REDUCING COMMUNICATION THROUGH BUFFERS ON A SIMD ARCHITECTURE

A Dissertation
Presented to
The Academic Faculty

By

Jee W. Choi

In Partial Fulfillment
Of the Requirements for the Degree
Master of Science in Electrical and Computer Engineering

Georgia Institute of Technology
April 28, 2004

REDUCING COMMUNICATION THROUGH BUFFERS ON A SIMD ARCHITECTURE

ACKNOWLEDGEMENT

I would like to express my sincerest gratitude to my advisor, Dr. Donald Scott Wills for his guidance, support and encouragement throughout my time at Georgia Tech as a graduate student. Without his help and his unending encouragements, I would not have been able to complete this work.

I would like to thanks Dr. Sudhakar Yalamanchili and Dr. Hsien-Hsin Lee for serving in my thesis committee.

I would also like to thank the member of the PICA group for their helpful advices in times of need.

Last but not least, I would like to thank my parents for their support, encouragement and sacrifices that have brought me this far in life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Advances in wireless technology and the growing popularity of multimedia applications have brought about a need for energy efficient and cost effective portable supercomputers capable of delivering performance beyond the capabilities of current microprocessors and DSP chips. The SIMPil architecture currently being developed at Georgia Institute of Technology is a promising candidate for this task. In order to develop applications for SIMPil, a high level language and an optimizing compiler for the language are essential. However, with the recent trend of interconnect latency becoming a major bottleneck on computer systems, optimizations focusing on reducing latency are becoming more important, especially with SIMPil, as it is highly scalable. The compiler tracks the path of data through the network and buffers data in each processor to eliminate redundant communication. With a buffer size of 5, the compiler was able to eliminate 96 percent of the redundant communication for a 9x9 convolution and 8x8 DCT algorithms. With 5x5 convolution, only 89 percent elimination was observed. In terms of performance, 106 percent speedup was observed with 9x9 convolution at buffer size of 5 while 5x5 convolution and 8x8 DCT which have a much lower number of communication showed only 101 percent speedup.

CHAPTER 1

INTRODUCTION

Advances in wireless technologies and the development of inexpensive storage media such as DVD (Digital Versatile Disc) in recent years have enabled large amounts of data to be transmitted or transported to any place on the globe. With growing popularity of multimedia applications, this allows more powerful multimedia applications such as video email and teleconferencing, manipulating captured image or video or real-time rendering of 3-D objects to be carried out on portable devices such as a PDA (Personal Digital Assistant), a digital camera, or a cellular phone. However, in order for processors to be used on such devices, they have to not only be small and powerful, but also energy efficient and cost effective. The processors found on current PDAs or cellular phones are not capable of providing the required computation. SIMPil (SIMD Pixel Processor), a focal plane SIMD architecture currently being studied at Georgia Tech is a promising candidate that may meet the requirements of portable multimedia supercomputing [15].

The SIMPil architecture is well-suited for multimedia applications because it exploits the tremendous amount of data parallelism inherent in media-centric workloads. Workload can be broken down in to smaller blocks and distributed over the Processing Elements (PEs) and processed in parallel, achieving performance that cannot be matched by common microprocessors and DSP architectures, or even by some other parallel architectures [13]. SIMPil uses only the NEWS mesh network connections. This differs from the MasPar MP-1 which employs the X-net mesh interconnect for 8-way local communication and the multistage crossbar interconnect for global communication [17], or the Connection Machine CM-1 which employs a NEWS (north, east, west, south) network for local communication and a router for global communication. Although this may seem as a disadvantage as communication between distant processors would take longer, it allows savings in terms of hardware costs and simpler operations as the communication latency is deterministic [9].

In order to facilitate software development on the SIMPil architecture, a high level language and a compiler for the language are essential. Although often times, programs written directly in assembly language tend to be more efficient than those written in high level languages and then compiled, writing and debugging programs in assembly language is time consuming and difficult, especially as programs size gets larger. Also, the SIMPil architecture, with the NEWS network as its only means of communicating data across nodes, provides another challenge. As data travels further and more frequently, optimizing their movement through the network becomes more and more difficult. Therefore, having an efficient optimizing compiler is absolutely necessary, as it can not only minimize disparity in performance between compiled code and hand-written code, it can also reduce the number of unnecessary communications in the network.

