An FPGA Based Reconfigurable Coprocessor Board Utilizing a Mathematics of Arrays

H. Pottinger, W. Eatherton, J. Kelly, T. Schiefelbein
Department of Electrical Engineering

L. R. Mullin, R. Ziegler
Department of Computer Science
University of Missouri - Rolla

Abstract -- Work in progress at the University of Missouri-Rolla on hardware assists for high performance computing is presented. This research consists of a novel field programmable gate array (FPGA) based reconfigurable coprocessor board (the Chameleon Coprocessor) being used to evaluate hardware architectures for speedup of array computation algorithms. These algorithms are developed using a Mathematics of Arrays (MOA). They provide a means to generate addresses for data transfers that require less data movement than more traditional algorithms. In this manner, the address generation algorithms are acting as an intelligent data prefetching mechanism or special purpose cache controller. Software implementations have been used to provide speedups on the order of 100% over classical methods to the solution of heat transfer equations on a uniprocessor. We extend these methods to application designs for the Chameleon Coprocessor.

1. Introduction

The coprocessor architecture presented in this paper is based upon the concept of using the FPGAs as reprogammable and intelligentcache controller in place of the general purpose cache controllers found in current microprocessors. The argument for the need of a special purpose cache controllers for array processing is that for large multidimensional arrays the cache is not utilized efficiently at all and has frequent misses. Additionally for most workstations with virtual memory references, the Translation Lookaside Buffer (TLB) can only accommodate several hundred KB of data [1]. Therefore TLB misses and cache misses will result in more time being spent on address generation and memory access than on the actual operation being performed on the array. One such example, a MC88100 RISC processor required 9 instructions to compute an address and only 3 instructions to compute and assign the data for that address. Each instruction in the loop took one clock cycle to complete. In this case three times longer was spent on address generation than the computation.

By implementing the generation of addresses in hardware, not only can methods of optimizing the main memory address patterns be explored, but the size of the cache being used can be taken into account

Classical array accessing provides unnecessary computational overhead. Use of MOA for hardware or software algorithms means that the address generation overhead of array referencing is reduced and array access is speeded. MOA provides a formal way of describing array operations. Generally, these expressions are at a high level and contain cartesian referencing. These expressions can be reduced to a normal form that only contains the information needed to generate a linear access pattern for the array in physical memory. These patterns are quickly computable since they contain only additions and multiplies and they are fast because they access the array minimally to carry out the operation at hand.

2. An Introduction to MOA

A Mathematics of Arrays can be used to describe mathematical array operations regardless of their shape, size, or dimensionality. MOA describes an array calculus containing a set of operator definitions, shape definitions, and reduction rules all based on a single indexing operator, ψ. For this reason, MOA is often referred to as the Psi Calculus. Algebraic operators are included in the Psi Calculus to form a broad set of operators needed to describe complex array operations. All the operators are extended for scalars, vectors, and multi-dimensional arrays.

MOA is defined in [2] and is based on Abrams' work on the simplification of array expressions [3]. The advantages of expression reductions and the correspondence between cartesian and linear referenced arrays are described in [4].

Table I lists some of the more useful Psi Calculus operators together with an example usage on the array,

$$\xi_e^{\,2} \equiv \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

Table I. Some MOA Operators

Operator	Function	Example
δ	Dimensionality	$\delta\xi_e = 2$
ρ	Shape	$\rho\xi_e = \langle 2\ 3 \rangle$
Δ	Take (subarray)	$1\Delta\xi_e = \langle 1\ 2\ 3 \rangle$
∇	Drop (subarray)	$1\nabla\xi_e = \langle 4\ 5\ 6 \rangle$
rav	Ravel (flatten)	$rav\xi_e = \langle 1\ 2\ 3\ 4\ 5\ 6 \rangle$
ι	Iota (count)	$\iota 5 = \langle 0\ 1\ 2\ 3\ 4 \rangle$
ψ	Psi (index)	$\langle 1\ 0 \rangle \psi \xi_e = 4$

From Table I, dimensionality returns a scalar indicating the total number of dimensions of an array. Shape returns a vector describing the length along each dimension. Take and Drop return subarrays by taking or dropping data from the right hand argument array by using the left hand shape vector argument. Ravel forms a vector from an array, filling it with data indexed in an incremental row major fashion, effectively flattening the array. Iota produces a vector containing a cardinal string of n elements starting from zero. Iota implies time referencing or looping if it is used for indexing. Thus iota is essential for generating array access patterns.

