An Automated Micro-architecture Design Tool for FPGAs

Matthew Areno
Utah State University
Logan, UT 84341
matthewareno@cc.usu.edu

Brandon Eames
Utah State University
Logan, Utah 84341
beames@engineering.usu.edu

Aravind Dasu
Utah State University
Logan, Utah 84341
dasu@engineering.usu.edu

Abstract

This paper proposes a novel approach to the automated generation of hardware micro-architectures targeting FPGA devices. The approach offers both high level architecture synthesis and VHDL generation based on dataflow graphs, as well as the generation of an architecture-level functional/performance simulator that can be used for quick and accurate testing of the design. We demonstrate the use of this approach through the derivation of architectures to implement computational kernels commonly encountered in Molecular Dynamics simulations.

1 Introduction

It is well known that the performance of applications on Field Programmable Gate Arrays (FPGAs) is highly dependent on the underlying sophistication of micro-architecture design features, such as pipelining and concurrency. The complexity of real world applications and algorithms tends to push the design burden farther away from an automated process and into the realm of expert VSLI design engineers. While there have been numerous efforts to automate the conversion of algorithms written in C or C++ to synthesizable HDL, there still remains a large amount of research that needs to be carried out to fully exploit those efforts. Instead of focusing on the syntactic aspects of high level languages, we intend to compliment other approaches by focusing on the semantics. We propose a methodology to automatically derive the micro-architecture for an algorithm based on the machine independent control-data-flow-graph representation of the algorithm. To facilitate a rapid evaluation of both the functionality and performance of the generated micro-architecture, we provide a C++, approximately-cycle accurate, simulator which implements the salient features of the micro-architecture, but in a software-based environment amenable to rapid-turnaround testing and evaluation.

Using FPGAs as targets, this tool creates both a VHDL and C++ representation of a given loop-unrolled dataflow graph (DFG). Generation of these DFGs is facilitated through the use of a new visual modeling tool, based on the Generic Modeling Environment (GME) [1]. The DFGs are composed of standard mathematical operations, such as addition, multiplication etc, and capture the semantics of a software algorithm.

From these DFGs, a micro-architecture is generated by way of a modified force-directed scheduler (FDS) [2]. Our modified FDS implementation is capable of performing loop unrolling, critical path relaxation, and graph decomposition, in an attempt to derive a near-optimal hardware design implementation. The FDS implementation simultaneously determines the schedule and resource allocation appropriate for a given loop-unrolled DFG. From this information, a VHDL representation is generated that targets the Xilinx ISE development environment. Additionally, a C++ simulator is also generated that may be used to evaluate the design and provide quick feedback on the impact of design alternatives.

To elucidate the details of this methodology and associated tool, the paper has been structured in the following manner. Section 2 will present related work that has been performed in this area. Section 3 will present information of the development of the GME meta-model and DFG generation. Section 4 will discuss the FDS implementation and environment generation prior to the VHDL and C++ code generators. Section 5 will discuss the generation of VHDL and C++ code representations. Section 6 will show the results that have been obtained through the use of the tool. Finally, section 7 will present additional work to be done and the conclusion of this paper.

2 Related Work

Several tool are available today capable of generating hardware architectures directly from higher level languages, such as C. These include, but are not limited to, Handel-C [3], HardwareC [4], Mitrion-C [5], and PACT HDL [6]. In addition to C-type languages, other tools derive architectures from assembly language [7], Matlab [8], and Simulink [9]. Such approaches attempt to raise the level of abstraction offered to the user to support hardware design, presenting an environment that is more familiar to a larger community of developers. The

downside of this approach is that by using too high of an abstraction level, the tool is not able to capture all of the relevant information that may be available through the designer. The development of hardware architectures can be better optimized when the tool works together with the designer, not in place of a designer. Higher level languages should be used as a way of defining what the architecture is doing, not how it is doing it.

