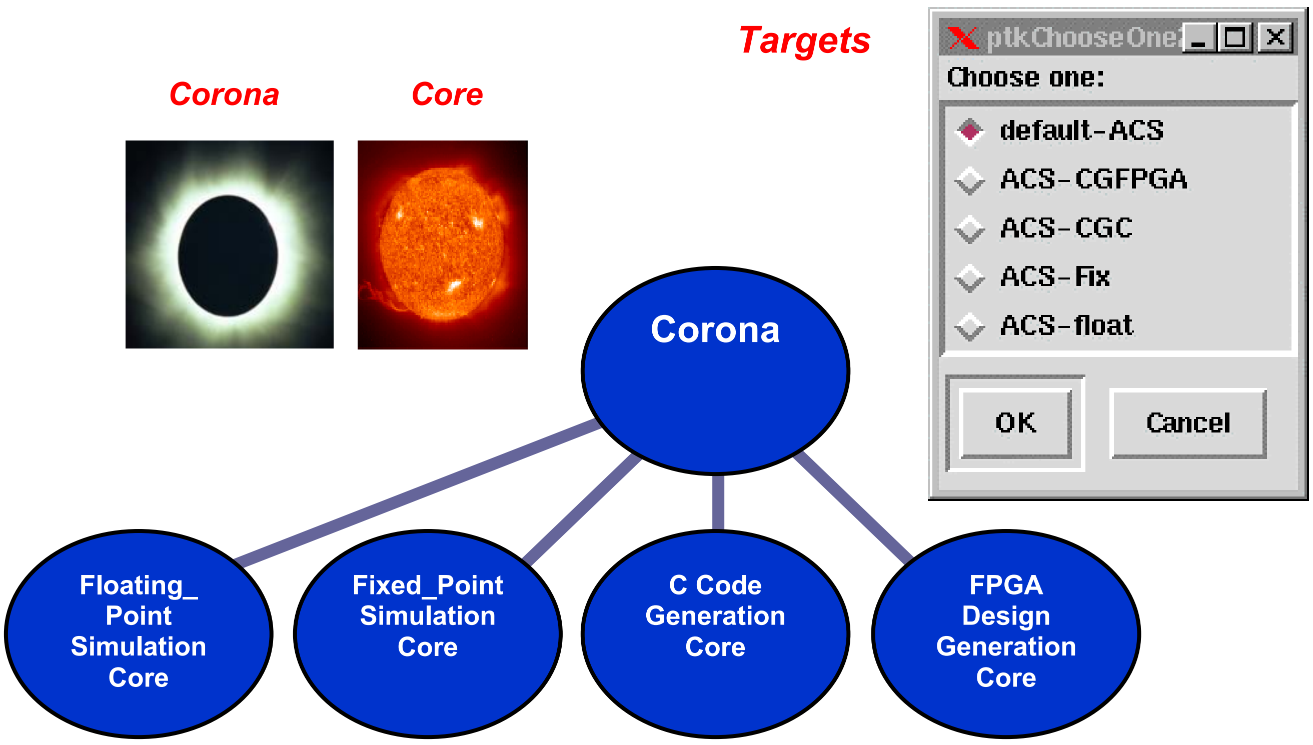


Figure 19. ACS Domain, Corona/Core and Targets

During January, 2000, in the process of integrating the multirate tool, we decided on several multirate blocks to add to the palette of Ptolemy ACS stars/core. These were as follows:

- "Commutator": This block sends one input to N outputs in a round robin fashion. Input and outputs same numerical format.

- "Mux": This multiplexer is actually a unirate block. Every input and every output is valid every clock cycle. One input is propagated to output, chosen by "select" input. Data formats on input and output must be identical. Select lines format must be internally locked. Note that a demultiplexer cannot be incorporated into the current system because the output activation times would be data dependent and therefore not deterministic (i.e. not synchronous).

- "Interleaver": This block is the reverse operation of the commutator, with N inputs and a single output. The output is valid every clock cycle, inputs selected in a round robin fashion. A starting phase field in the edit box will be implemented.

- "Downsampler": The input to this block is valid every clock cycle, and it produces a valid output every Nth clock cycle. Data formats on input and output must be identical. A starting phase field in the edit box will be implemented.

- "Repeater": This block has three parameters, upsample rate expansion (M), repeat amount (R), and a fill value (F). The input to this block is valid every M samples, with the output valid every sample. Every input gets replicated R times in a row.

- "Data Latch": This is a unirate block with all inputs and every outputs is valid every clock cycle. If latch input is asserted, the data input gets propagated to output. If latch input is not asserted, current data output is held. Data formats on input and output must be identical.

4.4 Demonstrations

The demonstration objectives for this program were to use the ACS tools to automatically implement several applications along the various stages of development of the tools capabilities, each with increasing challenge and scope. Four main demonstrations and several additional demonstrations exercising and proving the capabilities of the ACS tools during varying stages of the program were developed. Aside from the four main demonstrations described below, the additional demonstrations developed included comparison of fixed point vs floating point implementation/results for smaller algorithms (e.g. a “Butterfly” generation algorithm), the first demonstration of the tools automatic implementation of multi-FPGA use (for a magnitude calculation), and the development of an FPGA implementation of a high speed linear FM detector (for the Reconfigurable Algorithms for Adaptive Computing (RAAC) program).

In all cases above, the ACS tools were used to simulate the algorithm, select appropriate fixed-point representations, and generate the VHDL implementations. The final FPGA designs were obtained synthesizing the VHDL and performing place and route with commercial tools.

4.4.1 Demonstration: Design Generation for Single or Multiple FPGAs

The demonstration first showing implementation of multiple-FPGA use was first successfully generated by the ACS tools and executed on the Annapolis Micro Systems Wildforce^{TM} board during the first quarter 2000. To demonstrate this capability, a magnitude calculation algorithm (Figure 20) was developed under the Ptolemy interface environment. The dataflow graph consisted of two multipliers, an adder, and a square root block. This algorithm was manually partitioned across four FPGAs, with one multiplier in the first FPGA, and the other multiplier, adder and square root blocks were in placed in the third FPGA. Operand 1 was targeted for memory 1 (accessible to the first FPGA), operand 2 was targeted for memory 3 (accessible to third FPGA). The computed result was targeted for memory 4 (accessible to the fourth FPGA). The ACS tools inferred the use of the second FPGA as a means to connect the algorithm components in the first and third FPGAs. Since the result of third FPGA needed to be stored in the fourth FPGA's memory, this usage was thus inferred by our tools. The ACS tools thus constructed synthesizable VHDL for all four of the FPGAs which were then reduced to bitstream files for successful execution on the Annapolis Wildforce^{TM} board.

