LAMP: A TOOL SUITE FOR FAMILIES OF FPGA-BASED COMPUTATIONAL ACCELERATORS

Tom VanCourt *Martin C. Herbordt*

Department of Electrical and Computer Engineering
Boston University, Boston, MA 02215
herbordt|tvancour@bu.edu http://www.bu.edu/caadlab

ABSTRACT

Field-Programmable Gate Arrays (FPGAs) are becoming increasingly attractive as computation engines: they are currently being integrated into supercomputers as application accelerators. In order for widespread use of FPGA-based accelerators to be practical, however, design tools must resolve a number of conflicting needs: application-specific tuning versus wide applicability, stability of software investment versus use of the most recent and powerful acceleration hardware, and customization of complex computing structures by end users who lack logic design skills. We describe how the LAMP tools address these conflicts and report performance results from experiments in creating families of application accelerators.

1. INTRODUCTION

Recent generations of FPGAs offer computation resources that include hundreds of arithmetic units with aggregate computational capability of over 10^{10} operations per second, and on-chip memories with aggregate bandwidth over 1 terabit/sec. FPGAs are very attractive as computation platforms for a wide variety of calculations, including computational chemistry [1], string processing [2], and others. Despite isolated successes, however, FPGAs are not widely used as accelerators for general computing.

We present Logic Architecture Model Parameterization (LAMP), a tool suite that addresses some of the conflicting requirements in FPGA-based application acceleration. The first contradiction is that an accelerator's logic design must be highly tuned to its specific application in order to achieve the $100\times$ to $1000\times$ speed-ups desired, but must still be general enough to handle wide ranges of applications. The second is that end-users of application accelerators must be able to modify algorithms at will, but efficient accelerator design requires significant hardware expertise. The third is that accelerator users want to use the full capacity of the most powerful FPGAs available, but do not want to change their accelerator designs when porting to larger FPGAs. These contradictions emerge from a few observations about real applications: first, that applications occur in wide families, but efficient logic designs are point solutions; second that application expertise and logic design skills nearly never appear in the same person; and third, that FPGA capacity increases steadily over time, but current design tools do little to support application scalability.

THIS WORK WAS SUPPORTED IN PART BY THE NIH THROUGH AWARD #RR020209-01 AND FACILITATED BY DONATIONS FROM XILINX CORPORATION.

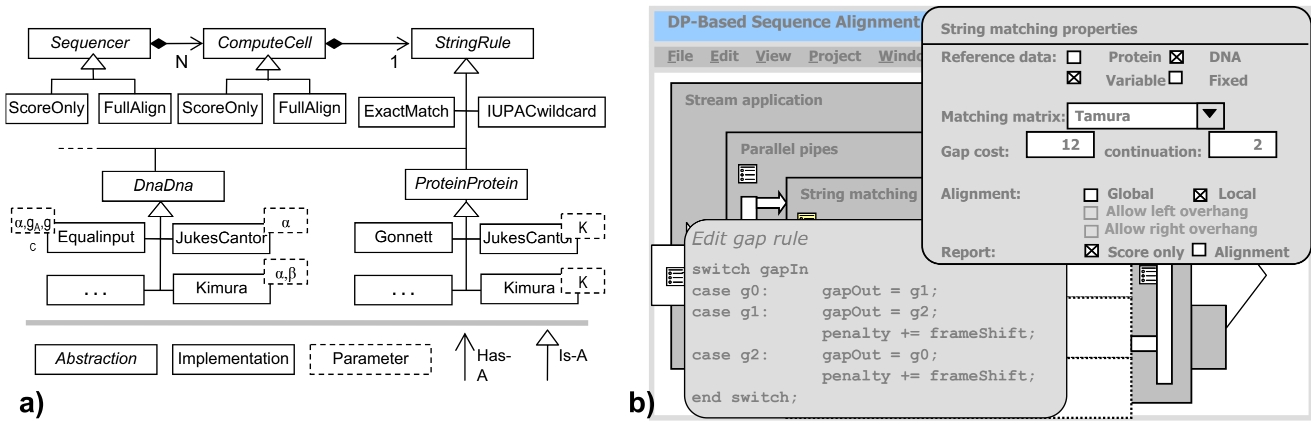
The LAMP tool suite meets these needs (i) by allowing the logic designer to easily define related domains of applications that share common computation structure (LAMP *Models*), (ii) by breaking the dependency relationship between the application's end user and the FPGA logic designer, and (iii) by automating the mechanisms for increasing parallelism up to the limit set by the FPGA's resource constraints. Through the rest of this article, we show how LAMP addresses the combined goals of high performance, flexibility in the computations addressed, end-user control over the application details, and automated scaling of logic resource utilization.