PACT HDL [6] was developed with the idea of creating power aware architectures. However, the scheduling methodology used does not lend itself to low resource allocation. Additionally, it does not support the use of any other scheduling methodologies aside from a window size, priority-based approach using ASAP and ALAP scheduling. HardwareC [4] suffers from a lack of abstraction, as the input into the program is virtually identical to Verilog, with only a few minor syntactical differences. Additionally, pipelining, a method commonly used in FPGA architectures, is not supported. Handel-C [3] provides one of the strongest emulations of the C languages, but requires the designers to specify parallel portions of code, rather than having the program automatically derive this information. Part of this is due to its lack of support for loop unrolling, a technique commonly used to expose instruction level parallelism. The designer should be able to interact with the program in the creation of the design, but not be responsible for every detail. Mitrion-C [5] appears to be the strongest of the candidate C-to-HDL programs. Rather than generating a schedule for the received code, a network is created that is capable to moving data from one processing element to another. The architecture is then presented with a data packet switching problem, rather than an instruction scheduling problem. Mitrion-C also uses a virtual processor, optimized based upon the code received, to execute instructions. The downside of this approach is that the program raises the level of abstraction so far that it almost prohibits the designer from having any impact on the generated hardware architecture. This problem manifests itself specifically when code relies upon data being entered at run-time. Without the help of the designer, hardware generation tools have no way of determining how often a segment of code may execute, and will therefore have no way of properly allocating resources to handle such code in an optimal fashion. Mitrion-C therefore provides efficient, but not optimal, hardware implementations.

The FREEDOM compiler [7] is a platform independent HDL generator that can use any generated assembly language file, along with an architecture description file, to generate VHDL code. The downside of working directly on assembly language files, as opposed to C files, is the loss of critical information declared in the C language. For instance, the compiler removes name dependencies by eliminating "unnecessary" writes to the same memory location. For any system that makes use of volatile variable, this can be critical. The tool also lacks the ability to customize the hardware design to the needs of the designer, but rather depends on the designer to customize the input appropriately. A tool called Compaan [8] translates algorithms specified in the Matlab language into hardware architectures. At the time this paper was written, no actual FPGA implementation had been created yet, though it had been discussed [10].

Additional relevant work is in the area of Molecular Dynamics (MD), a collection of algorithms used to analyze the interaction of molecules within a closed environment. The driving force behind the development of the tool described in this paper was to verify, and possibly improve, an existing FPGA-based implementation of a portion of a MD application [16]. The use of an FPGA for solving N-body problems, such as MD, was explored by Lienhart et al [11]. Using application specific processors on FGPAs was tested by Azizi et al [12]. Other attempts have been made to separate portions of the calculation, such as moving Lennard-Jones and Coulombic force calculations onto an FPGA, which communicates with the driving software, run on a standard CPU [13]. Further work has also been done in an attempt to speedup time intensive operations, such as double-precision floating-point division, through the use of pipelining [14]. For our approach, we are using an FPGA similar to [10], but with a novel architectural approach discussed in [15] [16].

The novelty of the tool being presented here is not in its ability to generate VHDL and C++ representations, but the method used to determine resource allocation, create a schedule, and provide multiple possible implementations based upon a set of constraints given to the program through the use of a modified FDS algorithm. The tool provides a visual modeling environment to support interaction with the designer, allow for the input of architecture critical specifications and operational unit customization. The development of this tool is divided into three components: GME front-end, FDS implementation and environment creation, and VHDL and C++ code generation, shown in Figure 1. Each of these will now be discussed.

3 GME Front-end

The Generic Modeling Environment (GME) is a meta-programmable modeling framework that may be used to create not only domain-specific models, but

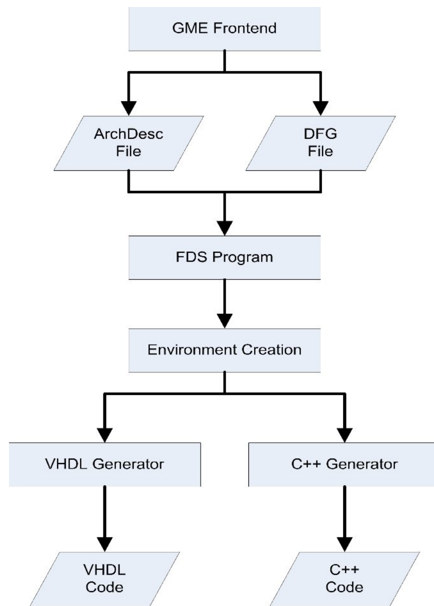

Figure 1 – Overall tool flow.

corresponding domain-specific modeling languages. GME does not provide inherent support for any one specific domain, but rather may be used by the developer to create custom modeling languages and paradigms which encode the visual notations and rules inherent to the domain and known by domain engineers. Paradigms are similar to grammars used by programming languages. Their purpose is to define the rules for how elements within the language may be used. Since GME does not provide native support for any modeling language or paradigm, it must be configured in order for the user to properly capture the relationship between the real-world and the elements defined within the GME model.

Although the GME model may do a very effective job at modeling a given environment, it makes no attempt to determine what the model is supposed to mean. To decipher the meaning of a specific GME model, an interpreter is created. Just as the keyword "double" means nothing without a C or C++ compiler, the same is true of a GME model without an interpreter. The interpreter could therefore be compared with a compiler, in that it is used to make sense out of what is otherwise nothing more than a collection of images.

