JACOBI LOAD FLOW ACCELERATOR USING FPGA[1]

J. Foertsch, J. Johnson, P. Nagvajara
Drexel University
Philadelphia, PA

Abstract – Full-AC load flow is a crucial task in power system analysis. Solving full-AC load flow utilizes iterative numerical methods such as Jacobi, Gauss-Seidel or Newton-Raphson. Newton-Raphson is currently the preferred solver used in industrial applications such as Power World and PSS/E due to it faster convergence than either Jacobi or Gauss-Seidel. In this paper, we reexamine the Jacobi method for use in a fully pipelined hardware implementation using a Field Programmable Gate Array (FPGA) as an alternative to Newton-Raphson. Using benchmark data from representative power systems, we compare the operation counts of Newton-Raphson software to the proposed Jacobi FPGA hardware. Our studies show that Jacobi method implemented in an FPGA for a sufficiently large power system has the potential to be a state of the art full-AC load flow engine.

Keywords: FPGA, Gauss-Seidel, iterative solver, Jacobi, load flow, Newton-Raphson

1 INTRODUCTION

Load flow calculations are the basis from which most power system analysis stems. The results of a load flow calculation are used to estimate the operation of a power system under a set of known conditions. Obviously systems are not static and any one of the many input conditions can change. The ability to quickly perform the load flow calculation allows engineers to be more confident in the safety, reliability, and economic operation of their system in the event of scheduled or unscheduled equipment outages.

Current technology does not adequately allow for real-time or dynamic analysis of these systems. The complexity and pure number of calculations for a given problem can take on the order of hours to complete. Much research has been done in this area to speed up calculations [1]. These studies can be summarized into three general approaches: reducing calculation by improving the algorithms; improving software efficiency by means of highly optimized libraries of routines and/or a cluster of machines running the software program concurrently; and adding parallelism in the calculation by means of custom hardware assistance. Each approach has built upon the previous research: the hardware approach leveraged the concurrent software research and the software approaches leveraged the algorithmic research.

In this paper we take the latest approach – custom hardware – and evaluate its affectivity in the light of the older, previously dismissed algorithms. In particular, we propose a deeply pipelined custom hardware to implement the Jacobi iterative load flow solver. This method is compared against the state-of-the-art software and hardware solutions.

2 BACKGROUND

The purpose of a load flow computation is to determine numerically all the voltage magnitudes and phase angles at load buses, voltage phase angle and reactive power at generator buses, and real and reactive power at the slack bus of a power system transmission network.

Popular methods used when performing full-AC load flow (or power flow) are Jacobi, Gauss-Seidel and Newton-Raphson [2]. Each one has its own set of benefits and drawbacks. The Jacobi and Gauss-Seidel methods are the easiest to understand because they use the known quantities of the power system. This makes for simple implementation, but the process takes significantly more iterations to converge to a solution. Jacobi takes more iterations than Gauss-Seidel. Its updates to the solution are applied each iteration rather than within the same iteration. The Newton-Raphson method is much more complex than either the Jacobi or Gauss-Seidel methods. The total number of calculations per iteration is increased. However, it takes less iteration to converge. Newton-Raphson method has a quadratic rate of convergence; Jacobi and Gauss-Seidel have a linear rate of convergence.

2.1 Gauss-Seidel and Jacobi Methods

Gauss-Seidel and Jacobi methods are very similar. They take the same input data, and produce the same outputs. The main difference is how newly calculated voltage data is handled in subsequent calculations. . The Jacobi method gets a solution vector for a constant set of inputs. Gauss-Seidel does not require a constant set of inputs; it uses the most readily available value. Load flow solution via Gauss-Seidel method involves iterating (1) and using (2):

$$V_i^{(\nu+1)} = \frac{1}{Y_{ii}}\left[\left(\frac{S_i}{V_i^{\nu}}\right)^* - \sum_{k=1}^{k<i} Y_{ik} V_k^{\nu+1} - \sum_{k=i+1}^{NZ} Y_{ik} V_k^{\nu}\right] \quad (1)$$

where:

$$S_i = P_i + jQ_i \quad (2)$$

until the maximum absolute change in each element of the resultant voltage vector, V, between iterations is less than 10^{-4} per unit. Where Y is the bus admittance matrix (Y-bus); P is a vector of real power injections; and Q is

[1] The United States Department of Energy (DOE) Grant No CH11171 supported the research reported in this paper.