2. OBSERVATIONS FROM APPLICATIONS AND RESULTING REQUIREMENTS

We have chosen two broad families of applications as example targets for FPGA-based acceleration using LAMP. Each of these is an application domain, representing an entire range of variations within one computational framework.

Modeling Rigid Molecule Interactions (MRMI) [4]: 3D correlation is a well-established technique for estimating the strength of interactions between two molecules, and for determining what relative offset and rotation gives the best interaction [5, 6, 7]. We have shown that FPGA acceleration using direct summation can give $100\times$ to $1000\times$ speed-ups compared to PC-based serial code [1]. Direct summation can also handle multiple and non-linear chemical phenomena difficult or infeasible for transform-based techniques.

Approximate String Matching using Dynamic Programming (DP) [2]: These are staple computations in bioinformatics and text processing. Hundred of variations on these



String matching properties

Reference data: ☐ Protein ☒ DNA
☒ Variable ☐ Fixed

Matching matrix: Tamura

Gap cost: 12 continuation: 2

Alignment: ☐ Global ☒ Local
☐ Allow left overhang
☐ Allow right overhang

Report: ☒ Score only ☐ Alignment

Edit gap rule

```
switch gapIn
case g0:     gapOut = g1;
case g1:     gapOut = g2;
             penalty += frameShift;
case g2:     gapOut = g0;
             penalty += frameShift;
end switch;
```

Fig. 1. **a)** UML class diagram for part of the **DP** application family. Parameter values modify behaviors, substitution matrices [3] and scoring functions. **b)** Shows a possible GUI for DP.

algorithms exist, however, differing in their alphabets, use of wildcards, character scoring functions, gap penalties, end gap policies, etc. Figure 1a shows part of the DP class structure while Figure 1b shows part of a possible DP end-user GUI. Note in particular that the user is allowed not only to vary simple parameters, but also to choose functions and even to create simple functions in a high level language. The implications for a DP accelerator generator are that it must support generation and swapping of modules, including those with differing parameter lists, together with the Hardware Description Language (HDL) module interface issues that this entails.

From our experience with creating accelerators for these computational applications [2, 4] and several others [8, 9, 10, 11, 12], we make a number of observations.

1. Applications come in families, not point solutions. Besides the issues just described, some others are that applications may allow for arbitrary scoring functions [8], objective functions [11, 4, 12], or chemical force models [4]. This observation about application families also concurs with the high degree of complex parameterization of widely used compute intensive applications; e.g., BLAST, CHARMM, and others are complex systems having dozens, if not hundreds of parameters.

2. Can't instantiate the whole family at the same time. Unlike a software system, application instances usually must be instantiated as needed. Therefore ...

3. Control must be provided, but leaf-nodes supplied by the user. The control structure, the communication, the memory access patterns, and the I/O must all be created by the logic designer, while often only the user can create the leaf nodes.

4. Generally, multiple skill sets are required. (i) Without an application specialist, it is impossible to know what is truly required in an application and what is merely an implementation artifact. For example, in MRMI, the standard technique uses FFTs with complex double precision floating point even though the goal is a correlation result, often using only two bits of information per datum. (ii) The optimal algorithm for a serial computer or MPP is often not the optimal algorithm for an FPGA. In almost all of our applications the algorithm was changed from that in the reference code; of many examples, the Steiner system implementation in [11] was particularly non-obvious. (iii) FPGA-specific logic design is sometimes required to prevent performance losses of 10× or more. One example was in MRMI where it was essential to efficiently implement FIFOs using block RAMs to pad rows for 3D rotations.