The "syntax" that the interpreter uses is derived from what is called the meta-model. A meta-model is what is created in GME as the model of a domain specific modeling language. The meta-model implemented for this paper is one that models DFGs and therefore facilitates the creation of nodes, edges, operational units, and architecture specifications. It also allows the scaling to large designs by handling multiple DFGs within a single project. Once a DFG is properly modeled, the interpreter analyzes the model and creates data files that may be used by the force-directed scheduler to determine a schedule for the DFGs. The steps for how this was accomplished will now be covered.

3.1 GME Meta-model

The GME Meta-model, shown in Figure 2, offers a hierarchical structure for developing hardware designs. A Design may contain multiple dataflow graphs, represented by the DataFlowGraph model, and must contain one, and only one, ArchDesc model, used to enter specifications about the target architecture. ArchDesc models contain actual operations, which consist of standard mathematical operations (i.e., addition or subtraction), logic operations (i.e., less than or equal to), and I/O operations (i.e., inputs or outputs). Each operation has an attribute for the area consumption and latency. Currently, such fields must be specified directly by the designer, but future work will integrate a pre-generated library of these values, corresponding to their implementation on specific FPGA chips. An additional atom, called the Chipset, is required and is used to specify the target architecture, as well as constraints to be imposed on the generated hardware architecture.

3.2 GME Interface

The GME meta-model is never actually seen by the designer. Rather, it is used to govern and interpret the design. Using GME, the designer may create a Design, with its associated ArchDesc and DataFlowGraph models. The designer inserts whatever components will be needed for the design into the ArchDesc model, and then creates instances of these customized components in the DataFlowGraph model. Connections may then be instantiated to create precedence relationships between multiple nodes within the DFG.

3.3 Interpreter

Once a design has been created in GME, an interpreter is used to parse the design and generate a DFG file and architecture description file that will be used, by the FDS implementation, to derive the hardware architecture. The architecture description file uses the following syntax:

“OPERATIONS”
<operation type> <operation latency> <operation area>
….
“CONSTRAINTS”
<type of constraint> <constraint value>

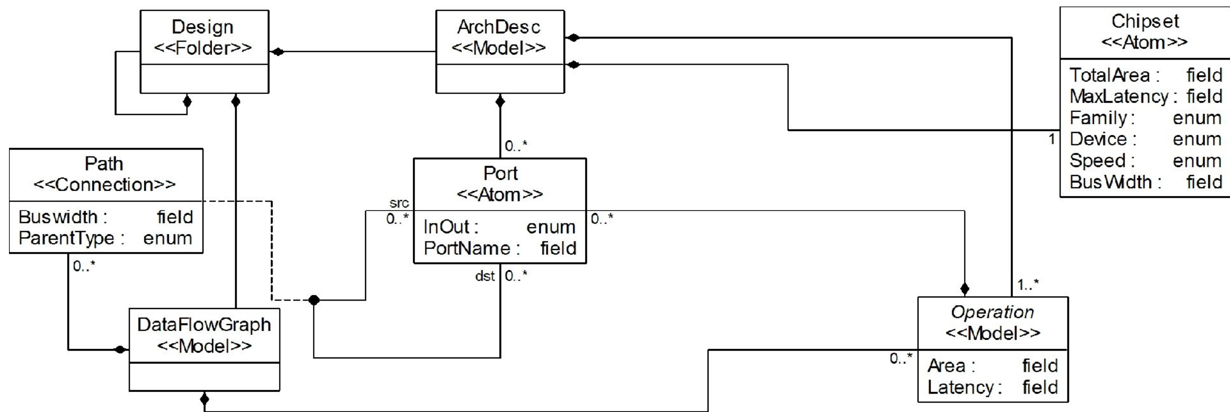

Figure 2 – GME meta-model.

Operation types were listed in section 3.1. As defined by the user in the ArchDesc model, the latency and area of each operational unit is also declared. The constraints currently supported by the tool are listed in Figure 2 under the Chipset atom. The dataflow graph files use this syntax:

//Node declaration
"NODE" <node id number> <operation type>
…..
//Edge declaration
"CONNECTION" <source node ID> <destination node ID> <parent type>
…..

Each operation declared in GME is given a unique ID number which is used to identify the node in the DFG file. Connections are made between two nodes by declaring a connection, and then providing the corresponding node ID numbers for the source and destination. The parent type is used to preserve the distinction in non-commutative operations, like subtraction and division. The parent type can be left, right, or both, to emulate first, second, and both operators, respectively.