0-7803-9255-8/05/$20.00 2005 IEEE

a vector of reactive power injections. An injection is the difference between generation and demand. All elements of the V vector and the Y matrix are complex quantities. For generator busses, the voltage magnitude is already known. Reactive power is first estimated for the Gauss-Seidel iteration by (3).

$$Q_i^{\upsilon} = V_i^{\upsilon} \sum_{k=1}^{NZ} \left(Y_{ik} V_k^{\upsilon}\right)^* \quad (3)$$

Reactive power becomes an output of the calculation when the voltage magnitudes and angles converge to a solution.

Load flow solution via the Jacobi method simplifies equation (1) by combining the two summations into a single summation from 1 to the number of non-zeros (NZ) omitting the ith entry. In other words, a snapshot of the input voltage vector is used in all N-1 calculations until a completely new voltage vector is determined.

2.2 Newton-Raphson Method

Load flow solution via Newton-Raphson method involves iterating (4) until f(x) = 0 is satisfied.

$$-J \cdot \Delta x = f(x) \quad (4)$$

The Jacobian, J, of the power system, is a matrix of partial derivatives of the real and imaginary power equations with respect to voltage magnitude and phase angle. This matrix is sparse in a similar fashion as the Y-bus matrix. Delta x is a vector of the change in the voltage magnitude and phase angle for the iteration in progress. And f(x) is a vector representing the real and imaginary power mismatch.

Newton-Raphson is the most widely used load-flow calculation engine due to its algorithmic advantages over Gauss-Seidel and Jacobi.

2.3 FPGA Technology

Field programmable gate arrays are reusable logic devices. Arrays of logical blocks containing logic gates, combined with dedicated functional and memory blocks, and interconnection wires form the underlying flexible fabric for FPGA integrated circuits. Software codes, typically written in a hardware description language (HDL) such as VHDL or Verilog, are synthesized and mapped to these devices allowing a designer to specify the functionality of the FPGA.

In 2002, FPGA device density was in the thousands of logic gates with operational clock rates in the tens of Megahertz. Today, device densities are in the millions of logic gates with synthesized logic capable of running at rates up to and exceeding 500 MHz (see www.xilinx.com). Beyond increases in logic density, the addition of high performance embedded arithmetic units, large amounts of embedded memory, high speed embedded processor cores, and high speed I/O has allowed FPGA integrated circuits to grow beyond just simple prototyping devices. FPGA architectures have evolved to the point where high performance floating-point computation is now feasible; and with additional hardware devoted to floating point computation, logic designs targeting to an FPGA can outperform high-end personal computers [3].

3 BENCHMARK SYSTEMS

A particular phenomenon that makes the full-AC load flow problem even more interesting is the sparse nature of the network interconnect matrix. This matrix, called the bus admittance matrix or Y-bus, defines the interconnection of all the generation and loads at specific points in a power system. This matrix is extremely sparse since entries are only significant (non-zero) if a branch between two buses exists. In our benchmark systems provided by PSS/E and PJM Interconnection shown in Table I, a typical bus contains only four branches; a typical power system can easily contain a matrix of tens of thousands of buses. Thus making the number of non-zero entries in the Y-bus matrix of a large system (>1000 buses) less than 1%.