Through out the last four decades, transistor density has increased at an exponential rate as predicted by Moore's Law, and the size of processors have become larger, incorporating even larger number of transistors, resulting in faster and more functional processors. However, these trends have also resulted in narrower and longer interconnects which increase the latency. In recent years, interconnect latency has become a major limitation on the scalability of system performance [20]. It is predicted in the ITRS (International Technology Roadmap for Semiconductors) 2003 that relative latency will continue to increase as transistors become even smaller, as shown in Figure 1 [19].

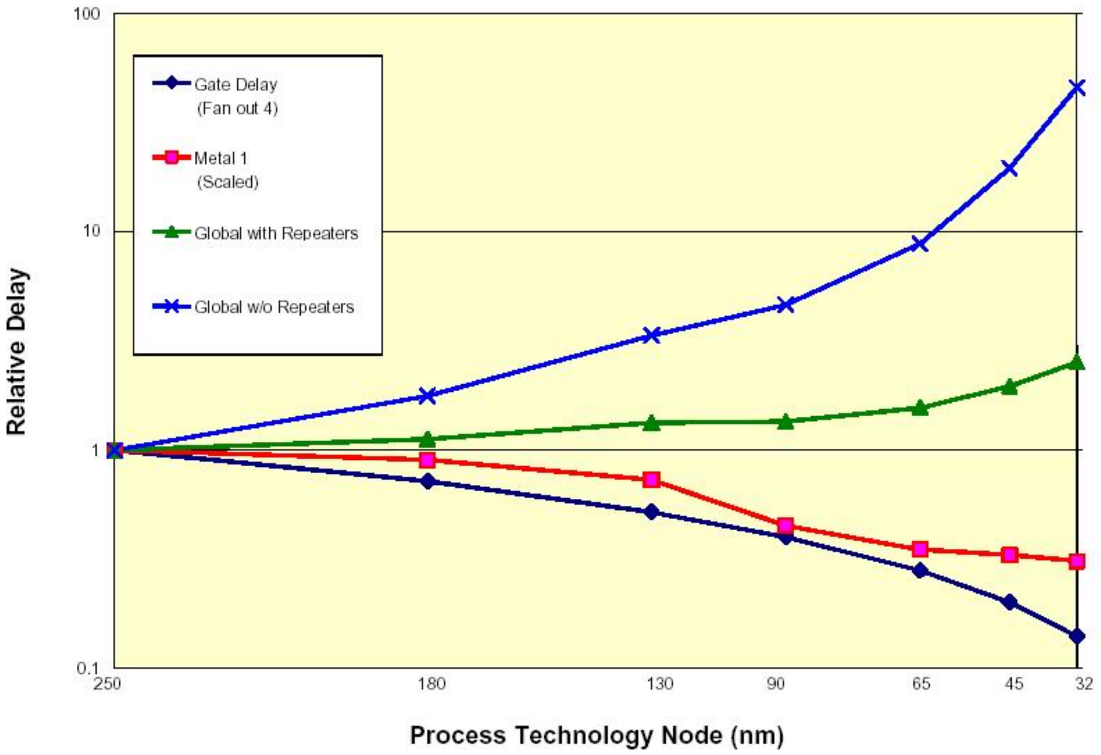

Figure 1 Delay for Metal 1 and global wiring versus feature size [19]

In the future, system designs will become interconnect-centric and design, process technology, packaging, and board construction will all need to come together to provide an integrated system level solution to interconnect requirements [19]. Some of the current research in interconnects include 3D interconnects [21], where interconnects link multiple layers of transistors and wiring stacked on top of each other, and new types of interconnects such as microwave and optical interconnects [20], and nanotubes.

Therefore, reducing redundant communication would, in addition to reducing the number of instructions, minimize the latency caused by communication that could reduce the performance of the system. This is especially important for SIMPil as it can easily be scaled by adding more processors. The added processors could increase the size of the network and the latency involved in communication. If the communication is not properly optimized, it could negate the benefit of having more processors. Although reducing communication comes at the costs of compilation overhead and extra memory usage, compilation occurs only once per platform and memory will become less of a cost as technology improves.