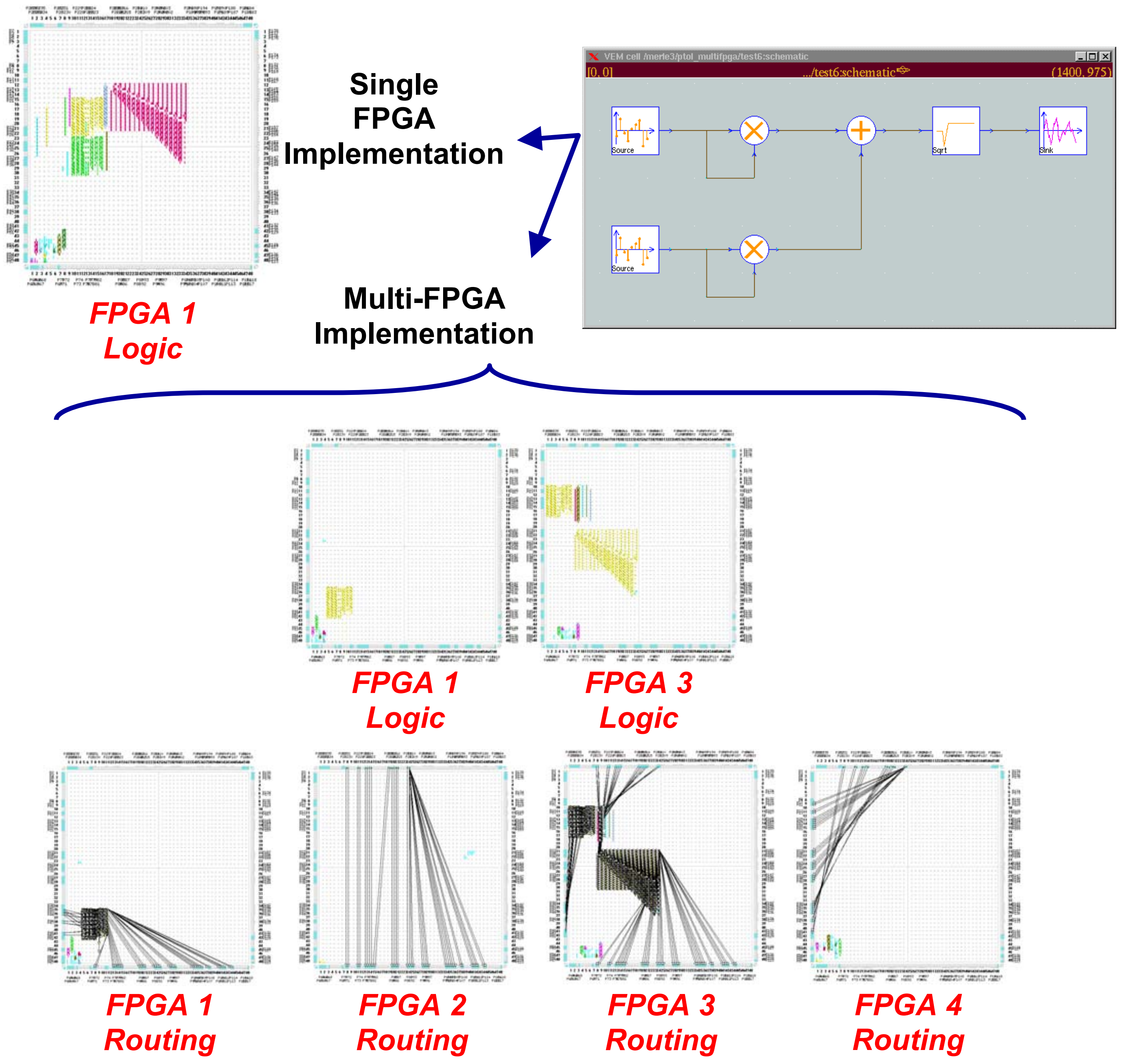


Figure 20. Design Generation for Single or Multiple FPGAs

4.4.2 Demonstration No. 1: Winograd DFT-Based FSK Communications Receiver

The first of the main demonstrations was a Winograd Discrete Fourier Transform (DFT) as part of a channelized frequency shift-keyed (FSK) receiver in ACS. The Winograd algorithm structure has the minimum number of multiplications for any DFT approach, and was thus ideal for FPGA implementation. This demonstration, shown in Figure 21, was shown at the April 1999 DARPA ACS PI meeting.

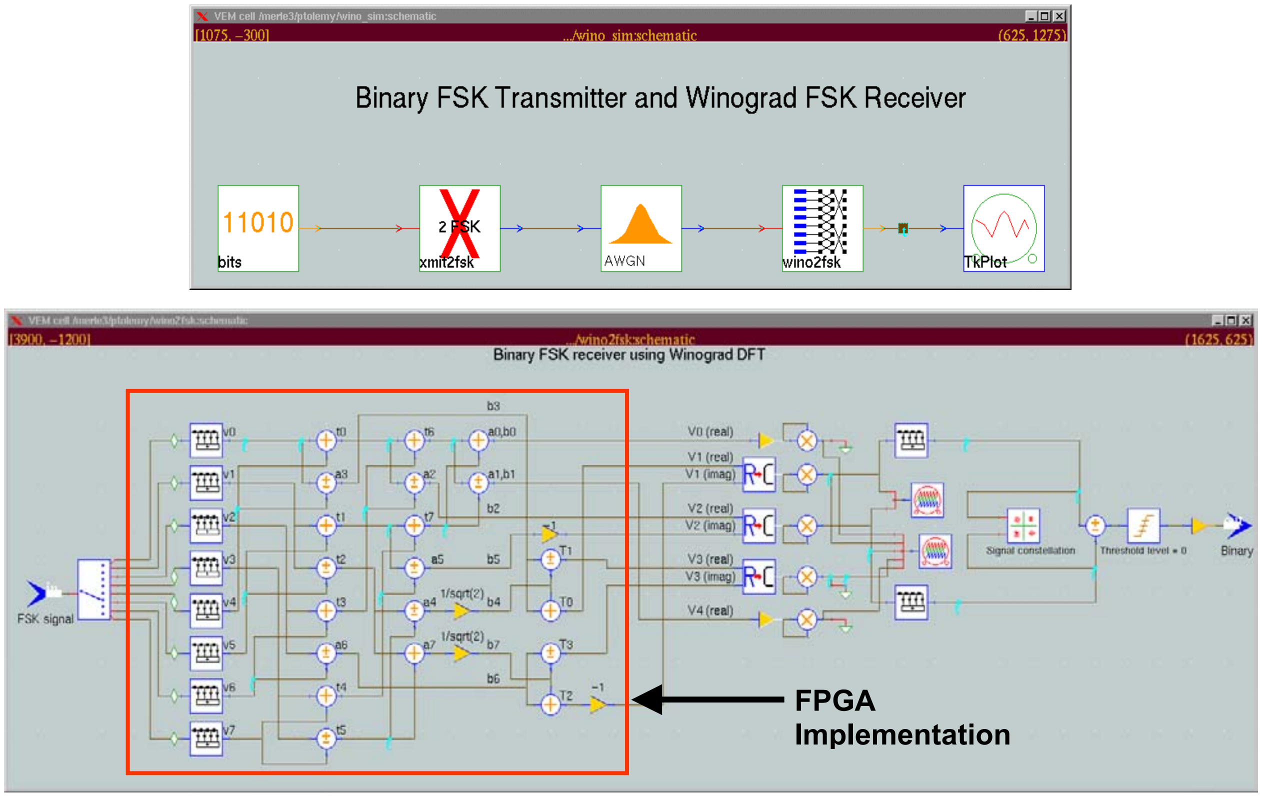
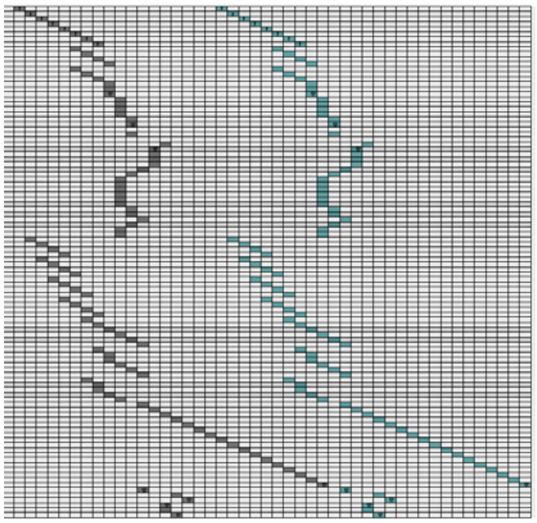


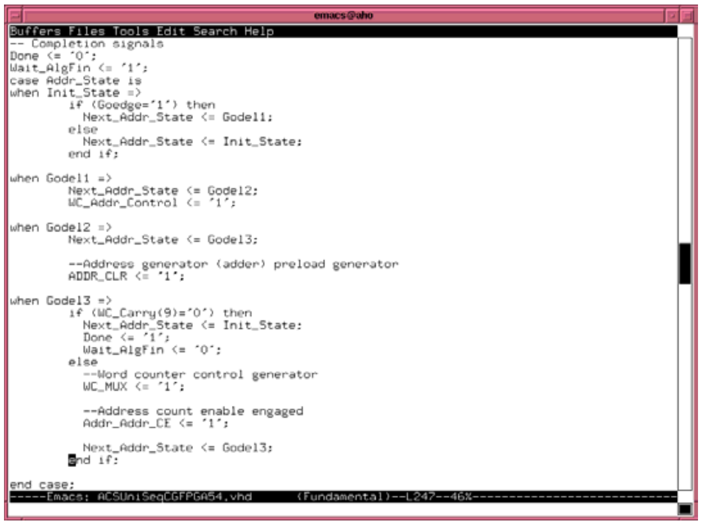
Figure 21. Winograd DFT-Based FSK Communications Receiver

With the Ptolemy data flow graph above, the ACS tools and environment for were used to generate the appropriate HW design, generate a viable optimized schedule and synthesizable VHDL. The back-end tools were then used for place and route of the design. These resultant VHDL code, schedule and FPGA design (placed and routed) are shown in Figure 22.