System	Max	Min	Avg	Total
118 Bus	13	2	4	490
300 Bus	13	2	3	1,118
1648 Bus	24	2	4	6,680
7917 Bus	16	2	4	32,211

Table I: Y-bus Elements per Row

Highly optimized dense linear algebra packages, such as Binary Linear Algebra Subroutines (BLAS), struggle with the sparse nature of the load flow computation. Some software packages implemented on clusters of workstations has been designed attempt to take advantage of the sparse nature of this problem with some degree of success. These computing engines are usually based on general-purpose microprocessors, such as the Intel Pentium or Motorola PowerPC with supporting floating-point units and hierarchies of cache memory. When performing sparse matrix operations, these systems fail to fully utilize available floating-point resources resulting in processor stalls. Software compilers often fail to optimally order arithmetic operations.

The thee most popular software approaches to load flow are summarized in Table II. It can be seen that although the Jacobi and Gauss-Seidel methods have the same number of floating-point operations per iterate, the Jacobi method has more iterations to acheive a solution. It can also be seen that the number of operations for the Newton-Raphson per iteration is greater than both Gauss-Seidel and Jacobi, but has significantly less iterations and fewer total operations to converge.

This simple analysis compares floating-point operations between the methods. Assuming that the data access rate of the sparse matrixes used in each calculation will roughly be the same between the methods, it shows that a sequential implementation (in software) of the Newton-Raphson method will significantly out perform the others. This is due to Newton-Raphson having a quadratic rate of

convergence compared to a linear rate for both the Gauss-Seidel and Jacobi methods.

Bus	Ops/Iteration	Iterations	Total Ops
Jacobi Method			
118	9,404	717	6,742,668
300	17,674	2207	39,006,518
1648	106,452	2031	216,204,012
7917	515,531	2183	1,125,404,173
Gauss-Seidel Method			
118	9,404	477	4,485,708
300	17,674	2130	37,645,620
1648	106,452	1683	179,158,716
7917	515,531	1845	951,154,695
Newton-Raphson Method			
118	5,366	4	21,464
300	40,621	4	162,485
1648	625,564	5	3,127,818
7917	4,062,382	6	24,374,294

Table II: Floating-Point Operations by Method

Analyzing each of the methods, it is perceptible that the Jacobi method has the most available parallelism. From a high level it can be seen that each voltage calculation is independent; multiple rows can be calculated concurrently. Neither Gauss-Seidel nor Newton-Raphson has this property. Within each voltage calculation, some of the intermediate calculations are also independent and can be computed in parallel. Gauss-Seidel has the same capability, but Newton-Raphson does not. Also, many of the operations containing complex numbers in each of the simple operations can be done with a high degree of parallelism. Gauss-Seidel could also benefit from this. Newton-Raphson does not operate directly on complex numbers.

With all this available parallelism, a deeply pipelined hardware design is attainable. This design and projected performance will be discussed in the following sections.

4 HARDWARE ARCHITECTURE

The proposed hardware design uses all facets of a high-end FPGA like the Altera Stratix (see www.altera.com): embedded dual-port memories, numerous densely packed logic elements, high-speed external memory interfaces, and input/output pins. These building blocks are efficiently used to create the overall pipeline design. Within the pipeline, multiple floating-point cores, written in VHDL, are instantiated and chained together. These cores combined with control logic, input/output, and memory interfaces make up the complete design. The intent of the design is to be simple, portable, and relatively scalable.

Since the operations per iteration are identical, the FPGA hardware design can implement either the Jacobi or Gauss-Seidel methods. The only difference between the two is the scheduling at the head of the pipeline. The Gauss-Seidel method may more than likely stall the pipeline since the voltage to be calculated usually needs a result of a voltage currently inside the pipeline. The Jacobi method has little likelihood of stalling the pipeline. It can even be piped over adjacent iterations if the values at each end of the voltage vector do not dependent on each other. The number of dependencies is proportional to the length of the pipeline. For large systems, it can be easily shown that no stalls would occur.