Psi is a generalized indexing function that takes a vector right hand argument representing, usually, a cartesian reference to data in the right hand argument array and returns that data which may be a scalar or a subarray. Every Psi Calculus operator is defined in terms of psi, therefore, through the use of reduction rules, an array expression can be simplified to pure indexing using psi along with any necessary arithmetic operations.

There are a few general procedures to follow to obtain an implementable form of an array expression. First, an array operation is described at a high level using the Psi Calculus operators. This expression is usually the most understandable and straight forward to the designer. It also describes the operation holistically without regard to how the arrays actually get referenced. For example, the expression **B** = **A** + 1 implies that every element of **A** should be incremented by 1 although no indication of how this is done is described. It is the job of the Psi Reduction Rules to describe how **A** and **B** get ultimately referenced.

Next, the expression is reduced to the Semantic Normal Form (SNF). To do this, the definitions of the operators are used to calculate the resultant shape of the expression. Then, reduction rules are applied to the expression, using the definitions of the operators, to obtain an expression that only uses the psi operator and arithmetics if necessary. At this point, referencing is cartesian. This process can become lengthy and complex but is partially automated in the Psi Compiler [5]. After compiling, memory access is implemented with starts, stops, and strides. It is important to note that the Psi Calculus has one very reassuring property. Any expression representing the high level description of the array computation will reduce to the same SNF.

The SNF is a representation of the array expression that uses the least amount of memory, memory access, and computational redundancy. Since, referencing in the SNF is cartesian, it is unclear how to implement this form in software or hardware. An application of the Psi Correspondence Theorem [4] will transform the SNF into the Operational Normal Form (ONF). At this point, the arrays are flattened and are indexed linearly as would be necessary in a physical linear memory. In fact, the Psi Compiler does this with the start, stop, and stride indexing. In a software implementation the number of loop variables is constant, regardless of dimension. This is also what is desired for a hardware representation.

MOA, then, offers a method to systematically derive address generation mechanisms. The alternative would be to hand derive an algorithm that is prone to errors. It is also difficult to incorporate architecture dependencies like data paths, caches, memories, and other architectures. With MOA, a high level expression may allow for cache sizes or burst transfer modes by repartitioning array data into smaller manageable pieces automatically. Many of these optimizations could be done by hand, but what is really needed is a formal method that keeps the whole picture in perspective which MOA provides.

3. An Application Example: 2D Heat Transfer

Heat transfer is a dynamic process usually requiring the solution to complex partial differential equations. An alternative solution is to use a numerical technique and a time simulation. Heat transfer can be simulated with a quantized mesh model using finite difference time domain methods. For large problems involving huge meshes and many iterations in time, every bit of computational performance counts in a system. Any optimization on such a system would be beneficial. In [6], the Psi Calculus was used to derive a software algorithm that outperformed traditional heat transfer algorithms because it took advantage of non-random array access patterns.

This work will be extended by the implementation of the heat transfer algorithm on the Chameleon Coprocessor. Results here are projected but will be conclusively obtained pending the completion of the board. An example derivation and comparison will be done for a two dimensional version of the heat transfer algorithm.

For 2D heat transfer, a two dimensional array represents a surface about which heat can flow. The value at each point is the relative temperature of the point at a fixed time. As time progresses, the heat will flow until an equilibrium is established determined by the constant boundary conditions. Consider the situation of Fig. 1. Here a sinusoidal distribution of temperature is initialized with the z-

axis representing the relative temperature at each point. As time progresses and more iterations are applied, the heat transfers out of the system as it reaches eventually a state of constant temperature. This is due to the constant boundary conditions of zero relative temperature which are not displayed.