Generated Schedule



Generated VHDL



```
-- Completion signals
Done <= '0';
Wait_AlgFin <= '1';
case Addr_State is
when Init_State =>
        if (Goedge='1') then
          Next_Addr_State <= Godel1;
        else
          Next_Addr_State <= Init_State;
        end if;

when Godel1 =>
        Next_Addr_State <= Godel2;
        WC_Addr_Control <= '1';

when Godel2 =>
        Next_Addr_State <= Godel3;

        --Address generator (adder) preload generator
        ADDR_CLR <= '1';

when Godel3 =>
        if (WC_Carry(9)='0') then
          Next_Addr_State <= Init_State;
          Done <= '1';
          Wait_AlgFin <= '0';
        else
          --Word counter control generator
          WC_MUX <= '1';

          --Address count enable engaged
          Addr_Addr_CE <= '1';

          Next_Addr_State <= Godel3;
        end if;

end case;
-----Emacs: ACSUniSeqCGFPGA54.vhd      (Fundamental)--L247--46%---------------------------
```

FPGA Design

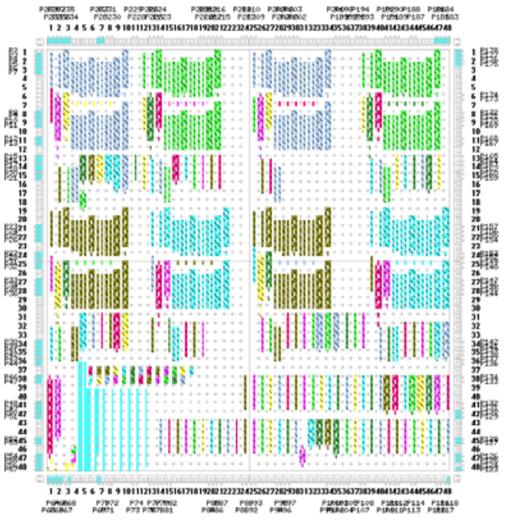


Figure 22. Winograd DFT-Based FSK Communications Receiver – VHDL, Schedule and HW Design

Lastly, Figure 23 shows the processing results of the receiver design. As can be seen, the resultant HW design successfully detected and separated the two signals of interest and successfully provided results capable of bit decoding.

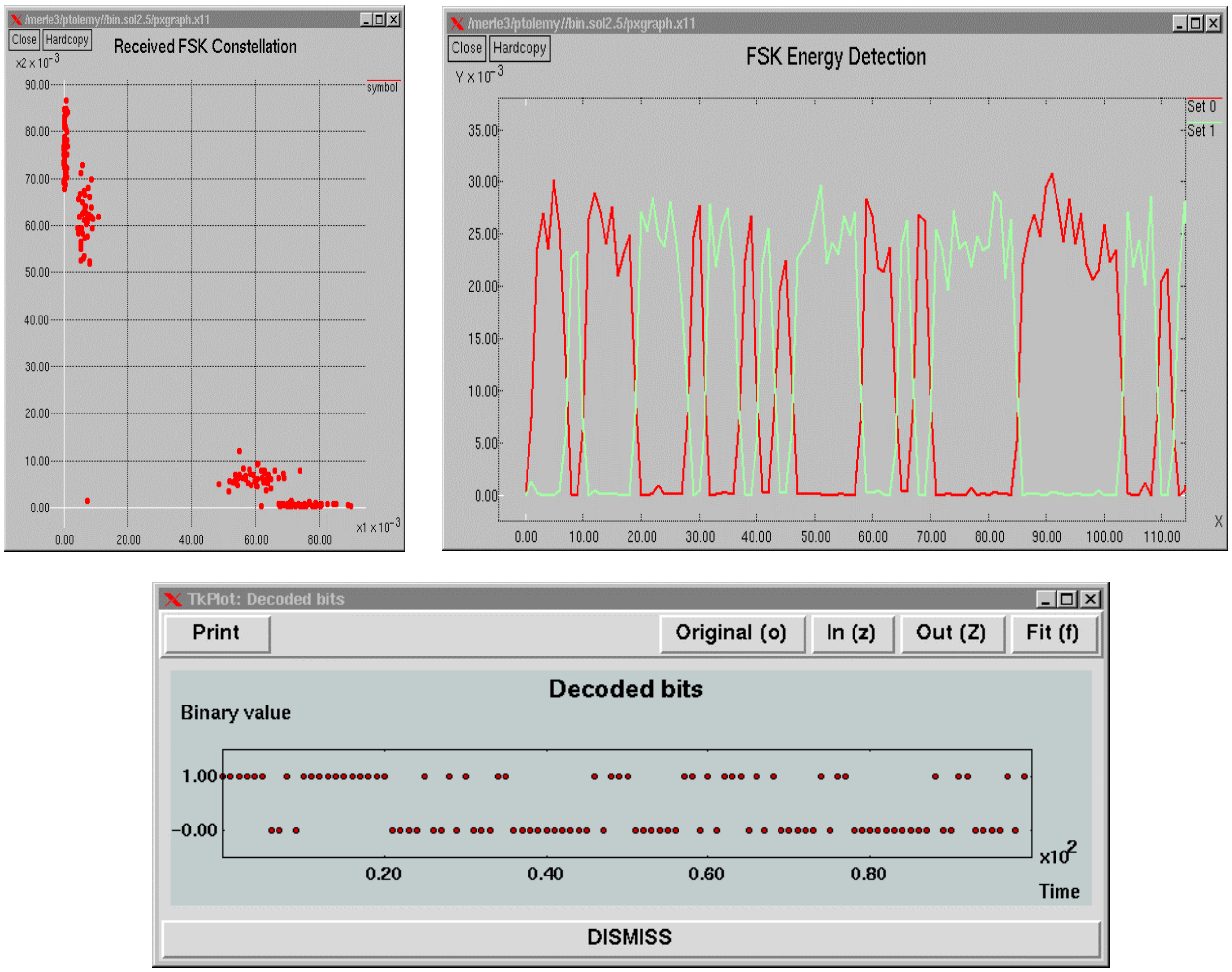
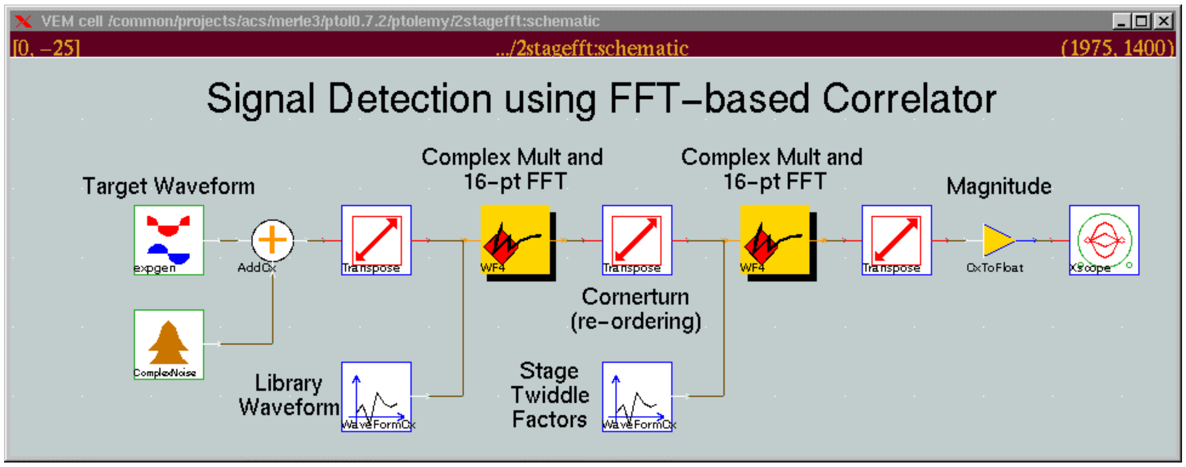


Figure 23. Winograd DFT-Based FSK Communications Receiver – Processing Results

4.4.3 Demonstration No. 2: Signal Detection using FFT-Based Correlator

The second main demonstration involved development of an ACS implementation of a signal detection scheme (template matching) using an FFT-based correlator as shown in Figure 24. This implementation involved an FFT-based complex correlator, requiring the use of two 16-point complex FFT cores in series with a cornerturn to obtain a 256-point spectrum. This demonstration was shown at the April 2000 DARPA ACS PI meeting.