4.1 Floating-Point Hardware

In order to perform any of these methods in an FPGA, efficient floating-point hardware is required. The ideal design would employ a random access latency of one cycle. Unfortunately this is not a feasible requirement. It is possible to synthesize such a device, but the resulting clock speed would be prohibitive. To account for this deficiency, pipelining is employed. Floating-point units with relatively deep pipelines but high throughput suffice. Our design utilizes three fully pipelined floating-point cores: multiplication, addition/subtraction, and division. The number of pipeline stages differs between the computational cores as seen in Table III. In the final implementation they will all run at the slowest synthesized frequency. If the final synthesis frequency fails to meet our expectations, more stages to the pipeline will be added to improve the overall throughput. This lengthening of the pipeline will degrade performance, but only by a handful of cycles that are insignificant to the overall computation.

Core	Pipeline Stages
Multiplier	5
Addition/Subtraction	4
Division	13

Table III: Synthesis Results on Altera Stratix

4.2 FPGA Pipeline Overview

The proposed FPGA calculation pipeline is designed to accommodate either the Jacobi or Gauss-Seidel full-AC load flow methods. In either case a string of multiplication, addition, subtraction, and division floating-point cores are strategically wired together to form the pipe. The strategy is derived from the work presented in [4],[5]. Each voltage element of the solution vector is calculated independently in row order. All calculations proceed through the pipe unadulterated

From benchmark systems provided by PSS/E and PJM, we found that the average number of elements in the Y-bus matrix tended to be between 3 and 4 regardless of the size of the system. By taking the sizes of the systems into account, one can easily see that an overwhelming majority of the buses in the network

connect to less than 4 neighbors. The number of floating-point cores in the pipeline was chosen to best facilitate these findings.

Fine-grained parallelism can also be exploited since all the operands in this calculation are complex numbers; the real and imaginary portions of the calculation can be computed independently. This is a substantial computation time saver; essentially the number of software operations is halved.

The entire pipeline is pre-configured with known power system data. The bus admittance matrix, real power for all buses, and reactive power for load buses are known prior to the calculation. This presents a limitation for infinitely large systems, but the on-board memory of a high-end FPGA is adequately large enough to handle systems on the neighborhood of 100,000 buses.

Implementation of this design is targeted for an add-in PCI-X card to a personal computer or workstation. The FPGA devices are laid out in a ring topology with respect to each other and in a star topology with respect to the host computer. The host connects to each FPGA with a 64-bit, 125 MHz interface. Each FPGA is connected to its neighbor with a 128-bit unidirectional data bus that operates at speed of the worst-case synthesis frequency.

This design is broken into five main parts: the pipe management unit, Q-estimator/inner product unit, S/V unit, Y-bus scaling unit, and the back-end processor. A block diagram of the implementation is shown in Figure1.

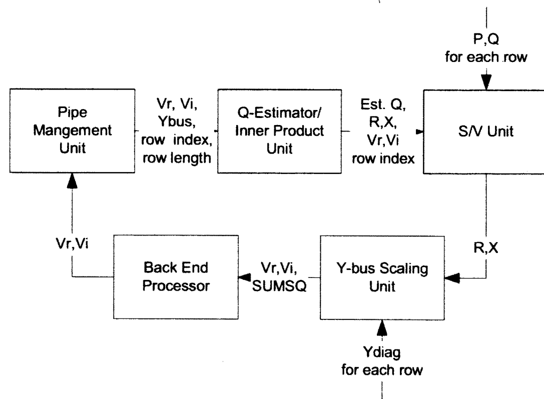

Fig. 1: Hardware Block Diagram

The first part is the head of the pipe management unit is implemented in software. It is assumed that the clock rate of the PC is significantly higher than the FPGA hardware allowing it to adequately keep the FPGA busy with useful work. Data is sent to the FPGA hardware pipeline through Direct Memory Access (DMA). The head FPGA in the pipe has a queue of data structures ready to be processed.

This management unit is designed to know what calculations are currently in the pipeline. If the Gauss-Seidel method is being used, the pipe management unit stalls the pipeline when a needed result is currently being calculated in the pipe. This means a data dependency between two (or more) calculations occurred. In the Jacobi method, there are no data dependencies by the nature of the algorithm. In this case the pipe is not stalled.