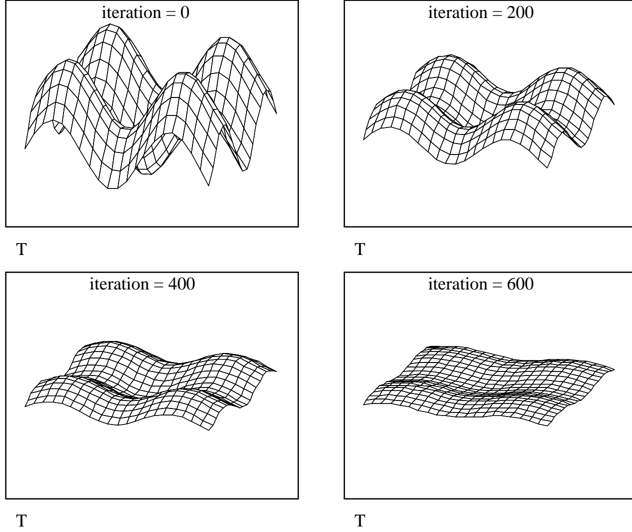

Fig. 1 Time Dynamics of Heat transfer

Due to the nature of the equations that describe heat transfer and what is physically observed, heat transfer is really then an averaging function. During one iteration, a new value for every interior point is calculated based on the average of it and its adjacent neighbors. This new value is placed in temporary storage so that it does not effect the computation of its next neighbor. All points on the surface are averaged in this manner to complete the iteration. When the finite difference between one iteration and the next iteration is below some predefined threshold, a solution is said to be found. From [a], discrete solutions to the PDEs that describe heat transfer yield the averaging function,

$$u'_{i,j} = u_{i,j}\left(1-4\lambda\right)+\lambda\left(u_{i-1,j}+u_{i+1,j}+u_{i,j-1}+u_{i,j+1}\right)$$

Eq. 1

where u' is the iterations result, λ is the heat transfer rate constant, and the other variables represent the current values of the central element and its neighbors as indicated by Fig. 2.

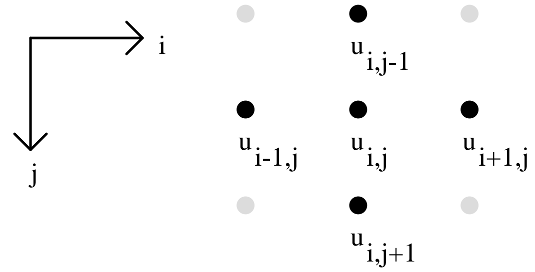

Fig. 2 Central Element and Neighbors

Traditionally, Eq. 1 is used to access the array data to perform the heat transfer algorithm. Performance is generally poor since each element is accessed five times (whenever it or its adjacent neighbor is being computed) during the course of one iteration. This assumes no form of data caching is present. If a general data cache is present, each element is accessed three times since there will cache hits for data in the same row. There is a method to access the data such that each element is only accessed once from main memory across the SBus. This access pattern is not random and can be found by utilizing the Psi Calculus.

To describe heat transfer, first, a high level description must be created using the MOA operators. This description must embody the mathematics of the solution to the PDEs as Eq. 1 does. One such description is,

$$\mathbf{A}_2 = \lambda\begin{bmatrix}\left(\mathbf{b}\Delta\left(\langle 1\,2\rangle\nabla\mathbf{A}\right)\right)+\left(\mathbf{b}\Delta\left(\langle 1\,0\rangle\nabla\mathbf{A}\right)\right)+\\ \left(\mathbf{b}\Delta\left(\langle 2\,1\rangle\nabla\mathbf{A}\right)\right)+\left(\mathbf{b}\Delta\left(\langle 0\,1\rangle\nabla\mathbf{A}\right)\right)\end{bmatrix}$$

Eq. 2

$$\left(\mathbf{b}\Delta\left(\langle 1\,1\rangle\nabla\mathbf{A}\right)\right)=\left(\mathbf{b}\Delta\left(\langle 1\,1\rangle\nabla\mathbf{A}\right)\right)(1-4\lambda)+\mathbf{A}_2$$

Eq. 3

where **b** is the shape vector of the interior of **A** (without boundaries).

The 2D array $\mathbf{A_2}$ in Eq. 2 is a temporary holding the sum of four partitions of the original data array **A** multiplied by the rate constant, λ. This expression describes the computation involving the four adjacent neighbors of each central element all at once and is analogous to the one element computation of the second part of Eq. 1. Next, in Eq. 3 the interior partition of **A** is multiplied by the rate expression as in Eq. 1 and is summed with the intermediate result array $\mathbf{A_2}$ to complete the iteration.

It can be seen at this level that **A** is being accessed linearly and not in the more complex pattern implied by Eq. 1. Because of this, features like a data cache or burst transfer modes can be taken exploited on a bus.