The goal of this approach is to provide an interface where the designer can still specify certain constraints on the system which can have a large impact on the overall system design, but still provide an high enough level of abstraction to make the tool intuitive to use. The generated files may then be used by the designer to invoke the FDS scheduler implementation to derive the hardware architecture.

4 FDS Implementation and Environment Creation

The FDS algorithm used in this tool is a modified version of the original algorithm [2]. The original algorithm prioritizes the scheduling of nodes with a DFG based upon their location in regards to the critical path of the system, as well as their impact on resource allocation. This is done by creating a domain of values for each node, representing the possible starting and completion times that each node may be assigned. Nodes along the critical path of the system are scheduled as-soon-as-possible (ASAP), which then further constrains the starting and completion times of other nodes within the system. Nodes not on the critical path are scheduled by using a percentage calculation that determines the effect on the resource requirements of the system by scheduling the nodes at specific times. Since the FDS algorithm attempts to minimize the resource needs of a system, the nodes are scheduled in a manor that accommodates this constraint.

In an attempt to utilize the powerful parallelization capabilities of FPGAs, loop unrolling is used to expose instruction level parallelism. In addition, the ability to perform critical path relaxation is also provided to the designer. Traditional FDS discovers the critical path of the graph, and imposes a requirement that all operations complete at, or before, the completion of all nodes on the critical path. Critical path relaxation facilitates the designation of a constraint which allows operations in the DFG to complete up to a fixed time after the length of the critical path of the graph. Critical path relaxation can result in further reductions to resource requirements when flexibility exists in the execution time requirements of the system. Experimental results showed that the FDS algorithm did not function as well when critical path relaxation was employed. To combat this problem, a new approach was taken to modify the available start and completion times of nodes with the DFG. This new algorithm attempts to balance the extra time allotted by critical path relaxation among the multiple number of iterations instantiated through loop unrolling. Additionally, it

$let\ pred : N \rightarrow P\{N\}$ be the set of all predecessor nodes
$let\ succ : N \rightarrow P\{N\}$ be the set of all successor nodes
$let\ iter : N \rightarrow \mathbb{R}^{+}$ be the iteration value of a node
$let\ upper : N \rightarrow \mathbb{R}^{+}$ be the upper bound of a node
$let\ lower : N \rightarrow \mathbb{R}^{+}$ be the lower bound of a node

$\forall\ n \in N :$

if pred(n) == NULL

$lower(n) = 0 + \frac{iter(n)}{\#ofiterations} \times$
$(cp_length_with_relaxation - cp_length_w/o_relaxation)$

else

$lower(n) = \min_{p \in pred(n)} st(p) + lat(p)$

if succ(n) == NULL

$upper(n) = critical_path_length_without_relaxation - lat(n) +$
$\frac{iter(n)}{\#ofiterations} \times (cp_length_with_relaxation - cp_length_w/o_relaxation)$

else

$upper(n) = \max_{p \in pred(n)} st(p) + lat(p)$

Figure 3 – Modified FDS algorithm.

reduces the amount of time needed to derive an implementation using the FDS algorithm by reducing the number of operations that must be considered at each time step. To implement this, the following algorithm is used, shown in Figure 3. Based up the iteration each node is associated with, and the total number of iterations being implemented, a certain amount is added to the earliest start time and latest completion time, proportional to the amount of additional time allocated through the use of critical path relaxation. The results of this implementation will be discussed in section 6.

The environment is the collection of all values and variables needed in order to generate a VHDL and C++ representation of the derived hardware architecture. To create the environment, a collection of variables are instantiated and populated using the results of the FDS implementation. This includes a list of all nodes in the DFG, as well as all required operational units and the number of each that is required to execute the generated schedule. To manage this information, the environment is created and used to create the VHDL and C++ code.

An important step to creating the environment is to derive the required memory needs within the DFG. Memory requirements within a DFG are the result of a difference in the time when values are produced and consumed. To determine where memory elements would be needed, a simple algorithm was developed that compares the start time, plus latency, of each node with the start time of each of its predecessors. Any time these two values are not equal, a memory element is instantiated. To support the reuse of memory elements, shift registers are used to store the data. This allows for an optimization in the number of elements required by allowing reuse of memory elements within a single iteration, as well as between multiple iterations, when identical memory needs are present. Whenever identical memory needs exist, as measured by the depth of the shift register, if the needs occur on separate clock cycles, the memory element may be reused. Additional optimizations are possible, but are left as future work and will be discussed in section 7.