This thesis discusses how a compiler can track the path of data through the network and minimize the number of unnecessary communications by buffering the data in each processor. The algorithm decides whether to buffer the data or not by looking ahead in the program to see how much communication the data can eliminate. 5x5 convolution, 9x9 convolution and 8x8 DCT algorithms were used to exercise the algorithm and the results were compared against those of random and FIFO replacement policies.

Given enough buffers, the look-ahead algorithm was able to eliminate all redundant communication instructions, but the optimal size was found to be 5 at which 89 percent of the redundant communication was eliminated for 5x5 convolution, and 94 percent of the redundant communication was eliminated for 9x9 convolution and 8x8 DCT algorithms. After 5, the buffer hit the point of diminishing return. No performance gain in terms of execution cycles was seen with 5x5 convolution or 8x8 DCT as the percentage of communication was too small to affect the performance, but with 9x9 convolution, 6 percent reduction in execution cycles was seen with buffer size 5.

CHAPTER 2

BACKGROUND

ILLIAC-IV and ICL DAP

The history of SIMD systems can be traced back to the 1960's with the design of the ILLIAC-IV which was composed of 64 64-bit processors and had a parallelizing FORTRAN compiler called the IVTRAN. ILLIAC-IV was later followed by the ICL DAP (Distributed Array Processor) in the 1970's. However, it was not until the 1980's that interest in SIMD really picked up and led to the development of machines such as the Connection Machine and the MasPar.

Connection Machine

One of the first successful massively parallel machines in the 1980's was the Thinking Machines Corporation Connection Machine first introduced in 1985. It was designed for general purpose applications whose inherent parallelism could be exploited to increase performance. The CM (Connection Machine) virtual-machine parallel instruction set, called Paris, presents the users with abstract machine architecture. Paris runs on the sequencers, where the instructions are parsed and appropriate sequence of nano-instructions are generated for the data processors. Paris allows for a rich instruction set and a virtual processor abstraction, where the Connection Machine is initialized with a virtual number of processors to fit the application needs, and then the physical processors time-slice itself over multiple data regions that has been assigned to it. This enables effective and flexible programming. Paris is the target language of the high level language compilers that are commonly used on the Connection Machine.

The Connection Machine system software uses existing programming languages with minimal extensions to support data-parallel constructs to ease the pain of learning a completely new programming style, and of the many that are available, such as Fortran, which uses the array extensions in the draft Fortran 8x standard, and *Lisp and CM-Lisp, which are data parallel dialects of the Common Lisp, C* stands out in particular with its unobtrusive data parallel extensions, as it can be read and written like the common serial C programs [3].

MasPar

Another recognized SIMD machine is the MasPar. The MasPar architecture uses a different approach to software development in that it uses the ACU (Array Control Unit), a fully programmable computer, rather than a micro-programmed sequencer, thus depending more on good compiler optimization techniques to generate code than on pre-programmed complex instructions. The approach to processor virtuality taken by the MasPar is also different than that taken by the Connection Machine in that instead of building virtuality into the instruction set, it utilizes optimizing compiler technology as well as elements of architecture and machine design. Instead of assigning blocks of data to processors statically, it uses optimizing techniques to minimize data motion and optimize register usage. The use of optimizing compilers by the MasPar system allowed class of optimizations and flexibility that was not possible with an instruction set concept of virtuality exhibited on the Connection Machine, thus demonstrating the importance of compilers in the development of software.

MasPar provides adaptation of C and FORTRAN suitable for massively parallel machines just as the Connection Machine does. Three such languages are the MasPar FORTRAN (MPF), MasPar C (MPC), and the MasPar Parallel Application Language (MPL). MPF and MPC are languages that generate code for all parts of the system (the front-end system, the ACU and the parallel array) whereas MPL generates code for only the parallel subsystem (the ACU and the parallel array). The MPF and MPC are analogous to the Connection Machine’s FORTRAN and C* languages. The MPL, however, provides a simpler, massively parallel C for programming the ACU and the parallel array, and has three features that distinguish it from the regular C language. First, a variable can be declared as plural, in which case it is instantiated on all PEs; otherwise it can be declared as singular and instantiated only on the ACU. Expressions can mix plural and singular variables, in which case the scalar variables are broadcasted to the PEs and it is promoted to a plural variable. Control structure semantics are adapted to a SIMD computation model in that if a control expression is plural, then the expression is evaluated in a plural form, and only the active PEs execute the relevant code. Lastly, language syntax has been added to support the use of the X-net and the global router which allows PEs access to a plural variable on a neighboring PE, or an arbitrary PE, respectively [4].