5. FPGAs and FPGA systems evolve; scaling cannot be done by resynthesis alone. It is desirable to use the FPGA system's resources fully for each application instance, independent of the needs of any other instance, and independent of the particular FPGA. In particular, unlike with most High Level Language (HLL) based programs, recompilation for a new platform (perhaps with simple parameter changes) is not likely to be sufficient to properly use the new resources, even if it is a simple increase in identical attributes.

As a result of the observations made in previous work, we determined the following set of requirements for LAMP:

1. Good interfaces between specialists, including support for customization of the accelerator by the end-user independent of the logic designer.
2. Use of existing EDA tools as much as possible.
3. No circuit-level design.
4. Scalability through recompilation and resynthesis only.
5. Reuse of control and communication; allow broad parameterization, including behaviors as parameters at the leaf nodes.
6. Extensions to EDA tools to support requirement 5.

3. UNIQUE LAMP SYSTEM FEATURES

1. System Design — Inversion of Flow-of-Control
In contrast to many graphical systems, we invert the flow-

of-control. Common graphically oriented systems provide the leaf elements (convolvers, FFT units, etc.) with the application specialist specifying the control and the data flow connecting the blocks. LAMP-based systems provide the control and the dataflow and allow the application specialist to provide *the behaviors* for the leaf nodes. Of course the application specialist is still allowed to use existing (parameterized) modules as leaf nodes.

Viewing this another way, what LAMP-based systems provide corresponds to the outer loops, loop interfaces, and major data structures that express the essential dependencies of an application family. Although the general range of calculations to be addressed is up to the application specialist initially (perhaps through a formal specification), the logic designer creates the LAMP-based system that represents the entire range of calculations within that application domain. The application specialist then specifies particular applications as desired. This extends the approach taken by the version of SA-C described in [13]. There, SA-C supports a single loop construction (for template-based image processing) and allows the internals to be varied; we provide a framework for *generalized loop construction (i.e., LAMP) while still allowing the internals to be varied.* Note that this approach provides firm boundaries between application specialists and logic designers. It is also what software developers have been doing for years: The connection between the domain model and the application-specific logic roughly matches Java's AWT or the Strategy design pattern [14].

2. Tool Extension – Functional behaviors as parameters
To support the functionality just outlined, LAMP extends the underlying HDL to allow functional behavior to be expressed as a component parameter, and to allow kinds of reuse not currently possible within the HDL's accepted standard. Existing HDLs have type systems that do not meet LAMP's requirements. For example, VHDL components allow only changes of port array and integer ranges, prohibiting many kinds of component reuse, and cannot accept functions as parameters. Balboa requires port types of leaf components to be defined before creating connections between them, and does not address non-leaf components [15]. Some systems limit inheritance hierarchies to fixed structure and depth [16]. Other HDLs [17] omit critical OO features from their synthesizable subsets. Java and other standard programming languages deal with highly variable object interactions, unlike the relatively fixed component interactions in synthesized logic. LAMP's parameter type system addresses all of these issues. It

- supports parameter types that define logic functions,
- enables kinds of component reuse that are not possible within VHDL's architecture/configuration mechanism,
- allows abstract type definitions, so that implementation details can be deferred to the particular Application Instance,
- supports arbitrary levels of partial Model refinement and concretion,
- specifies and integrates OO inheritance rules into the basic synthesizable feature set, and
- addresses static component interactions as a source of novel type safety relationships.