The final step in creating the environment is to develop an ordered list of nodes based upon their scheduled execution time. The purpose of such a list is to reduce the computational complexity of the VHDL and C++ code generation. By pre-sorting the nodes in the DFG, the algorithm must execute only on clock cycles where changes are needed. Otherwise, the computational complexity of analyzing what changes need to occur on what clock cycle is $O(mn)$, where m represents the number of clock cycles in the derived schedule, and n represents the number of nodes. With the ordered list, the complexity reduces to $O(n)$, which can results in a large performance gain, especially considering this algorithm must execute during both the VHDL and the C++ code generation.

The implementation of the modified FDS algorithm and environment provide the designer with another opportunity to customize the design by specifying the number of loop unrolls to perform, as well as the amount of critical path relaxation to perform. It also may derive architectures with a much smaller footprint, as opposed to traditional implementations of the FDS algorithm. The memory allocation method also removes the need for the designer to determine such needs *a priori*.

5 VHDL and C++ Code Generation

Once the environment has been properly established, the program may then invoke the VHDL and C++ code generators. The VHDL code may be used to develop hardware architectures in Xilinx ISE. The purpose of the C++ code is to provide the designer with a simulator of the derived hardware

architecture. This allows the designer to see the results of modifications to the design without the needed for time intensive synthesis and place and route routines. The C++ simulation also allows the designer to determine the number of clock cycles required to complete execution of the derived architecture. In this way, the designer is given fast feedback on how modification to the architecture will affect its results.

5.1 VHDL Code Generation

VHDL code generation entails the process required to generate not only VHDL files, but also all files that will be needed in order to instantiate Xilinx IP Coregen components. To create a VHDL representation of the derived hardware architecture specific to Xilinx ISE, a main controller file is needed. In addition, .xco and .vhd files are generated an instance of each operational unit using Xilinx IP Coregen. By using Coregen to generate operational components, designers are provided with a maximum amount of flexibility in how the component is instantiated, i.e. latency, pipeline stages, and memory initialization.

In addition to creating Coregen files, but prior to the creation of the main controller file, a node to operation mapping must be made. Each node in the DFG must be associated with a specific operation unit before the main controller file can be developed. The mapping is performed by using a simple greedy algorithm. This algorithm works by examining each node in the system against the available execution slots of the operational units. If an operational unit has already been allocated to execute a node on a given cycle, it is unavailable. If the operational unit is not already allocated to another node, a mapping is created and that operational unit's status for that clock cycle is set appropriately. Since all functional units are assumed to be fully pipelined, the execution of an operation effectively consumes only one issue slot for each unit. The scheduler guarantees that at least one operational unit will be available for every node in the DFG at the required time.

The main controller file is the heart of the VHDL generated code. Contained within the controller are the initializations of the required operation units, instantiations of signal to connect the operational units to the main controller, and a case statement to determine the flow of information through the system. The case statement makes use of the ordered list created during the environment generation. The nodes in this list are traversed and a case is declared for each unique time in which a node is scheduled. Starting at the beginning, a new state is declared for the starting time of the first node, and a local variable designated to track the time is set equal to this value. Once all data has been properly channeled, the program moves to the next node and compares its starting time with the current time value. If they are the same, a new state is not needed and the data flow statements to mange this node's data exchanges are made within the current state. Otherwise, a new state is created, the current time is set equal to this node's start time, and the new data flow statements are made in this new case. This continues until all nodes have been visited.

The result of this process is a complete set of files that may be used by Xilinx to synthesize the derived architecture. The mapping, translate, and place and route routines require a .ngc file for each IP Coregen component, which is currently not supported, but is included in the list of future work.

5.2 C++ Code Generation

The C++ code generation handles the creation of a group of classes that may be used to simulate the derived hardware architecture. The initial code representation, whether done in C, Matlab, or assembly language, merely shows what operation the hardware is to perform. The purpose of the C++ representation is to show how the derived architecture works. These classes emulate the hardware generated previously in VHDL, and allow the designer to test the architecture in a faster manner than through synthesis of the hardware. For instance, a primary concern of designer may be how quickly the derived architecture can complete the required calculations. The generated C++ simulator may be used to quickly deliver such information, as well as to determine performance metrics and the locations of bottlenecks within the design.