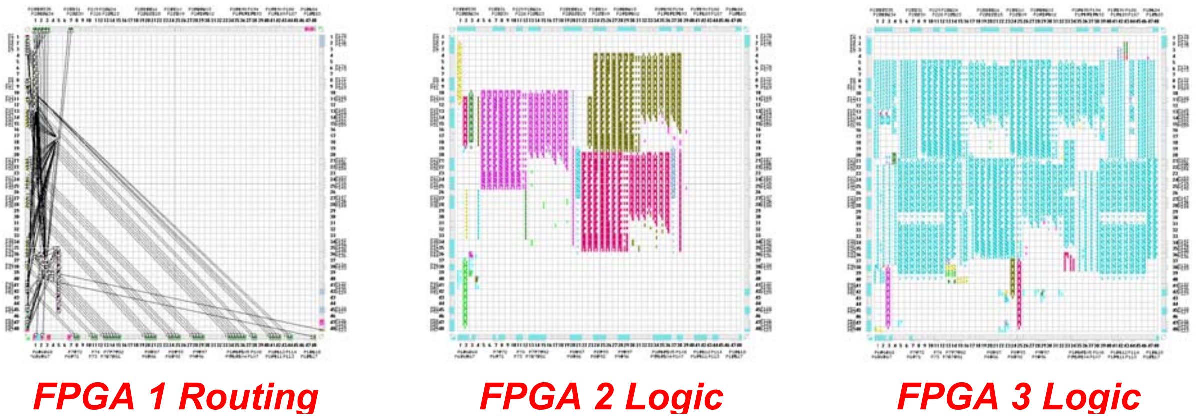


Figure 24. Signal Detection Using FFT-Based Correlator

As stated, the algorithm contains a complex multiplier followed by a 16-point FFT. The complex input data is started in the memory for FPGA 1, where the real component occupies the upper 16 bits, and the imaginary component occupies the lower 16 bits. The data is passed from FPGA 1 to the FPGA 2 via the systolic bus, where the real and imaginary components have been split into two independent signals. The memory for FPGA 2 contains the other set of complex data stored in a similar format to FPGA memory 1; it is also split and fed into the complex multiplier within FPGA 2. Once the result of the complex multiplier has been packed into a single data stream, the net result is fed to FPGA 3 via the systolic interconnect. Lastly, FPGA 3 unpacks the data stream, computes a 16-point FFT using a Xilinx standard core, packs the result into a single data stream for storage in memory for FPGA 3. This algorithm was composed into synthesizable VHDL and core EDIF files by the ACS tools.

4.4.4 Demonstrations No. 3 and 4: Acceleration of a High Resolution Radar Template-matching Application (System-Oriented High Range Resolution Automatic Recognition Program (SHARP))

The third main demonstration involved the initial workings/progress towards accelerating a high resolution radar template-matching application (System-Oriented High Range Resolution Automatic Recognition Program (SHARP)). This demonstration was shown at the October 2000 DARPA ACS PI meeting. This demonstration again showed the multi-FPGA capabilities of the ACS tools and concentrated on the normalization routine segment of the SHARP algorithm. Continued efforts, including efficient HW designs for accelerating both the normalization and correlator portions of the algorithm, progressed to complete development of the full complex SHARP algorithm for Automatic Target Recognition (ATR), the fourth and final demonstration.

Due to the significant and close involvement of the demonstration efforts for the last two main demonstrations and the significance of accelerating the SHARP ATR algorithm, the SHARP algorithm itself and the details of the efforts involved in analyzing and mapping the algorithm to ACS are described here.

4.4.4.1 SHARP Algorithm

The Air Force Research Laboratory (AFRL) at Wright-Patterson Air Force Base (WPAFB) has developed the SHARP algorithm as an algorithm that implements a forced-decision method. The algorithm attempts to classify targets given their high range resolution (HRR) radar signatures. The basic premise of the SHARP algorithm is shown in Figure 25.

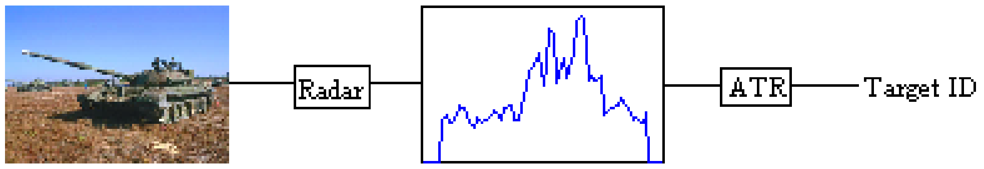


Figure 25. SHARP Approach to ATR

The SHARP objective is to develop and mature advanced air-to-ground HRR ATR capabilities for transition into suitable operational Air Force airborne platforms. The SHARP algorithm that was designed implemented a forced-decision ATR approach. The algorithm classifies target based on their signatures and aspect angles using an extensive template set. It is clear that in an application of this type, it is desirable to minimize the latency between data collection and classification. Therefore, with customer input, we focussed our last two demonstrations on developing a framework that provided the SHARP algorithm with a significant speedup via hardware acceleration (FPGAs).

The SHARP algorithm makes use of the dataset from the Moving and Stationary Target Acquisition and Recognition (MSTAR) database to measure its performance. The subset of the MSTAR dataset used had seven classes, consisting of three BMP2 armored personnel carriers (APCs), one BTR70 APC, and three T72 main battle tanks. There were two sets of data available: the training set and the test set. The training set was taken at a 17 degree depression angle and consisted of roughly 230 different aspect angles for each class spread across the total 360 degree span. The test set was taken at a 15 degree depression angle and consisted of roughly 195 aspect angles for each class.

The results of using the SHARP algorithm to classify the entire test set are stored in a confusion matrix. Confusion matrices provide a way to measure the performance of a classification algorithm. The rows of the confusion matrix represent the test set, while the columns represent the template set. The value in row i and column j of the matrix is the percentage of signatures in Class i that were classified as Class j.

The MSTAR data was available as SAR imagery rather than HRR signatures. It was therefore necessary to perform some preprocessing of the data in order to be able to simulate HRR data and get a better estimate of the algorithm performance. The dimensions of the original SAR images were 128 x 128. These were subsampled by taking the center 101 x 70 pixels where the actual target was contained. This was done to simulate the results of the Doppler filtering which would be performed on actual HRR signatures to separate the target from the ground clutter. The subimages were then zero padded back up to 128 x 128 so that an inverse Fast Fourier Transform (IFFT) could be taken. The IFFT was taken on the dimension that was 70 pixels wide before zero padding and resulted in range signatures collected over the radar aperture (the dimension that was originally 101 pixels wide). The resulting signatures were finally deweighted in angle using an inverse Taylor window to make the energy of the signatures uniform. The result was a 128 x 128 matrix, representing 128 signatures that were each 128-wide in range. It is important to note that only the center 101 x 70 of this matrix represented real data while the rest was zero padding. This 128 x 128 matrix was used as the input to the algorithm; it would be compressed to a single signature, which was then used for classification. The method of performing this compression is discussed below following discussion of the least squares fitting performed for the SHARP algorithm.

The SHARP algorithm minimizes an error metric and bases its classification decision on that minimum error. Consequently, it made sense to begin our development of the algorithm by defining this metric. Figure 26 shows the basic algorithm flow for the SHARP/HRR algorithm.