3. Automatic Scaling
Fortunately, accelerators for many applications consist of regular arrays of PEs, memories, etc., which can generally be expanded in one or more dimensions, with no change to the fundamental structure of the computation. There are many complicating factors, however, including the dimension of the computation, the relationships among phases of computation, and the varying types of resources required and available. In general, automatic scaling solutions can be complex as even a simple, linear computation array may involve non-linear terms for broadcast networks for control signals or for larger worst-case sums. LAMP's approach to these difficulties is to isolate the estimation logic in easily replaceable software components and to enlist the designer's aid in composing the scaling expressions. Estimation algorithms can be updated as new techniques become available.

4. LAMP SYSTEM DESIGN

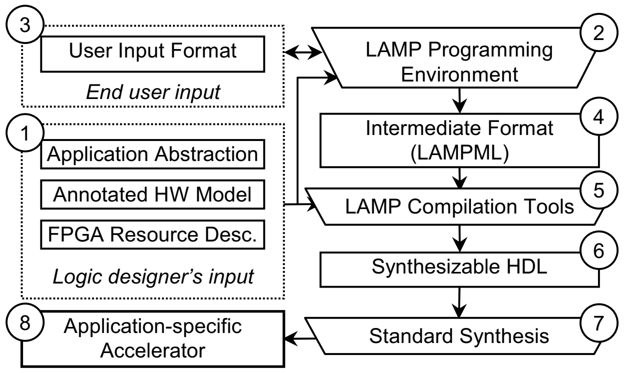


Fig. 2. LAMP design tool flow.

The system design of the LAMP design tool suite is shown in Figure 2. Some key terminology: a single LAMP **Model** created by a logic specialist corresponds to a single end-user environment for a *family of applications*; when the end-user specifies a single application using that environment, a **Model Instance** is instantiated.

(1) Logic designer's input. These are created using XML-based **LAMPML** (LAMP Markup Language) and standard HDL. The heart of the system is the **Annotated HW Model**, or just **Model**. The Model specifies the communication paths, control mechanisms, and host interface elements that are constant across all members of the application family. These accelerator structures are expressed in standard VHDL (in our current environment), annotated using LAMPML tags.

These tags, which are based on the Application Abstraction, show where application-specific features affect the logic design. The Model also includes the periphery logic that connects the FPGA core to its board and host environment.

The **Application Abstraction** is the bridge between the application specialist and the logic designer. Like a physical bridge, it connects the two. It also narrows the connection to a single point of contact between them, blocking the incidental facts of each implementation that could unintentionally have affected the other. In particular it is an abstract description of the data types and functions that that are left undefined by the Model; i.e. which are to be supplied by the application specialist or end-user.

The **FPGA Resource Description** has the amount of each computing resource (cells, block RAMS, hard multipliers, etc.) available on a given FPGA (and eventually, cluster of FPGAs) and is used in automatic scaling. The abstract definition of the FPGA Resource Description can be used to define the general form of resource allocation expressions, without need for exact values until the actual FPGA is chosen.

(2) LAMP Programming Environment. The Programming Environment naturally accepts input from the application specialist; however, (and most significantly) it *also accepts input from the Application Abstraction.* The latter identifies the functions and data types that are used, but undefined, in the Model (corresponding to the white-space in the application illustrations in the previous section). The Programming Environment uses the input from the Application Abstraction to create a semi-graphical user interface through which information is supplied to the Model.

(3) User Input. The application specialist makes configuration choices at the Programming Environment's graphical UI. The user enters function selections and/or data type definitions and function definitions in a simple C-like representation, filling in text fields within the GUI. The programming environment saves the human-readable form of input for UI purposes, but converts it into validated LAMPML for further processing.

(4) LAMPML Intermediate Format. Although the LAMPML Intermediate Format is editable text data in XML format, very few users would find it a natural or convenient way of expressing application logic. The LAMPML intermediate format is readily machine readable, however, and separates the input and parsing issues from the compiler's internal processing. This corresponds to the compiler's intermediate code, a step between a compiler's front end and back end that is normally not visible to the user.