Other Data Parallel Languages and Compilers

Other than the languages and their compilers for the Connection Machine and the MasPar that were mentioned above, many other languages and compilers exist for massively parallel architectures. The dbC language is a data parallel extension to the ANSI C similar to C* on the Connection Machine and the MPL on MasPar [6]. SC is another enhancement of C language for the Connection Machine which adds a few new data types and primitives to facilitate the development of data parallel programs. SC does not have a compiler for it but has a translator that converts SC language to C* language which is then compiled to Paris assembly instructions, thus providing a different approach to creating data parallel programs [5]. By utilizing C as its base language, most of the languages available for massively parallel machines depend on an existing compiler, namely the C compiler, to do much of its serial optimizations and compilation, but also provide an extension for programming parallel applications. This is also the approach taken by the SEP (Simple Explicit Parallel) language developed for the SIMPil architecture.

SEP Language for SIMPil and its Compiler, SIMcc

Before the development of the SEP language for SIMPil, all programming on SIMPil was done in assembly language. Although this satisfies most of the optimization for relatively small programs it is difficult to create large programs due to the sheer amount of coding required. The SEP language and the SIMcc compiler for the language was created to remedy this problem. The SEP language closely resembles the MasPar's MPL in that it uses a subset of C language enhanced with the three special grammar features of the MPL as described above to support massively parallel programming. Since SIMPil was targeted for portable multimedia computing, only a small subset of the C language deemed necessary for such applications was selected. This made the SEP language easier to program and to optimize [14]. The original SIMcc compiler for the SEP language was, however, inadequate in terms of achieving any sort of optimizations, as the compiler translated the SEP language directly in to SIMPil assembly language, without the use of any IR (Intermediate Representation) or dataflow analysis. This resulted in inefficient use of registers and thus reduced performance drastically.

Convolution and DCT in Multimedia Applications

Some of the most common multimedia applications use certain algorithms repeatedly. For example, many image processing applications such as noise removal and edge enhancement use convolution [7], and image compression techniques such as those used in the popular JPEG standard use DCT (Discrete Cosine Transforms) [11]. Techniques such as convolution and DCT have a common property in that they both require a large number of inter-processor communications. When programs are written directly in assembly language and the programs are small it is easy to minimize the total number of communications. However, when the program becomes more complex it becomes more difficult to hand optimize the code. Moreover, if the algorithm is modified for different applications, it becomes cumbersome to re-optimize the code. Using a high level language relieves the problem of complexity in programming but the problem of increased communications becomes more apparent, as the architectural details are hidden from the programmer. Since inter-processor communication time is generally longer than computational time, communication could become a bottleneck in algorithms that have a large number of communications.

Reducing Communications

In the past, several solutions to the problem of reducing communication time were proposed on SIMD systems. CCSIMD (Concurrently Communicating SIMD) hides communication latencies by overlapping communication and computation or by communicating simultaneously in multiple directions [9]. In another example, by utilizing a shared memory, the total number of communications for a bitonic sort on a sorting network was reduced [12].

Summary

It can be seen from comparing the software development environments of the Connection Machine and the MasPar utilizing an optimizing compiler can lead to improvements in performance that cannot otherwise be achieved. Similarly, in the case of the SIMPil architecture, the total number of communication can potentially be reduced while keeping the programming complexity minimal by utilizing the SEP language and an optimizing compiler. Although the SEP language provides a simple and easy programming environment, the original SIMcc compiler that was first developed for the language is inadequate in reducing the total number of communications as it provides no means of optimizing the generated assembly code.

In this paper, methods of reducing inter-processor communication for the SIMPil architecture through the use of high level language and its compiler will be developed and discussed. The overall relationship between different SIMD systems and their compilers and the SIMPil architecture and its SEP language and compiler is illustrated in Figure 2.