The next step to be taken is to reduce Eq. 2 and Eq. 3 to SNF. This is done by the Psi Reduction rules and only the results will be shown here.

$$\forall \mathbf{i} \text{ s.t. } 0 \le^* \mathbf{i} <^* \mathbf{b}$$

$$\mathbf{i}\Psi[\mathbf{A}_2] = \lambda \begin{bmatrix} (\mathbf{i} + \langle 12 \rangle)\Psi\mathbf{A} + (\mathbf{i} + \langle 10 \rangle)\Psi\mathbf{A} + \\ (\mathbf{i} + \langle 21 \rangle)\Psi\mathbf{A} + (\mathbf{i} + \langle 01 \rangle)\Psi\mathbf{A} \end{bmatrix}$$

Eq. 4

$$[(\mathbf{i} + \langle 11 \rangle)\Psi\mathbf{A}] = [(\mathbf{i} + \langle 11 \rangle)\Psi\mathbf{A}](1 - 4\lambda) + \mathbf{i}\Psi\mathbf{A}_2$$

Eq. 5

Notice, the only MOA operator present is Ψ for indexing. The bounds on **i** imply two nested loops to carry out the operation. An application of the Psi Correspondence Theorem is necessary to uncover how memory access should be performed. The result is,

$$b = n - 2$$

$$\begin{aligned} (\text{rav}\, A_2)[\iota(b) + (b)[\iota(b)]] = \lambda[(\text{rav}\, A)[\iota b + 2 + n(\iota b + 1)] + \\ (\text{rav}\, A)[\iota b + n(\iota b + 1)] + (\text{rav}\, A)[\iota b + 1 + n(\iota b + 2)] + \\ (\text{rav}\, A)[\iota b + 1 + n(\iota b)]] \end{aligned}$$

Eq. 6

$$\begin{aligned} (\text{rav}A)[\iota b + 1 + n(\iota b)] = (\text{rav}A)[\iota b + 1 + n(\iota b)] \\ (1 - 4\lambda) + (\text{rav}\, A_2)[\iota b + b(\iota b)] \end{aligned}$$

Eq. 7

The iotas provide the means of indexing at this point and can easily be implemented as loops in software or counters in hardware. The ravels simply indicate base addresses of respective arrays on which they operate. What is left is simple arithmetics.

Thus, through the use of MOA, an implementation for intelligent address generation has been systematically derived that is provably correct. This algorithm will generate address streams that are simple starts, stops, and strides. Again, this means that architectural elements like local caches or burst transfer modes can be exploited.

4. Hardware Acceleration

While the ideal architecture for exploring address optimizations would involve placing the CPU for a system on the same die or in the same Multi-Chip Module (MCM) as a reconfigureable cache controller, we believe significant speed ups can still be achieved at much less initial cost by implementing the device as a peripheral coprocessor card. The simplest implementation of such a coprocessor card would be the generation of addresses as described above and then the loading of the host workstations primary cache. However, as suggested above most current microprocessors incorporate the CPU, cache controller, and primary cache on the same die or MCM making it impossible to externally control the cache. Therefore, to achieve significant speedups over current microprocessors through this method of speeding up address generation, it is necessary for the coprocessor board to generate the addresses, fetch the data from the host memory, do the necessary computations, and replace the date in the host memory. To perform these series of functions the Chameleon Coprocessor board needed to have a reconfigurable address generator, an interface to a relatively high bandwidth bus, a method of doing high speed floating point computations, and high speed on board memory (this would be the board's "cache").

4.1 Reconfigurable Address Generator

The primary component of the coprocessor board described above is the reconfigurable address generator. After investigating the results of using MOA to simplify several array algorithms, the end result was two nested loops regardless of the number of dimension in the array as seen in Eq. 6 and Eq. 7. However , there were many variations in the size and boundary handling of these loops due to the number of dimensions and the nature of the operation being performed. Therefore a standard cell implementation of an address generator to handle all such possibilities would not be practical. Many general purpose processors would be capable of generating the addresses but would take several clock cycles and not allow significant speedups over conventional techniques. FPGAs however would be ideal for the purpose of the address generation since each array application could have its own special address generator. The large number of flip flops in Xilinx FPGAs [7] for example, makes the large counters necessary for each loop index feasible and the availability of 10,000 gates or more allows construction of the adders and other combinational structures necessary to handle array boundary conditions. Additionally the flexibility of the FPGAs allows you to separate the generation of the input and output addresses for each calculation. In this way the board can support block processing of data.