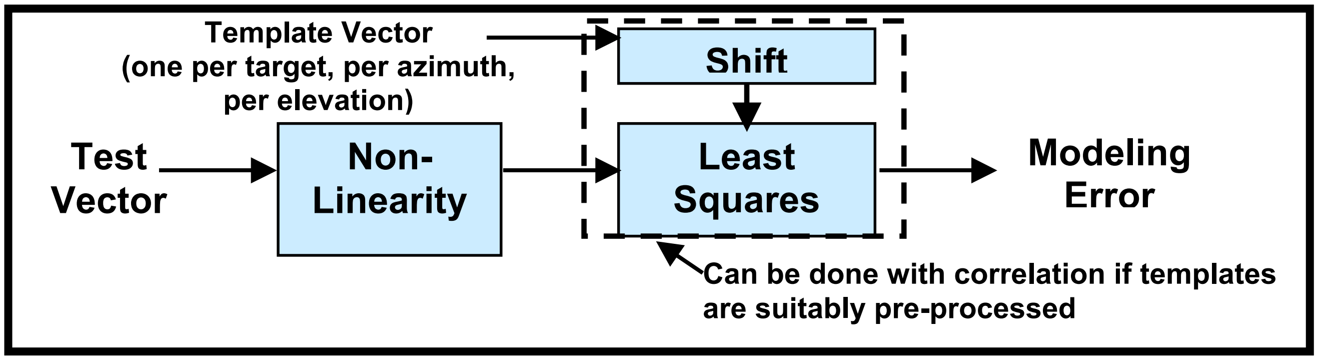

Template Vector
(one per target, per azimuth, per elevation)
Shift
Test Vector
Non-Linearity
Least Squares
Modeling Error
Can be done with correlation if templates are suitably pre-processed

Figure 26. SHARP Approach Using Minimum Error

Given a target vector and a template vector, the SHARP algorithm tries to fit the vectors to each other by altering the bias and magnitude of the template vector. A least squares method is used to provide for the solution that minimizes the error.

Now that we had defined a metric by which to compare the various template signatures, we had to define the template set to which this metric would be applied. The template set spans two dimensions which we had to take into account: aspect angle and range. Clearly, one possible solution was the brute force attempt; we could have considered template vectors spanning all aspect angles and considered all possible range shifts and minimize the error across all the resulting comparisons. However, this was viewed as a highly inefficient approach that fails to utilize knowledge of the data.

The SHARP algorithm assumes that the signal information is contained in the center 70 range gates of the entire 128-gate span. The algorithm further assumes that the templates are already close to being correctly aligned. Consequently, the decision was made to only consider shifts within five range gates. This results in a dramatic savings in computation when compared with considering all 139 possible range shifts. In order to facilitate performing the eleven necessary range shifts, the target vector is padded on each end with five zeros to produce an 80-wide vector. The eleven shifts are generated by taking subvectors (1-70, 2-71, ….., 11-80) of the padded vector.

To cut down on the number of aspects we had to consider, we made use of the fact that we know the aspect angle of the target vector. It therefore made sense to choose some subset of angles centered around the target aspect. The SHARP algorithm only considers templates whose aspects are within 5 degrees of the aspect angle of the target. This is again a huge savings since a 360-degree span has been reduced to a 10-degree span.

The result of these decisions was to reduce the number of error calculations that must be computed to a manageable number. The template data contained signatures that were multiples of 1 degrees apart in aspect. Consequently, the maximum number of aspect angles to be considered for any given target aspect angle a is eleven (i.e. a-5, a-4, …., a+5). Therefore, we saw that for a given target signature, we had to perform at most 11 (aspects) x 11 (ranges) x 7 (classes) = 847 error calculations.

The method of performing the compression on the 128 x 128 input data matrix discussed earlier involved first preprocessing to remove automatic gain control and range effects from the signatures in order to improve classifier performance. This was achieved through a normalization.of a power vector formed from each input vector range gate magnitude values. Before the resulting signatures could be averaged to produce a single vector, an additional transformation had to be performed to satisfy the constraints of the least squares fit routine.

Up to now, the preprocessing was identical for both the template and target signatures. At this point, the computation finally differed. Because we wanted the template signatures to be as accurate as possible, all 128 signatures were averaged together to obtain a single 70-gate vector. However, the target signatures should attempt to simulate the quality of real HRR signatures. Consequently, only the center eight profiles were averaged. Target vectors were finally padded with five zeros on either end to accommodate the range shifting of the least squares fit.

To establish a baseline performance, we first established the results of execution of the SHARP algorithm on the 1,362 MSTAR test signatures in two different formats. Table 4.1 shows the confusion matrix arranged by vehicle. Table 4.2 shows the confusion matrix arranged by vehicle class. Recall that element (i,j) indicated the percentage of Class i signatures that were assigned to Class j. Through these tables, we can see quantitatively how the SHARP algorithm performs.

Table 4.1: Confusion Matrix by Vehicle Using Original AFRL Code

	BMP263	**BMP266**	**BMP2c21**	**BTR70c21**	**T72132**	**T72812**	**T72s7**
BMP263	47.69%	16.92%	23.08%	3.59%	3.59%	2.56%	3.59%
BMP266	7.69%	60.00%	14.87%	4.10%	4.10%	3.59%	5.64%
BMP2c21	21.54%	12.31%	49.74%	4.62%	4.10%	3.59%	4.10%
BTR70c21	3.59%	4.10%	5.64%	76.92%	3.08%	3.08%	3.59%
T72132	3.59%	1.02%	4.59%	0.51%	74.49%	10.20%	6.12%
T72812	0.00%	2.05%	1.54%	1.54%	5.64%	75.38%	13.85%
T72s7	1.57%	4.19%	2.62%	1.57%	8.90%	10.99%	70.16%

Table 4.2: Confusion Matrix by Vehicle Type Using Original AFRL Code

	BMP2	**BTR70**	**T72**
BMP2	84.62%	4.10%	11.28%
BTR70	13.33%	76.92%	9.74%
T72	6.87%	1.20%	91.92%

We saw from Table 4.1 that the performance by vehicle was certainly not optimal. In fact, for the BMP263 and BMP2c21, less than 50\% of the signatures were correctly classified. However, we saw from Table 4.2 that the majority of misclassifications were still the same vehicle type. Consequently, we saw that while the SHARP algorithm did not perform well on a vehicle-by-vehicle basis, it did perform well on vehicle types. In fact 86.64% of the target signatures were correctly matched with their vehicle class.

We noted here that the time required for classification of the test signatures (excluding the time necessary for template formation) was approximately 21.64 minutes. While a speedup of a process that only takes 21.64 minutes to execute is not very significant, it is important to note that in a practical system, this time would be much larger. It is likely that the template set would be much more densely populated in aspect and would contain data for many depression angles. Consequently, we could expect a realistic execution time to be on the order of days. In this case, the speedup provided by hardware acceleration would clearly be significant.

We examined the results of algorithm execution not only to demonstrate what the capabilities and shortcomings of the SHARP algorithm are, but also to provide a point of reference for our further analysis. Since we were attempting to provide the same capability as the SHARP algorithm, but with hardware acceleration which will significantly decrease the execution time, we needed a quantifiable way to compare the results. We therefore needed to ensure that executing some or all of the SHARP algorithm in hardware did not significantly change the confusion matrices from those in Tables 4.1 and 4.2

4.4.4.2 Software Analysis

In attempting to accelerate the SHARP algorithm, it was deemed important that the original software be optimized as much as possible. Achieving a 10x speedup through the use of hardware acceleration is not very impressive if a 5x speedup can be gained simply by improving the software. Thus, it is was determined only possible to get a true measure of the impact of hardware acceleration by comparison to a software implementation that was fully optimized.

Additionally, this software optimization can typically lead to better hardware performance. Naturally, functions that require few computations will execute faster in both hardware and software. However, the lack of computational elements results in extra FPGA area. Therefore, the existing blocks can be implemented using larger area designs which can either improve the precision of the calculation or reduce its latency. Consequently, it was clear that software optimization was an important step towards hardware acceleration.

It was important to recognize that the SHARP algorithm consisted of two separate parts. Template signature formation comprised the first part while target signature formation and classification made up the second part. The important distinction between these parts is that the former could be computed and stored ahead of time.

The compilation of the template library was not be considered part of the algorithm execution since the template data was static. Once the templates had been computed, they could simply be stored and read on subsequent execution. Consequently, our analysis of the SHARP algorithm focused only on the formation and classification of target signatures.