(5) LAMP Compilation Tools. The LAMP Compilation Tools accept the Model, defined in terms of the Application Abstraction, and combine it with the LAMPML from of the user input (item 4 in Figure 2). This phase includes synthesis estimation and scaling, allowing the largest possible computation array for the logic resources available and required. The output from this compilation is Figure 2's item 6, the synthesizable HDL. These tools do the real work of combining the application-specific code (optimization data and functions) with the hardware model. This stage integrates user logic with the HDL application framework. It also performs synthesis estimates for the code provided by the user, and evaluates estimation functions provided in the hardware model. Together with the FPGA Resource Description, these imply the largest number of computational units that the available FPGA fabric can support. The compilation phase also generates interface code for the host programming environment. This small amount of code, currently presented in ANSI C, establishes the common data formats shared between the host application and the hardware accelerator. These code fragments supplement the hardware vendor's device drivers.

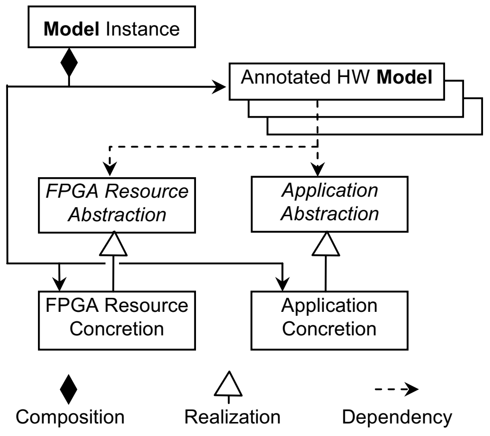


Fig. 3. LAMP UML class diagram.

(6) Synthesizable HDL output. LAMP tools do not perform HDL synthesis directly, but rather create synthesizable HDL. This leaves synthesis, place-and-route, etc. to other EDA tools. This has the advantage of (mostly) keeping the LAMP tools and hardware model independent of specific FPGAs, allowing easier porting and scaling. It also allows LAMP to take advantage of the lower-level compiler optimizations available in current HDL tools.

(7) Standard synthesis tools. LAMP works with any available synthesis tools; it has no architectural dependency on any particular HDL or compiler. That said, it is understood that tool-dependent pragmas or VHDL attributes in synthesizable HDL code may be critical for achieving performance and resource allocation goals. To date, those tool-dependent features have all been managed within the Model.

(8) Application-specific accelerator. The output is a binary image ("bit file") for implementing the accelerator in the target FPGA. It is possible to create and reuse multiple bit files to serve different purposes.

Application accelerators constructed with LAMP have

Table 1. Performance of LAMP-generated MRMI accelerators. Speed is in 10^9 score/accumulate operations per second (GSAC/s). FPGA is a Xilinx XC2VP70.

Force law	Voxel (bits)	Logic (slices)	Clock (ns)	PEs ($= n^3$)	Speed GSAC/s
K-K	2	11	20.8	2744 (14^3)	131.9
GSC	2	11	19.4	2744 (14^3)	141.9
PSC	7	21	21.7	1331 (11^3)	61.3
ACP	5	44	32.7	729 (9^3)	22.2
ES	6	35	26.1	729 (9^3)	27.9
SNorm	7	16	21.6	1331 (11^3)	61.7

the structure shown by the UML [18] class diagram in Figure 3. A Model Instance, an actual accelerator in this application family on this actual hardware, is a binding of resource and application concretions to the Annotated Hardware Model. Once the Model is defined, these relationships are constant across all actual implementations of accelerators for this family of applications. The FPGA Resource Description is instantiated from the FPGA Resource Abstraction and together with a particular FPGA Resource Concretion. Likewise, the specifics of any one application are provided by the user in the application concretion.