4.2 Interface to High I/O Bus

The next item needed for the board is an interface to a relatively high bandwidth bus. The SBus used on many Sun Workstations was chosen for the target bus. While initially it was thought that a higher bandwidth bus supporting multiprocessors like the MBUS would be more desirable, it was decided that the wider availability of the SBus (at least to us) and the simpler protocol was more

important. One major point of interest about the SBus implementation on most Sun Workstations is a half megabyte limit on the amount of space that can be allocated to any peripheral master on the SBus at one time.

The requirements of the interface to the SBUS was that it first be able to fetch data from the addresses generated on the board. Then it needed to be able to direct the fetched data to the appropriate destination which turned out not to be the address generating chip. Finally, it needed to be able to take the computation results and the appropriate addresses and write the data to main memory. Additionally the interface to the SBUS was needed to support a wide variety of transfer sizes, both slave and master mode of operation, have fast internal buffering, and finally detect and record both SBUS and board errors for retrieval by the host workstation. To maximize performance of the board it was also hoped that the data transfer operations could be performed in parallel in order to most fully saturate the SBUS. While there are some fine off the shelf components for interfacing to the SBUS, none were found to posses the flexibility and data transfer patterns necessary to fit these requirements. Once again FPGAs, specifically the Xilinx 4013-4, was found posses both the size and speed to meet our requirements. The present version of the interface chip utilizes approximately 85% of the chip and is able to meet SBUS timing specs at up to 20 MHz. While the SBUS specs call for a compliance of the specs up to 25 MHz, the workstations the board will be tested on run at 20 MHz or below. Additionally current work on the optimization of the interface design will hopefully soon yield compliance at up to 25 MHz.

4.3 Computation Implementation

Before deciding on how to go about performing the actual array computation, the form of the algorithms being implemented should be defined. Most array operations being considered for optimization thus far, are of an averaging nature were each point in the next iteration is a weighted average of the its last value and the last values of the bounding points. This type of calculation includes heat transfer through time as observed in this paper and many other applications like electromagnetic field calculations on a printed circuit board. These calculations typically involve multiplication of the input operands by the appropriate weight constant, and then accumulation and multiplication by the averaging factor like $(1\text{-}4\lambda)$ from Eq. 7. Another notable characteristic of these calculations is several input operands and only one output operand.

It has been shown for that for a wide variety of computations, FPGAs can provide substantial speedups over general purpose processors. Examples are simple arithmetic operations [8], image processing algorithms [9] [10] [11], and sequence comparisons [12], Most of the architectures in these papers achieve speedups by using varying degrees of parallelism. However, without software assistance in designing the FPGAs used in these systems, the partitioning and layout is a monumental task. It is due to this lack of software support that FPGAs were not chosen to perform the actual computations for the board.

What is ideally needed to perform the computations for the board is a FPU that is easy to control, and does high speed multiplication and addition. For available FPUs like the Cyrix 83D87, the multiplication of two floating point numbers takes 19 clock cycles. Even for some of the high cost chips like the Weitek 4167 a multiply takes 3 clock cycles and an addition 2 clock cycles [13]. The lack of powerful, standalone arithmetic units with features like internal registers has been documented previously [14] . Therefore the ADSP21020 was selected due to the fact that it not only can perform all desired operations in a single cycle but can simultaneously do arithmetic operations and memory accesses. By creatively programming the DSP, the DSP chip can be utilized as a high speed FPU under the control of a FPGA. In this way a DSP program can be created once for each application and then the FPGA will handle changes in dimensionality and size.

4.4 Board Cache Selection

The final major design area to be explored is the task of selecting what memory storage should be included on the board. First of all a 48 bit wide block of memory is needed to hold the DSP program and therefore a bank of 6 128Kx8 SRAMs was placed on the DSP chips program memory bus. The important decision is the size and type of memory storage for the on board cache. Some choices for the form of the cache were large FIFOs, dual ported SRAMs, standard SRAMs, and drams. In order to run the DSP chip selected at it's top speed of 33 MHz, an access time of 15ns was needed. This requirement eliminates the DRAM choice and most dual ported SRAMs. Additionally in order to allow the DSP chip to easily reuse data within the "block" loaded onto the board, the DSP chip needs to be able to access the data in a non-linear pattern eliminating the FIFO option. This leaves standard SRAMs for the memory implementation. With the 15ns access time requirement, the largest size easily available was 128Kx8. Since there is a half megabyte DVMA allocation limit for the SBUS, a cache 32 bits wide made up of 128kx8 SRAMs is just large enough to fit the entire space addressable with DVMA at one time. Then before moving data from the board, the host processor needs to reallocate a new portion of memory to write the new array.