We began our analysis by examining the execution of the SHARP algorithm to determine the relative amounts of computation involved. A profiling of this sort was helpful to determine what calculations dominated the execution time and would therefore demonstrate the greatest benefit from an acceleration. Conversely, it also prevented us from undertaking a detailed analysis to gain a significant speedup in a computation that did not make up a significant portion of the algorithm execution time. Through analysis of the initial execution of the algorithm, we were able to better focus our efforts and achieve the greatest speedup.

All timing tests were conducted on a Sun Microsystems Ultra 5 running at 360 MHz with 320 MB of RAM. As mentioned above, we ignored the time required for template formation. Consequently, we broke our analysis into three categories: target loading/preprocessing, least squares fitting, and classification. The timing is shown in Table 4.3 for all seven target classes, which we recall consisted of 1,362 signatures (128 x 128 matrices). All timings are shown in seconds.

Table 4.3: Timing Analysis of Original Software

	Loading/Preprocessing	**Least Squares Fit**	**Classification**
BMP263	24.55	164.16	.11
BMP266	24.89	160.08	.11
BMP2c21	24.56	162.55	.11
BTR70	24.62	161.00	.11
T72132	24.54	156.27	.11
T72812	25.05	161.48	.11
T72s7	24.50	159.52	.11
Total	**172.71**	**1125.06**	**.77**

As expected, the first priority for acceleration was the least squares fit routine. Least squares fitting required many error calculations, each of which required several vector multiplications. Additionally, the least squares fit was performed on many vectors. Consequently, it was no surprise that it was responsible for most of the execution time.

We also saw that the secondary area for acceleration was the preprocessing. While we could not do much to improve the load time of the data, we could improve the time required to preprocess the data into power-transformed HRR signatures. Consequently, we also focused on the power transformation and attempted to improve its execution time.

As identified earlier, the goal of the least squares fitting was to minimize the error between the target signature and the class to which it was assigned. Specifically, the SHARP algorithm determined the class that minimized the error between the target signature and all template signatures that were within 5 degrees in aspect and within 5 range gate shifts.

The approach taken by the original AFRL code to calculate the fitting error was relatively straightforward. The AFRL code employed a QR factorization in order to reduce the amount of calculation necessary. In using the QR factorization approach, it provided a savings in the amount of multiplications that were required. For example, recalling that in order to compare one target signature to one template signature, there were eleven range shifts to be considered, straightforward calculation resulted in 11 x 430 = 4,730 multiplications. However, in the QR approach, factorization only had to be done once per template. Consequently, only 280 + 11 x 161 = 2,051 multiplications were required.

While this was clearly an improvement over the straightforward approach, it could be improved further using a matched filter.

We therefore determined what a matched filter approach would require. Specifically, it was determined that by removing the bias from the template and target vectors and then normalizing them to unit magnitude, we could accomplish the necessary error minimization by maximizing the correlation between the two vectors. However, it was important to realize that calculating the bias and normalizing were expensive calculations. Consequently, it was desirable that these operations not be performed frequently.

Recall from previous discussions above that a single target vector was compared against seven template classes. Within each template class, signatures were only considered if their aspect angles were within 5 degrees of the target signature. Additionally, the target signature was shifted up to 5 range gates in either directions to align the signatures. Clearly, the choice of template vector had no bearing on processing the target vector. However, a bias removal and normalization was necessary for every range shift using the AFRL approach for range shifting.

The range shifting done by the algorithm was performed by padding the original 70-vector with 5 zeros on either end and then taking a window of size 70 from the resulting 80-vector. However, it was determined that a problem exists with such an approach. We examined an 80-vector obtained by zero-padding one of the bmp263 target signatures and the 11 range shifts of
that signature.

The 70 samples containing target information were elements 6 through 75 of the 80-vector. However, every window besides the center one left out at least one of these samples. In the extreme case, windows 1-70 and 11-80 left out 5 samples of the original vector. In many cases, this did not matter because the actual target signatures were generally smaller than 70 range gates, but in some cases, real data samples were lost. Consequently, the mean of the vector could potentially change with every range shift, requiring constant bias removal and normalization.

Instead of accepting this conclusion and taking the resulting performance degradation, we proposed a modification to the algorithm which enabled the bias removal and normalization to be performed only once for each target vector. This modification was to ensure that samples were never dropped. This was accomplished by providing additional zero padding and taking a larger window. Specifically, the original 70-vector was padded with 10 zeros on either end resulting in a 90-vector. The 11 range shifts were then performed by taking a subwindow of size 80. Note that the target samples were contained in elements 11-80 of the 90-vector. We then showed that each range shift included all 70 of the target samples. Consequently, the mean did not change across the range shift, implying that bias removal and normalization only needed to be performed once for each target vector.

Making this modification did slightly change the performance of the algorithm since forcing all the target samples to be kept changed some of the error calculations from the original version. The modified performances was actually slightly improved because by not dropping samples, we were using a more accurate measurement of the least squares error.

To compare against M template signatures, QR factorization required 2,051M multiplications, while the matched filter approach required 80 + M x 11 x 80 = 80 + 880M multiplications. In addition, we showed that adopting a matched filter approach provided a significant advantage over the QR factorization approach in the algorithm timing. Table 4.4 shows the timing analysis results for the optimized software approach.

Table 4.4: Timing Analysis of Optimized Software

	Loading/Preprocessing	**Least Squares Fit**	**Classification**
BMP263	2.79	34.22	.11
BMP266	2.53	33.93	.12
BMP2c21	2.58	34.55	.12
BTR70	2.54	34.14	.11
T72132	2.63	34.67	.11
T72812	2.69	34.20	.11
T72s7	2.57	33.64	.12
Total	**18.33**	**239.35**	**.80**

By comparing Tables 4.3 and 4.4, we saw that the software optimizations made a significant impact upon the execution time of the algorithm. The total execution time was reduced from 1298.54 seconds to 258.48 seconds. Additionally, the Ratio between least squares fitting time and preprocessing time had almost doubled, further supporting the supposition that least squares fitting dominated the execution. Therefore, accelerating the least squares fitting remained our highest priority.

4.4.4.3 Hardware Analysis

Following the software analysis, we examined the SHARP algorithm from a hardware standpoint. Thus far, hardware had not been a factor in our analysis. Our original approach in mapping the SHARP algorithm was to include hardware mappings for both the power transform and least squares portions of the algorithm. However, it was ultimately decided, due to time constraints and the desire to focus our efforts on the least squares section which provided maximum benefits from hardware acceleration, that the power transform would remain in software.

The least squares fitting consisted of two distinct parts. First, the input vectors had their biases removed and their magnitudes normalized. Second, the correlation between the target vectors and the appropriate template vectors was calculated.

A block diagram of the desired calculation for the bias removal and normalization is shown in Figure 27.

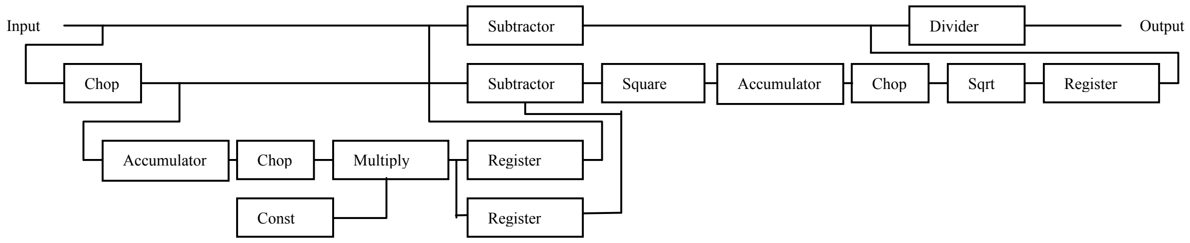


Figure 27. Block Calculations for Bias Removal and Normalization

In order to calculate the bias, a Chop was used to take an 80-subvector of the input 90-vector. The vector was accumulated and a Chop was used to extract the total sum. The bias was finally calculated by multiplying by a constant of 1/80. The bias was then repeated 80 and 90 times, so that it could subtracted from both the original 90-vector and the 80-subvector. Note that this could also have been achieved through the use of one Register and Subtractor operating on the 90-vector. However, because the magnitude which was calculated next operated on the 80-vector, an additional Chop would have been necessary, increasing the latency of the design. Consequently, two Registers and Subtractors were used to operate on both the 90-vector and the 80-subvector.