The C++ representation is made using two classes, the Element class and the Component class. The Element class is used to represent each of the different operational units that are currently supported, containing information on the latency and operation type of each unit. The Element class is written using a template, allowing it to be used for integer or pointing-point representations. The Component class is similar to the main controller created in the VHDL representation. It contains functions for initializing and executing the design, as well as retrieving the current status of the system. These functions are automatically populated with the required instances of Elements, as well as a switch statement that works identically to the case statement in VHDL. The designer is only required to call the initialize function and execute function to simulate the design. Functionality is provided to see the current state of the system, but currently only returns the local time. Additional changes are planned to have this function display the current information

contained within each stage of the unit's pipeline, as well as data being stored or currently on connections. This information would be similar to a register dump on a standard microprocessor.

The goal of this portion of the program was to provide the designer with a quick and easy way to verify his/her design, as well as to implement changes and see how they affect the design. By using C++ to create this simulator, these goals have been met. The designer may use any standard C++ compiler that supports the use of the STL to simulate their design. Changes in the execution of these units, such as pipeline stages, are easy to make. Rather than having to re-synthesize the design, the designer merely has to recompile the code to see how such changes can affect the system.

6 Results

Several key new ideas were presented in this paper. This section has been developed to show the results of the implementation of these features. Results will be presented on the modified FDS algorithm, effects of critical path relaxation, and the results of deriving a micro-architecture for a previous MD implementation.

6.1 Modified FDS Algorithm Using Graph Decomposition

To show the results of the graph decomposition and modified FDS algorithm verses the original FDS algorithm, implementations of the distance calculation (DC) and Lennard-Jones potential calculation (LJPC) were developed. The results of these implementations using the original and modified FDS algorithms are shown in Figures 4 and 5, respectively. These figures show the resulting resource allocation for floating-point adders, subtractors, and multipliers, using from 1 to 57 iterations of the DC unit and 1 to 40 iterations of the LJPC unit. The required units using the modified, graph decomposition method are labeled with a (gd) next to the unit name. As shown in the DC implementation, the modified version derived architectures needing only half the number as the original, on average. The same is true of the LJPC implementation.

The other strong support for this implementation was in the reduction of time required to derive these architectures, using the two approaches. Figure 6 shows the time required to derive each architecture, as done on a Pentium 4, 2.0GHz computer. Each version of the algorithm generated an architecture for the LJPC unit, starting with a single iteration, up to a total of 40 iterations. The time taken to derive each architecture was measured and is presented in this figure.

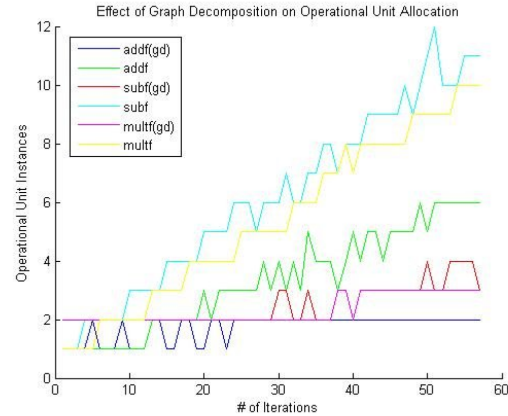

Figure 4 – Operational unit allocation for DC unit with 100% CPR.

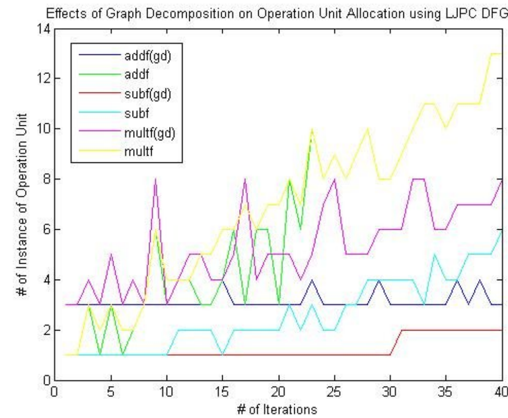

Figure 5 – Operational unit allocation for LJCP unit with 100% CPR.

6.2 Critical Path Relaxation

The purpose of critical path relaxation (CPR) is to show the possible changes in resource requirements by relaxing the critical path by some measure of time, or cycles. The results of this are shown in Figure 7. This figure shows the change in resource requirements as the critical path is relaxed, starting at 0% relaxation to 100% relaxation, by increments of 10%, using 57 iterations of the DC unit. As can be seen in this graph, a CPR of only 10% resulted in a decrease of 75% of required resources, verses no relaxation. As may also be evident, at certain times plateaus are reached where additional, but not substantial, relaxation may not create increased benefit in reduction of resource