4.5 Chameleon Coprocessor Data Flow

Fig. 3 shows the data flow for the final design based upon the requirements as described thus far. The first

step in running an iteration of a computation like the heat transfer involves several writes by the host workstation to the board in slave mode. These writes access the X1 FPGA and identify the size and possibly the dimensionality of the application that the X0 and X1 FPGAs have already been configured for. Since the board when in master mode utilizes virtual memory addressing , it is not necessary to exchange the starting address for the initial and final array. The board would simply as a rule assume the arrays begin at address 0 and it is up to the host processor to perform the correct memory mapping.

Once the board has the necessary configuration data, the X1 FPGA begins address generation. The transfers are grouped into 16 word bursts when possible and the interface chip would retrieve the data as the addresses are generated. Once a block of data has been retrieved into a on chip buffer constructed from the LUT lookup tables, the interface chip controls the movement of data to the X0 FPGA. The order of the data as far as variables in the computation has been pre-determined prior to FPGA configuration. An important note is that since the interface chip does have three separate busses and two 16x32 buffers as shown in Fig. 3, it is possible to simultaneously receive an address from the X1 FPGA, fetch data from the SBUS into one buffer, and load data to the X0 FPGA from the second buffer.

Once the data has been loaded into the X0 FPGA, the next step is the intelligent loading of the data memory SRAMs by the X0 FPGA. The X0 FPGA does need to load the SRAM in linearly but can do so in an intelligent fashion such that the DSP chip expends minimal effort in its address generation. While it would be possible to have the DSP fetch the data directly from the X0 FPGA and eliminate the need for any board cache this would involve a DSP wait state , create a bottleneck, and eliminate the advantages of doing block processing which allows the DSP chip to reuse operands. After the data memory block has been fully loaded, the DSP chip running at 33 MHz with no wait states and performing multiple instructions per clock cycle, can very quickly run through the computations.

As the DSP completes each computation the results are written out to the X1 FPGA on the program memory bus. The X1 FPGA stores up to sixteen results at a time an then generates the appropriate address for host memory storage and then passes the address and data to the interface chip.

5. Example Application Performance

For the Chameleon Board then, the FPGAs can be used to implement the address generation algorithms. It is a goal of future work to implement the algorithms on the board and compare performance to software methods and the more traditional address generation method. For now, only projections of results will be compared.

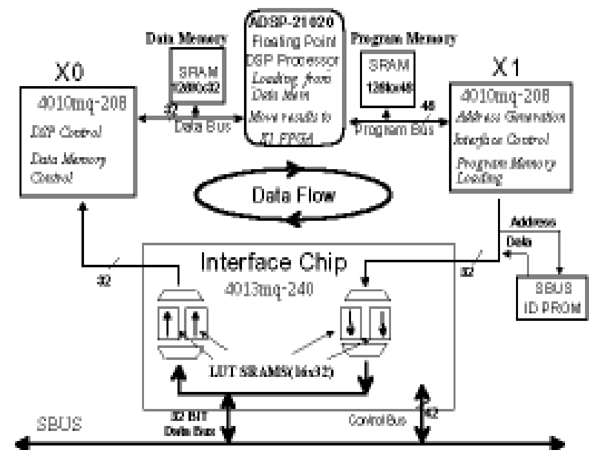

Fig. 3 Data Flow Block Diagram

Two assumptions are made to gain insight on predicted performance. First, the bandwidth predicted for the SBus is about 20 MB/s. This is a worst case possibility and is due to the interrupt service routine of the sun operating system Sustained bandwidth could theoretically reach 60 MB/s if there were no latency in interrupt servicing. Second, with the board architecture, each generation of data must be loaded, computed, and stored back to main memory. Since in the heat transfer application, there are more operands than results per computation, it is assumed the results can be written while the next data point is being computed.

The run times for a 160x160 2D heat transfer simulation for 500 iterations were compared. According to [a], a classical software approach on a Sun IPX took an average of 535.80 seconds to complete and an MOA derived software algorithm on a Sun IPX that took an average of 447.55 seconds to complete, a 16.5% speed increase.