Fig. 1, 2, and 3 pictorially depict the pipeline implemented in the FPGA hardware. The circles represent elementary floating-point units: multiplication (x), addition or Subtraction (+), and division (/). Circles on the same horizontal compute concurrently. The arrows indicate the data flow through the pipeline. Resultant data from Fig. 1 is passed to Fig. 2; results from there are passed to Fig. 3.

The second part of the overall pipeline and the first part of the FPGA pipeline is the Q-estimator and/or inner product unit. This portion of the pipeline is shown in Fig. 1. Queued data (Y-bus line impedance values G and B, corresponding voltages V, row index, bus type indication, and packet size) is read by the hardware when marked ready by the host. For generator bus types, the reactive power injection is unknown and must be estimated. This calculation is very similar to the inner product required by all bus types. The inner product is calculated in parallel to the Q estimation. The estimated Q and the inner product values are passed down the pipeline to the S/V unit (fig. 2). For load buses, Q is already known; the inner product still needs to be calculated. The downstream S/V unit ignores the superfluously calculated Q value in this case. In addition to the calculated values, the row index, the complex voltage previously calculated for this row, and bus type indicator is also passed down the pipe.

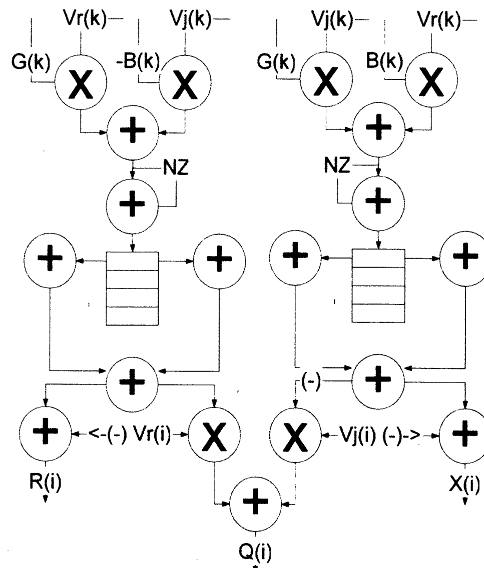

Fig. 2: Q Estimation / Inner Product Unit

The S/V unit performs the complex division of the real and reactive power injections by the previously calculated complex voltage for this indexed element. This portion of the pipeline is shown in Fig. 2. The real power for this calculation is pre-configured and resident

in the FPGA at this stage. It is keyed by the passed row index. After the division completes, the passed in inner product complex value is subtracted from the result. The resulting complex number along with the row index is passed to the Y-bus scaling unit to complete the process.

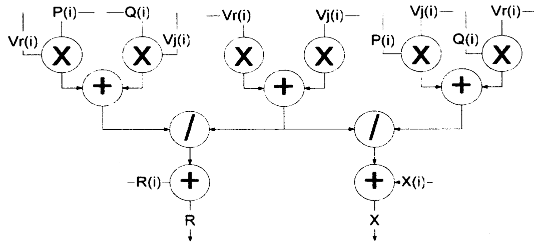

Fig. 3: S/V Unit

The Y-bus scaling unit keys up the pre-configured diagonal entries of the Y-bus using the row index upon reception of the result from the S/V unit. This portion of the pipeline is shown in Fig. 3. The required complex division is performed. The final result is queued to memory along with the original row index and the sum-of-squares of the result. This queue entry is marked ready for the host computer to retrieve and process. The sum-of-squares is included to help the host maintain the voltage magnitude requirement of generator buses. This value is not recorded for load buses.