requirements. This type of information, together with

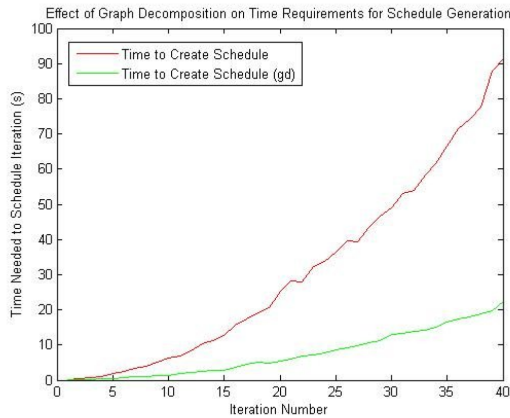

Figure 6 – Time comparison of standard vs. modified FDS algorithms.

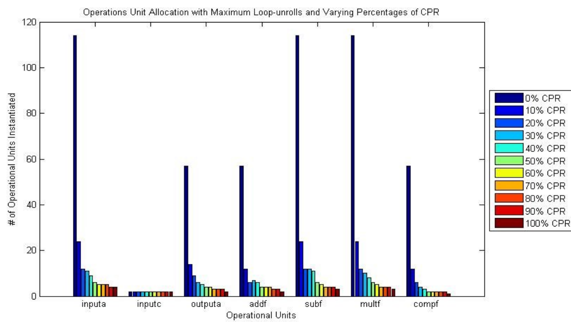

Figure 7 – Comparison of resource requirements for 57 iterations of the DC unit and varying CPR.

throughput, utilization, and latency constraints, can greatly affect the automated generation of hardware architectures.

6.3 MD Architecture Generation

The MD architecture referenced previously was implemented in an attempt to produce run-time load-balancing of an N-body problem. The result of this architecture's ability to perform such load-balancing is shown in Figure 8. The purpose of this tool was to validate the design initially created for this project. The two main components of the MD algorithm, the DC and LJPC, have been run through this tool and generated the results shown in Figures 4 through 7. From these results, analysis has been performed to determine where the optimal solution lies. Due to the load-balancing problem faced by N-body problems, generating the entire architecture autonomously is a very taxing problem, and currently beyond the capabilities of this tool. However, it has verified our initial design as being effective, and efficient, but took only a very small fraction of the time.

In addition to verifying the generated VHDL, we were able to use the generated C++ code to analyze the performance of this architecture. We were quickly able to determine the number of clock cycles required to complete a given number of calculations, as well as to generate information on the current state of internal elements within the design. The values, shown in Figure 6, were generated through the use of the C++ simulator. Although this feature has not yet been completely integrated into the C++ simulator at this time, it has proven to be accurate and is currently in the process of being included.

7 Conclusion

We have presented a new and novel methodology for the automated development of hardware architectures. We have used this tool to verify our initial design and have found it to be extremely successful at generating verifiably functional code in both VHLD and C++ in a fraction of the time. A large amount of future work remains to be done in order to better customize the hardware design. GME provides an excellent front- end that greatly facilitates the incorporation of design operations and constraint implementation. Additional tools, such as Mozart, are also being added to provide better constraint satisfaction and optimization for those wishing to customize architectures in the areas of latency, power, area, and throughput.

7.1 Additional Work

Additional work still needs to be performed in each other three major areas discussed in this paper. Much of the work that still remains has been discussed previously in this paper. Expansion of supported elements in GME, an automated DFG creation tool, and further constraint satisfaction analysis are the three primary areas being explored at this time. Given the results seen so far, we are confident that the improvements being made will help this tool to become an excellent resource for any hardware designer.