To calculate the magnitude, the 80-vector was squared and accumulated. Once again, a Chop was used to get the total sum. The square root of this sum was the desired magnitude. The magnitude was repeated 90 times and used to normalize the 90-vector by using a Divider.

We note once again that we had to manually set the output range of the divider. Fortunately, the purpose of this divider was to divide the elements of a vector by its magnitude, so we fixed its output range to [-1,1]. Aside from this, all precisions were left for the wordlength analysis tool to assign.

The normalization design was estimated to be large enough to require an entire FPGA dedicated to it. Consequently, PE1 was used for normalization, requiring the signature data to be written to the memory bank of PE1. Figure 28 below shows the Ptolemy graph, the auto-generated schedule and the FPGA design layout for the normalization routine.

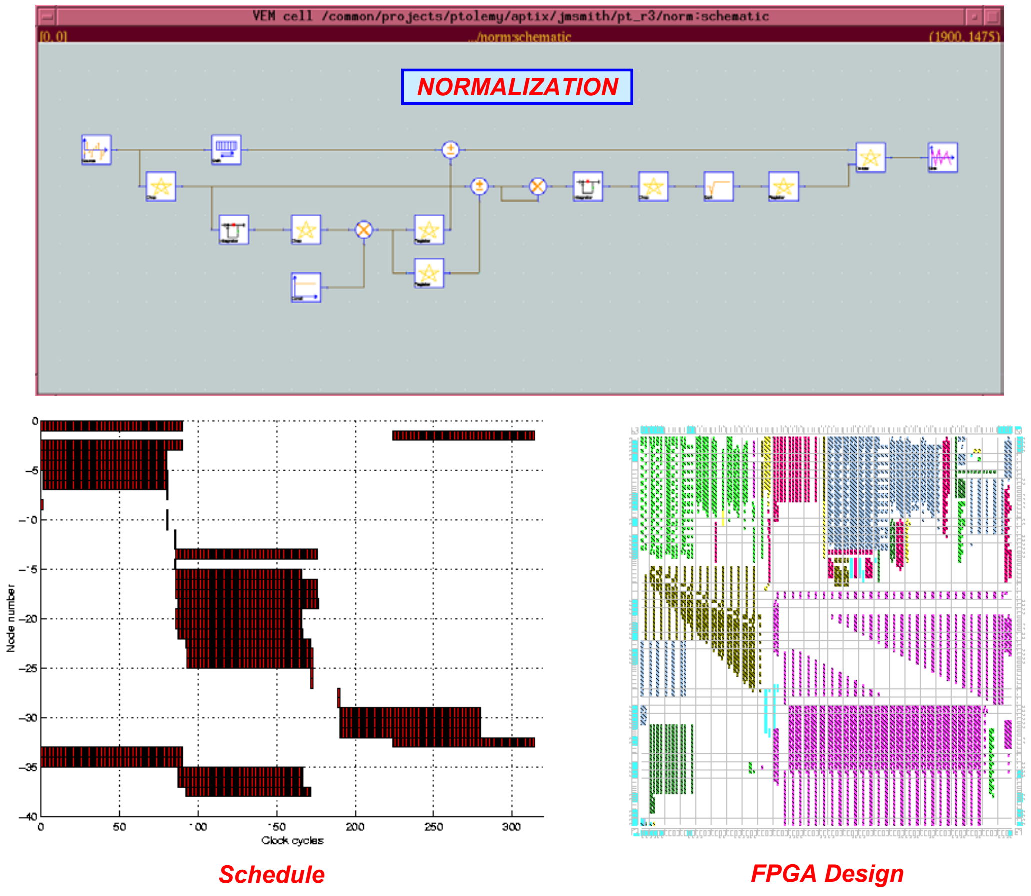


Figure 28. Normalization Routine

Figure 29 below, which provides software versus hardware results for the normalization routine (including bias removal), showed acceptable results were achieved with the normalization hardware design.

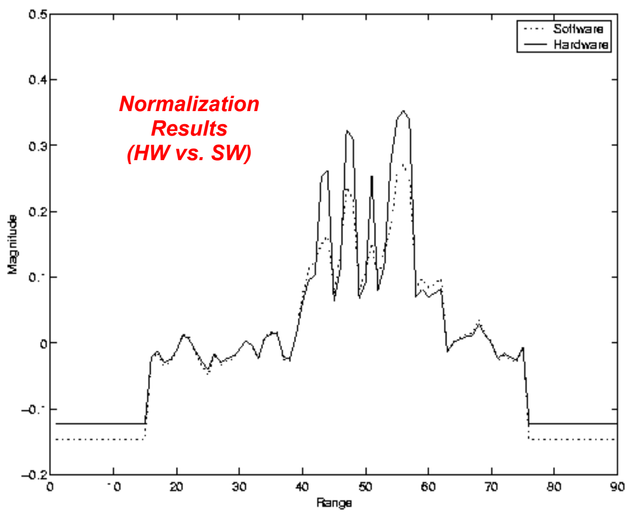


Figure 29. Normalization Results - HW vs. SW

For the correlator, there were two inputs: a target signature and an angle input which was used to produce a template signature. Once both signatures had been generated, the target signature (which was 90-wide) was passed to 11 parallel Chops which took the 11 different range shifts. Each of these 80-vectors was correlated with the template vector through the use of a Multiply, an Accumulator, and a Chop. Finally, the 11 correlations were passed through a binary tree of Maximums to produce the maximum correlation.

There were many issues to be resolved with the correlation design. The most difficult decision was how to handle the template library. In software execution, all templates within 5 degrees in aspect were considered. Due to nonuniform aspect spacing, this meant that the number of templates being considered was aspect-dependent. Consequently, there was a data-dependent control flow.

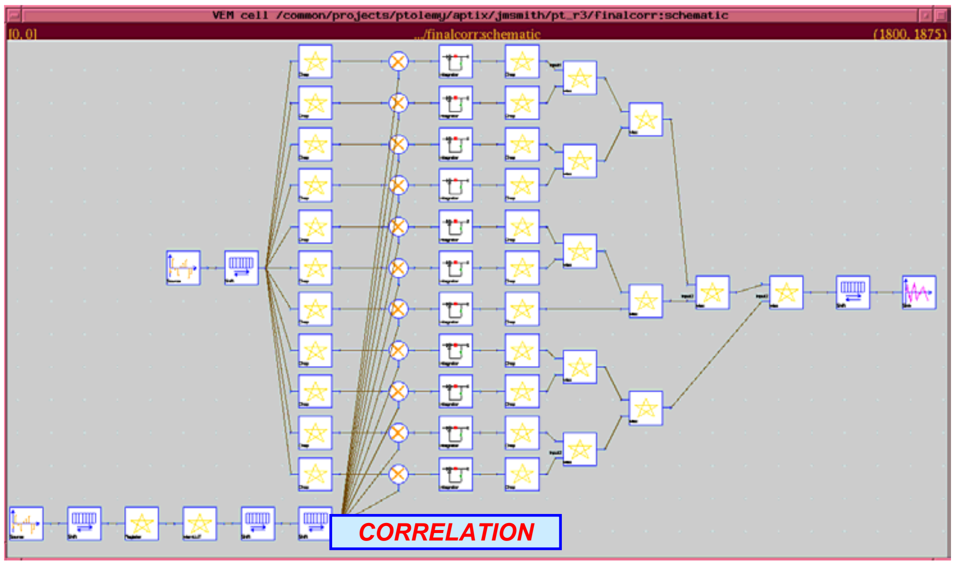
In order to remove this dependency, the template set was filled out to contain 360 templates. Because templates were spaced in multiples of 1degrees, any angles that were not represented had templates containing all zeros inserted. Additionally, to facilitate the 5 degrees aspect span, the template sets were given 5 degrees wraparound at each end. Therefore, for a given angle, there were exactly 11 templates per class to be considered and they appeared sequentially in the template library. This resulted in a template library size of 7 (classes) x 370 (vectors) x 80 (elements/vector) = 207,200 elements.