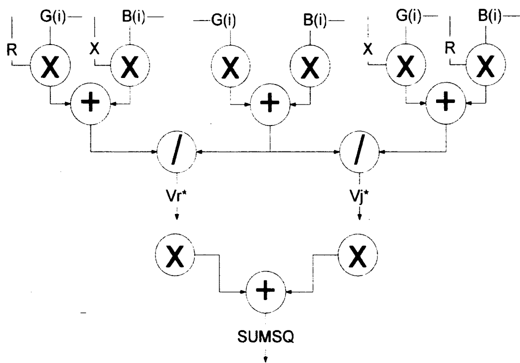

Fig. 4: Y-bus Scaling Unit

The final stage of the calculation is the back-end processor. This process is handled through software. If the Gauss-Seidel method is being performed, the pipeline is likely stalled. The head of the pipe management unit waits for the back-end processor to complete before issuing the next row to the pipeline. If the Jacobi method is being performed, the back-end processor executes after the pipe management unit completes. The back-end processor is responsible for maintaining voltage magnitudes for generator bus types and checking the results for convergence.

4.3 Newton-Raphson Hardware

Johnson and Vachranukunkiet have proposed an FPGA LU solver for use in the Newton-Raphson method [6]. They claim a speedup of greater than 2x with one update unit and upwards of 10x with only four update units with a synthesized FPGA clock of 200 MHz. This speedup applies to the LU factorizations necessary to solve one iterate of Newton-Raphson, which they note takes 85% of the complete iterate. Their proposed design utilizes fewer FPGA resources then the proposed Gauss-Jacobi pipeline, but assumes a much greater synthesized FPGA clock rate for performance.

5 PERFORMANCE

Performance of the Jacobi pipeline can be estimated by adding up all the latencies for each of the floating-point units. This assumes that the pipe can be kept full at all times. This is not always possible. Additional cycles are added to the pipe length to account for bubbles introduced at the head of the pipe due to the PCI bus interface being slower than the pipeline, in the Q-estimation/inner product unit when the density of the row being calculated is greater than four elements, and any data dependencies between rows. The first two cycle adders are apparent in both the Jacobi and Gauss-Seidel pipelines. The third is only evident in the Gauss-Seidel pipeline. Table IV compares the cycle counts for the Jacobi and Gauss-Seidel Methods. Data in the 'First' column indicates the first iteration. This includes the initial pipe latency. Data in the 'Others' column indicates the bubbles incurred each subsequent iteration. The total cycles are the first iteration cycles plus the number of iterations minus one times the remaining iterations.

Jacobi				
	Iteration Cycles			**Total**
Bus	**First**	**Others**	**Total**	**Cycles**
118	890	567	717	406,862
300	1945	1250	2207	2,759,445
1648	11223	7828	2031	15,902,063
7917	55649	39707	2183	86,696,323
Gauss-Seidel				
118	890	890	477	424,530
300	1945	1945	2130	4,142,850
1648	11223	11223	1683	18,888,309
7917	55649	55649	1845	102,672,405

Table IV: Estimated HW Pipeline Cycle Counts

Combining the total operations to perform the Jacobi method with the total number of cycles to do the same calculation results an operations per second measure. This measure is detailed in Table V. The Jacobi FPGA hardware runs at an average of 1.4 Giga-Operations per

second over our benchmark systems when synthesized to 125 MHz.

Bus	Total Cycles	Total Ops	Mega-Ops/sec
118	406,862	4,485,708	1378.14
300	2,759,445	37,645,620	1705.31
1648	15,902,063	179,158,716	1408.30
7917	86,696,323	951,154,695	1371.39

Table V: Jacobi HW Performance @ 125MHz

As can be seen in Table IV, the Jacobi method outperforms the Gauss-Seidel method even though the the same hardware is deployed and Gauss-Seidel method has fewer iterates. This is due to keeping the hardware pipeline as full as possible. Some improvements to these numbers can be attained by re-ordering the rows of the calculation possibly at the expense of additional iterations.

These results were calculated using a single Jacobi pipeline. By the nature of the Jacobi method, multiple pipelines may be used concurrently on different equations in the problem. As noted earlier, there is no dependence between the equations when using the Jacobi method. Performance could be boosted by a factor of how many pipelines are running concurrently.

5.1 Projected Performance vs. Newton-Raphson

As shown in the previous section, the FPGA implementation of the Jacobi method outperforms a similar FPGA implementation of the Gauss-Seidel method. In this section, we will use our findings to predict what an implementatin of Newton-Raphson would have to perform to meet the performance of our proposed Jacobi FPGA pipeline.