For a traditional heat transfer algorithm, each piece of data must be fetched from main memory in single word transfer mode 5 times. With an average single word bus transfer rate of 11.5 MB/s, the traditional heat transfer simulation of a 160x160 (25,600 32 bit floating point numbers) will take 44.5 ms to complete the bus transfer. Additionally, the DSP and write cycles add another 4.66 ms for a single iteration. This requires 49.2 ms for one iteration over all and 24.6 s to complete the entire 500 iterations. Translating to hardware provides a 18.2 times decrease in run time over the MOA derived software implementation.

With an average worst case bus transfer rate of 20 MB/s in burst mode, the MOA derived heat transfer simulation of a 160x160 (25,600 32 bit floating point numbers)

will take only 5.12 ms to complete the bus transfer. The DSP and writes add another 4.66 ms as before. Thus, one iteration requires only 9.78 ms and 500 iterations require only 1.77 s. This is more than a 5 times increase over the traditional method in hardware. This is due to the reduction in the redundancy of data transfer and the reuse of data operands. If the IPX SBus bandwidth reaches its top rate of about 60 MB/s, then performance would jump to over 7.6 times that of the traditional hardware method.

6. Conclusion

A hardware accelerator for a Sun workstation, called the Chameleon Coprocessor, consisting of an interface FPGA, a pair of application reconfigurable FPGAs and a high speed DSP is currently being fabricated to support this effort. MOA will be used as a basis for a design paradigm for applications on the Chameleon Coprocessor. FPGAs were selected for the board in order to implement the large, fast state machines needed to realize the data prefetch algorithm and to provide a means for prototyping many applications. Finally, an initial application based on the heat transfer algorithm has been designed for the board using the MOA hardware design paradigm.

It is important to note that while the coprocessor architecture presented in this paper has projected performance gains over software implementations, this architecture is not presented as optimal. Instead the Chameleon Coprocessor is a proving grounds in the area of optimizing data flow. Replacing the DSP chip with a more powerful computational unit like a parallel computing architecture based on FPGAs could take advantages of the concepts suggested here while reducing computation time and therefore reducing total processing time.

References

[1] Dowd, K., *High Performance Computing*. O'Reily & Associates, Inc., Sebastopol, CA, first edition, 1993.

[2] Mullin, Lenore, "A Mathematics of Arrays", Ph. D. dissertation, Syracuse University, December 1988.

[3] Abrams, P.S., "What's wrong with APL", APL 75, ACM, June, 1975.

[4] Mullin, L., “The Psi Correspondence Theorem: Array Mapping Using the Psi Calculus”, Department of Computer Science, University of Missouri - Rolla.

[5] Mullin, L., Thibault, S., "A Reduction semantics for array expressions: the PSI compiler", TR CSC-94-05, March 9, 1994, Department of Computer Science, University of Missouri - Rolla.

[6] Coffin, Larry, “Designing a New Programming Methodology for Optimizing Array Accesses in Complex Scientific Problems”, OURE Paper, University of Missouri - Rolla, 1994.

[7] Xilinx, Inc, 2100 Logic Drive, San Jose, CA 95124 . *The Programmable Logic Databook, April 1994.*

[8] Wo, D., Forward, K., “Compiling to the gate Level for a Reconfigureable Co-Processor”, Proceedings of FPGAs for custom computing machines (1994), pp 147-154.

[9] Abbott, A., Athanas, P., Chen, L., Elliott, R., “Finding Lines and Building Pyramids with Splash 2”, Proceedings of FPGAs for custom computing machines (1994), pp 155-161.

[10] Gent, G., Smith, S., Haviland, R., “An FPGA-based Custom Coprocessor for Automatic Image Segmentation Applications”, Proceedings of FPGAs for custom computing machines (1994), pp 172-179.

[11] Quenot, G., Kraljic, I., Serot, J., Zavidovique B., “A Reconfigureable Compute Engine for Real-Time Vision Automata Prototyiping”, Proceedings of FPGAs for custom computing machines (1994), pp 91 - 100.

[12] Arnold, J., Duncan, Buell, A. , Hoang, D., “The Splash 2 Processor and Applications”, Proceedings ICCD ‘93.

[13] Ferguson, W., “Selecting Math Coprocessors”, IEEE Spectrum, July 1991, pp 38-41.

[14] Bergmann, N., Mudge, J., “Comparing the Performance of FPGA -Based Custom Computers with General -Purpose Computers for DSP Applications”, Proceedings of FPGAs for custom computing machines (1994), pp 164-171.