5. EXPERIENCE USING LAMP

We now describe our experience with using LAMP to construct the two reference application families: MRMI and DP. Both are at the highest level systolic arrays, but have very different patterns of memory reference and communication. Important to this discussion is that *this variation also extends to various applications within each family.*

Table 1 and Table 2 contain results from a sample of the various accelerators created with the MRMI LAMP Model and the DP LAMP Model, respectively. Some of the axes of variation in DP (not all shown) are global versus local alignment, DNA versus protein, match type, termination rule, and parameterized substitution table. The two major axes of variation in MRMI are the choice of force law and the evaluation function (not shown). In Table 1, K-K represents the force law used by Katchalski-Katzir's team [7], based on a two-bit encoding for each voxel in each molecule. GSC is just slightly more complicated than K-K, but PSC uses additional information about each voxel's number of neighbors [5]. ACP is a simplified form of atomic contact potential combined with collision detection. ES is based on collision detection and electrostatic forces. SNorm uses geometric effects based on surface normal vectors [6]. Dozens of other laws could also have been addressed.

We now discuss four issues: programmability, performance, performance plus flexibility, and generality.

1. Programmability. Perhaps the most critical issue is how much logic-designer time it takes to create an FPGA accelerator for a complex application family. MRMI and DP were each created in less than six months by a graduate student with modest circuit design experience. However, the most important metric is not design-hours; rather, it is the number of design-hours per accelerator use. In this context, we now compare the LAMP approach with standard HDL-based logic design.

In creating LAMP Models, the HDL subset of the Model must still created. In fact, creating these HDL modules for generality is somewhat more complex than creating the modules for point solutions. The benefits, however, are substantial. The first is that dozens to hundreds of accelerators can now be generated *and optimized to the capacity of the FPGA* with no further intervention by the logic designer. These accelerators can, of course, be generated by any number of independent end users, and as their own experiments require. The second benefit occurs if logic designer intervention *is* again needed for a unique new feature. Since the structures are already in place, little additional time is required beyond the design of the particular component.

2. Performance. The Tables report implementations for a Xilinx XC2VP70 FPGA. For MRMI, each PE handles one voxel scoring operation per clock cycle. We report both the number of PEs and the edge dimension of the cubical computation array implemented. The array limits the size of the smaller molecule, but multiple passes can be used to handle the molecule in parts. For MRMI, the speed-ups over a 3.06GHz Intel Xeon PC range from $100\times$ to $1000\times$. For DP, the speed-ups range from $77\times$ to $215\times$.

3. Performance plus flexibility. The speed-ups are satisfying given the logic design effort and the importance of the applications. However, for any one application instance (especially in DP), a hand-crafted circuit-level solution would certainly yield even better performance. Perhaps our key result is that this is of little consequence: time and again, high-performance point solutions have been introduced, but found to be too brittle for production use. In contrast, we achieve speed-ups of two to three orders of magnitude over entire ranges of family members.

4. Generality. This addresses the question: for an application family, should a single accelerator be built that does everything, i.e., that supports all of the applications within the family, or should accelerators be generated and optimized independently? To answer this question, observe that in the former case, the accelerator structure must always run at the rate of the slowest application instance. This means that other applications pay a performance penalty. For DP this is roughly a factor of 4; for MRMI a factor of 6. This argues strongly for customizable families of accelerators, and therefore tools that address families of problems.

6. CONCLUSION

We have presented LAMP, a tool suite for generating FPGA-based coprocessors, especially for compute-intensive appli-

Table 2. Performance of LAMP-generated DP accelerators. Speed is in character comparisons per second (CC/s). FPGA is a Xilinx XC2VP70. PC has a 3.06GHz Intel Xeon CPU.

Task	Data	Match type	Logic: Slices per PE	Clock (ns)	PEs	Speed 10^9 CC/s	Speed FPGA/PC
global	DNA	Exact match	109	12.9	125	9.68	215×
global	DNA	IUPAC wildcard	108	13.7	126	9.19	204×
global	DNA	Fixed table	111	14.6	123	8.42	187×
global	DNA	RAM table	108	16.8	126	7.5	167×
local	DNA	Exact match	190	13.3	72	5.41	186×
local	DNA	Fixed table	193	15.9	70	4.4	152×
local	protein	Exact match	205	13	66	5.07	175×
local	protein	Fixed table	239	25.5	57	2.23	77×
global	ASCII	Exact equality	(Xeon PC)	0.33	1	0.045	1×
local	ASCII	Exact equality	(Xeon PC)	0.33	1	0.029	1×

cations. The key results are that using LAMP, we have had success in generating families of optimized accelerators, rather than point solutions; that this generality was achieved with modest marginal logic design effort; and that speed-ups of two to three orders of magnitude over a PC were consistently obtained.