Newton-Raphson algorithm is regarded as the industry standard for load flow calculations. It is much more complicated to program than Gauss-Jacobi with having to solve the real and reactive power mismatch equations and the creating the Jacobian on every iterate. Newton-Raphon has many more computations for each row of the Y-bus, but the number of iterations to compute a solution is significantly less than Gauss-Jacobi.

As it can be seen previously in in Table II, the total number of floating-point operations for Newton-Raphson is fewer than Gauss-Jacobi for all bus sizes, but the ratio of operations between the two methods reduces as the problem size increases. This means that Newton-Raphson has to work harder for larger systems to obtain the same throughput as Gauss-Jacobi. Table VI shows the ratios of operations for our benchmark systems as well as the necessary effort of Newton-Raphson to calculate the same load-flow problem as the 125 MHz Gauss-Jacobi FPGA pipeline.

Bus	Jacobi:NR Operations Ratio	Equivalent Mega-Operations/Sec
118	208.99	6.59
300	231.69	7.36
1648	57.28	24.59
7917	39.02	35.14

Table VI: Comparing Jacobi and Newton-Raphson Operations

The Newton-Raphson method significantly outperforms the Jacobi FPGA pipeline for the smaller systems; the larger systems are more interesting. By extrapolating the results shown to even larger systems, an estimate of nearly 100 MOps/sec would be needed to compute a system of 10,000 busses.

To adequately compare this FPGA engine to software implementations, the speed of the CPU executing the software is important. Popular personal computers at the time of this writing are capable of running at internal CPU speeds nearing 4 GHz. They are also armed with multiple layers of memory caching and practical on-chip floating-point units. Johnson and Vachranukunkiet stated that machines of this caliber can only realistically support approximately 1% sustained floating-point performance. On a 4 GHz machine that would equate to 40 MFLOPS.

Comparisans of FPGA hardware methods need to be performed when prototypes become available. These comparisons will entail performance, but also resources used, synthesized clock speed, and scalability.

6 CONCLUSION

The Jacobi method was once discounted as a viable algorithm in large scale load flow computations due to its slow rate of convergence. Software research and the creation of the Gauss-Seidel and Newton-Raphson methods have almost made the Jacobi method extinct. Our preliminary results show that the Jacobi method for large scale load flow should be further investigated when applied to FPGA technology. The simple algorithm translates well to a deeply pipelined hardware architecture. It is the power of the pipeline and available parallelism that allow the FPGA hardware to have performance to potentially outpace the Newton-Raphson implementations.

REFERENCES

[1] Feng Tu and A. J. Flueck, A message-passing distributed-memory parallel power flow algorithm, Power Engineering Society Winter Meeting, 2002. IEEE, Volume 1 (2002), pp 211 – 216

[2] A. R. Bergen, V. Vittal, Power Systems Analysis, 2nd Edition, Prentice Hall, 2000, pp. 323 – 367

[3] Keith Underwood, FPGAs vs. CPUs: Trends in peak floating point performance, ACM/SIGDA Twelfth ACM International Symposium on Field-

Programmable Gate Arrays, (Monterrey, CA), February 2004

[4] Ling Zhuo, Gerlad R. Morris, and Viktor K. Prasanna, Designing Scalable FPGA-Based Reduction Circuits Using Pipelined Floating-Point Cores, Reconfigurable Architecture Workshop (RAW), joint with IPDPS, April 2005

[5] Ling Zhuo and Viktor K. Prasanna, Sparse Matrix-Vector Multiplication on FPGAs, Thirteenth ACM International Symposium on Field-Programmable Gate Arrays, February 2005

[6] J. Johnson, P. Vachranukunkiet, S. Tiwari, P. Nagvajara, C. Nwampka, Performance Analysis of Load Flow Computation using FPGA, to be published at Power Systems Computation Conference, 2005