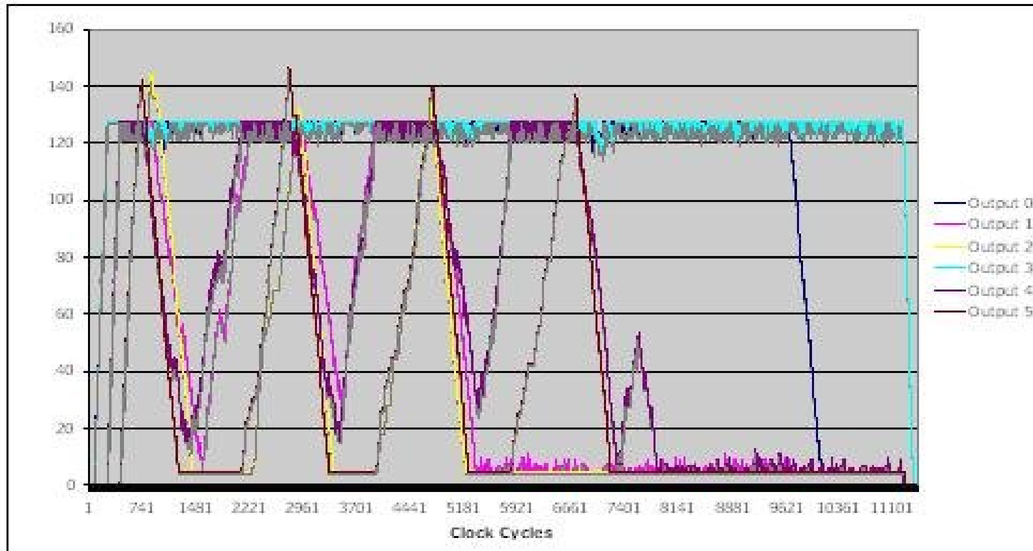

Figure 8 – Amount of output data waiting to be processed by LJPC units within the MD architecture.

[1] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, and P. Volgyesi; "The generic modeling environment"; Vanderbilt University, Nashville, Tennessee, USA.

[2] Paulin, P.G. and Knight, J.P.; "Force-directed Scheduling for the Behavioral Synthesis of ASIC's"; Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, Vol.8, Iss.6, Jun 1989, Pages:661-679.

[3] C. Inc., \Handel c compiler," http://www.celoxica.com.

[4] D. Ku and G. DeMicheli; "Hardware C - a language for hardware design (version 2.0)"; Stanford, CA, USA, Tech. Rep., 1990.

[5] I. Anders Dellson, Mitrionics; "Programming FPGAs for high-performance computing acceleration"; http://www.mitrionics.com.

[6] Jones, A.; Bagchi, D.; Pal, S.; Tang, X.; Choudhary, A.; Banerjee, P.; "PACT HDL: a C Compiler Targeting ASICs and FPGAs with Power and Performance Optimizations"; *Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems,* 2002, pp. 188-197.

[7] Zaretsky, D.; Mittal, M.; Xiaoyong Tang; Banerjee, P.; "Overview of the FREEDOM compiler for mapping DSP software to FPGAs" in *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on*, Vol., Iss., 20-23 April 2004 Pages: 37- 46**.**

[8] B. Kienhuis, E. Rijpkema, and E. Deprettere, "Compaan: deriving process networks from matlab for embedded signal processing architectures"; *CODES '00: Proceedings of the eighth international workshop on Hardware/software codesign*, pp. 13-17. New York, NY, USA: ACM Press, 2000.

[9] Reyneri, L.M.; Cucinotta, F.; Serra, A.; Lavagno, L.; "A Hardware/Software Co-design Flow and IP Library Based on Simulink"; *Proceedings of the 38th Conference on Design Automation*, 2001, pp. 593-598.

[10] E. Rijpkema, E. F. Dprettere, and B. Kienhuis; "Compilation from matlab to process networks"; *Signals, Systems, and Computers, 2001. Conference record of the thirty-fifth asilomar conference on*, pp. 458-462. Washington, DC, USA: IEEE, 2001.

[11] G. Lienhart, A. Kugel, and R. Manner, "Using floating-point arithmetic on FPGAs to accelerate scientific N-Body simulations," in *Field-Programmable Custom Computing Machines. Proceedings. 10th Annual IEEE Symposium on*, 2002, pp. 182-191.

[12] N. Azizi, I. Kuon, A. Egier, A. Darabiha, and P. Chow, "Reconfigurable molecular dynamics simulator," Proc. *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. 2004, pp. 197-206.

[13] R. Scrofano and V. K. Prasanna, "Computing Lennard-Jones Potentials and Forces with Reconfigurable Hardware", Proc. *International Conference on Engineering of Reconfigurable Systems and Algorithms* 2004, pp.284-292.

[14] Y. Gu, T. VanCourt, and M. C. Herbordt, "Accelerating molecular dynamics simulations with configurable circuits," *Computers and Digital Techniques, IEE Proceedings-,* vol. 153, pp. 189-195, 2006.

[15] Phillips, J.; Areno, M.; Eames, B.; Dasu, A.; "An FPGA-Based Dynamic Load-Balancing Processor Architecture for Solving N-body Problems"; *Proceedings of the 10th Annual High Performance Embedded Computing Workshop, 2006.*

[16] Phillips, J; Areno, M.; Rogers, C.; Eames, B.; Dasu, A.; "A Reconfigurable Load-Balancing Architecture for Molecular Dynamics"; *Proceedings of the 14th Annual Reconfigurable Architecture Workshop, 2007.*