Our current work is focused on increasing the capabilities of the LAMP tools. We are also prototyping Models for other application domains, especially those involving different scaling laws and resource utilization.

Acknowledgments
We would like to thank the anonymous reviewers for their many helpful suggestions.

7. REFERENCES

[1] T. VanCourt, Y. Gu, and M. Herbordt, "FPGA acceleration of rigid molecule interactions," in *Proc. Field Programmable Logic and Applications*, 2004.

[2] T. VanCourt and M. Herbordt, "Families of FPGA-based algorithms for approximate string matching," in *Proc. Appl. Spec. Systems, Architectures, and Processors*, 2004.

[3] M. Nei and S. Kumar, *Molecular Evolution and Phylogenetics*. Oxford University Press, 2000.

[4] T. VanCourt and M. Herbordt, "Rigid molecule docking: FPGA reconfiguration for alternative force laws," *submitted for publication*, vol. TBD, no. TBD, p. TBD, 2005.

[5] R. Chen and Z. Weng, "A novel shape complementarity scoring function for protein-protein docking," *Proteins: Structure, Function, and Genetics*, vol. 51, pp. 397–408, 2003.

[6] F. Jiang and S. Kim, "Soft docking: Matching of molecular surface cubes," *J. Mol. Biol.*, vol. 219, pp. 79–102, 1991.

[7] E. Katchalski-Katzir, I. Shariv, M. Eisenstein, A. Friesem, C. Aflalo, and I. Vakser, "Molecular surface recognition: Determination of geometric fit between proteins and their ligands by correlation techniques," *Proc. Nat. Acad. Sci.*, vol. 89, pp. 2195–2199, 1992.

[8] D. Brunina, *FPGA Acceleration of BLAST*, Masters Project, ECE Dept., Boston University, 2005.

[9] A. Conti, T. VanCourt, and M. Herbordt, "Flexible FPGA acceleration of dynamic programming string processing," in *Proc. Field Programmable Logic and Applications*, 2004.

[10] Y. Gu, T. VanCourt, and M. C. Herbordt, "Acceleratiing molecular dynamics simulations with configurable circuits," in *Proc. Field Programmable Logic and Applications*, 2005.

[11] T. VanCourt, M. Herbordt, and R. Barton, "Microarray data analysis using an FPGA-based coprocessor," *Microprocessors and Microsystems*, vol. 28, no. 4, pp. 213–222, 2004.

[12] T. VanCourt and M. Herbordt, "Three dimensional template correlation: Object recognition in 3D voxel data," in *Proc. of Computer Architecture for Machine Perception*, 2005.

[13] R. Rinker and et al., "An automated process for compiling dataflow graphs into reconfigurable hardware," *IEEE Trans. on VLSI Systems*, 1999.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[15] F. Doucet, S. Shukla, M. Otsuka, and R. Gupta, "Balboa: A component-based design environment for system models," *IEEE Transaction on Computer Aided Design of Integrated Circuits and Systems*, vol. 22, no. 12, pp. 1597–1612, 2003.

[16] F. Vermeulen, F. Catthoor, D. Verkest, and H. D. Man, "Formalized three-layer system-level reuse model and methodology for embedded data-dominated applications," in *Design Automation and Test in Europe*, 2000.

[17] E. Gimpe and F. Oppenheimer, "Extending the SystemC synthesis subset by object-oriented features," in *CODES ISSS 03*, 2003.

[18] M. Fowler and M. Scott, *UML Distilled, 2nd Ed.* Addison Wesley, 2000.