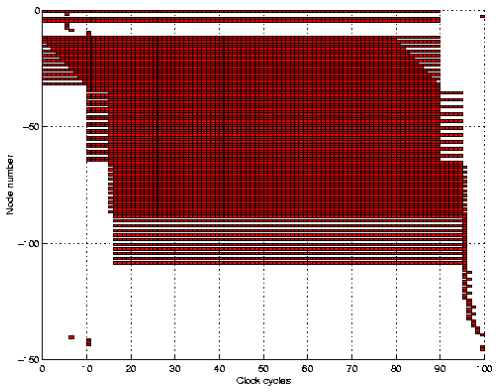
An additional problem was how the template library would be used. Because there was no way to hold a target vector in place, the decision was made to only perform one vector comparison per initiation. Note that this meant that each target vector had to be repeated 77 times (7(classes) x 11 (aspect angles/class)) for all the comparisons to be made. Consequently, a simple address generator was built into the template library which would output one signature given its starting address. Clearly, this is not the most efficient approach, but timing constraints prevented an alternate approach (i.e. a memory element could be used to hold the target vector in place) from being investigated.

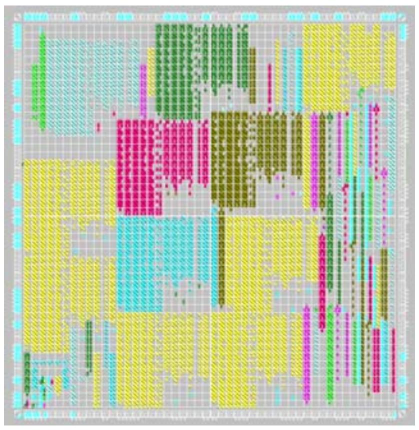
The target set consisted of 1,362 vectors. Since each vector had to be repeated 77 times, the input dataset consisted of 104,874 90-vectors, meaning 9,438,660 elements had to be written to the hardware memory. Because the memory banks only had 262,144 addresses, this dataset was broken into 37 smaller pieces. To make processing simpler, vectors were not split across pieces; rather, each download consisted of 37 90-vectors repeated 77 times (256,410 elements).

The only remaining difficulties were related to bit precision. The decision was made to put all of the correlation calculation (not necessarily the template address generation) in a single FPGA. If the correlation was not able to fit into a single FPGA, additional designs would be necessary in order to be able to route the template data to multiple FPGAs, due to the limited number of interconnects between FPGAs.

The most expensive elements in the correlation design were the Multiplies. It was not possible to fit 11 16-bit multipliers in a single FPGA. Consequently, the multipliers were reduced to 12-bit input precision. This still resulted in accuracy down to 2^{-10} (assuming <1.12> precision), which would be sufficient for determining maximum correlations. Figure 30 shows the Ptolemy graph for the correlation design along with the corresponding schedule and FPGA design.


VEM cell /common/projects/ptolemy/aptix/jmsmith/pt_r3/finalcorr:schematic
[0, 0]
.../finalcorr:schematic
(1800, 1875)
CORRELATION





Schedule

FPGA Design

Figure 30. Correlation Routine

At this point, since the crucial design elements had been decided, the only remaining issue was to make FPGA assignments. Since there were only two large components (the normalization and the correlation), this was relatively easy. The data input and normalization had already been placed in PE1. The address input was sent to PE2. The template library and address generator were placed in PE3. Finally, the correlator and output memory were assigned to PE4.

In performing final comparison estimates, our first concern was the fidelity of the algorithm. Whether the hardware version of the algorithm executes faster than its software counterpart is irrelevant if the hardware version performs very poorly. Thus, we determined and analyzed the confusion matrices obtained through software execution, but with the least squares fit performed using the precisions we chose for the hardware implementation.

We showed that the performance degraded slightly due to the use of fixed-precision calculation. However, this degradation did not appear to be very significant since the fixed point results were within 2% of the floating point results.

The full hardware implementation of the SHARP algorithm, running the matlab program interfaced to the hardware accelerated normalization and correlation algorithm portions, was successfully completed for the fourth and final program demonstration. The project team successfully met and far exceeded its target goal for acceleration of the SHARP algorithm provided to us by AFRL. An end-to-end acceleration of 35X was achieved, with successful identification of all targets represented in the test data set provided by AFRL. A 10X speedup had been established as a milestone for success. In addition, the 35X speedup is in comparison to the software-only version of the program that incorporates modifications developed by the program team to first optimize the original SHARP program provided to us. In comparison to the original pre-optimized SHARP program, the achieved acceleration is over 175X. These results are clearly a significant improvement over the software execution time of 258.48 seconds and indicates that even with a suboptimal design, a significant benefit can be gained over software execution.

5 Summary

The Algorithm Analysis and Mapping environment program has been a four year effort to develop tools to allow the specification, analysis, and "compilation" of signal processing algorithms to reconfigurable hardware. The program based its work on the Ptolemy system from the University of California, Berkeley. BAE SYSTEMS, working with the Ptolemy group as a subcontractor, has developed extensions to the Ptolemy system that enable a signal processing algorithm developer to specify a signal processing dataflow graph, process data through the graph in simulation, analyze the finite precision effects likely to occur if that graph were implemented in hardware, tradeoff precision for hardware resource and generate synthesizable VHDL both for the dataflow graph as well as for timing and control logic and interface logic.

The tools developed on this program were implemented in C++ and Matlab through extensions to the Ptolemy environment. A new capability, called the ACS Domain, was added to the Ptolemy tool and is available for distribution through the Ptolemy website.

The goals and objectives of this program identified in Section 2 have been successfully achieved.

Our approach to meeting the challenging objectives of this program consisted of four major thrusts, each of which we successfully employed. The first was to focus on a system that starts with a high level language input and restrict ourselves to a limited application domain. In our case we accomplished this through use of the dataflow domain of signal processing algorithms in the Ptolemy system. The second thrust was to support algorithm development at the bit level by combining analytical and simulation methods to help the algorithm developer optimize performance. This included development of methods for providing the algorithm designer with feedback about the implications of algorithm implementation decisions. We included in this capability provisions to rearrange algorithms as required by implementation requirements. The third thrust was to leverage the structure of signal processing algorithms to improve algorithm mapping. We took advantage of the coarse-grain nature of signal processing dataflow graphs to improve partitioning of algorithms in space and time. We also investigated the use of scheduling tools, developed for mapping signal processing dataflows onto programmable processors, to develop hardware schedulers that are used in adaptive computing system implementation. The fourth thrust included providing a library of signal processing building blocks, with their hardware implementations, and the ability to rapidly compose these blocks as well as the ability to automatically generate the software to drive the hardware.

As the program progressed and the tools capabilities developed and matured, four main demonstrations, including associated software releases, as well as incremental demonstrations to test and stress the tool capabilities, were developed.

A signal processing algorithm developer now has an available and viable framework and tools in which to implement signal processing algorithms in an adaptive computing system in a dramatically reduced time frame.

6 References

[1] P. D. Fiore. "A Custom Computing Framework for Orientation and Photogrammetry". MIT Ph.D. Thesis, June 2000.

[2] I. Beichl and F. Sullivan. "The Metropolis algorithm". Computing in Science and Engineering, pages 65-69, January 2000.

7 Acronyms

ACS: Adaptive Computing Systems

AFRL: Air Force Research Laboratory

APC: Armored Personnel Carrier

ATR: Automatic Target Recognition

CLB: Configurable Logic Block

DFT: Discrete Fourier Transform

EDIF: Electronic Design Interchange Format

FPGA: Field-Programmable Gate Array

HRR: High Range Resolution

IFFT: Inverse Fast Fourier Transform

LSB: Least Significant Bit

LUT: Lookup Table

MCMC: Markov Chain Monte Carlo

MSTAR: Moving and Stationary Target Acquisition and Recognition

MTI: Moving Target Indication

RMS: Root Mean Square

SAR: Synthetic Aperture Radar

SDF: Synchronous Dataflow

SHARP: System-Oriented HRR Automatic Recognition Program

VHDL: VHSIC Hardware Description Language

VHSIC: Very High Speed Integrated Circuit

WPAFB: Wright-Patterson Air Force Base