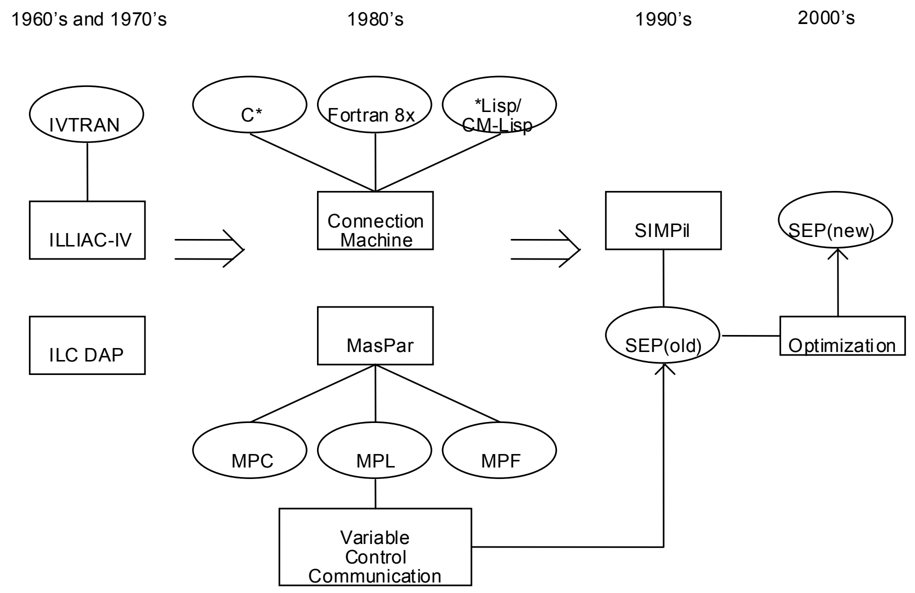

Figure 2 Impact of different SIMD systems and their compilers to the development of SEP and SIMcc and the need for an optimizing compiler for SIMPil

CHAPTER 3

METHODOLOGY

3.1 SIMPil: A SIMD Pixel Processor with Integrated Optoelectronic Detectors

SIMPil is a SIMD architecture developed at Georgia Tech for portable multimedia applications that exploits inherent data parallelism in images and videos. SIMPil also incorporates an integrated array of thin film detectors stacked directly on top of the processing plane to reduce the image data transfer bottleneck [15]. A cross sectional diagram showing the two layers is shown in Figure 3.

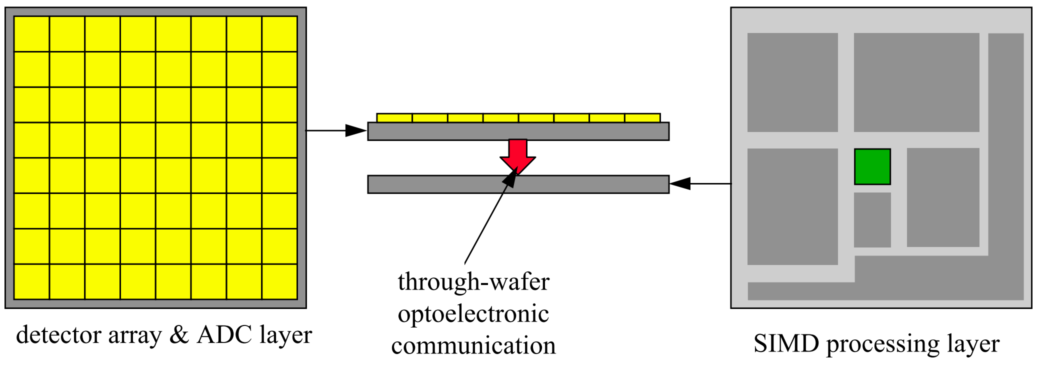

Figure 3 Cross sectional diagram showing the coupled detectors and processors [15]

SIMPil architecture consists of an Array Control Unit (ACU) connected to an array of SIMD PEs. Programs are stored in the ACU and broadcasted to every PE which, in turn, executes the instructions on the data in its local memory. Each PE communicates with its neighbors via a NEWS mesh network closed as a torus. The general architecture of SIMPil is shown in Figure 4

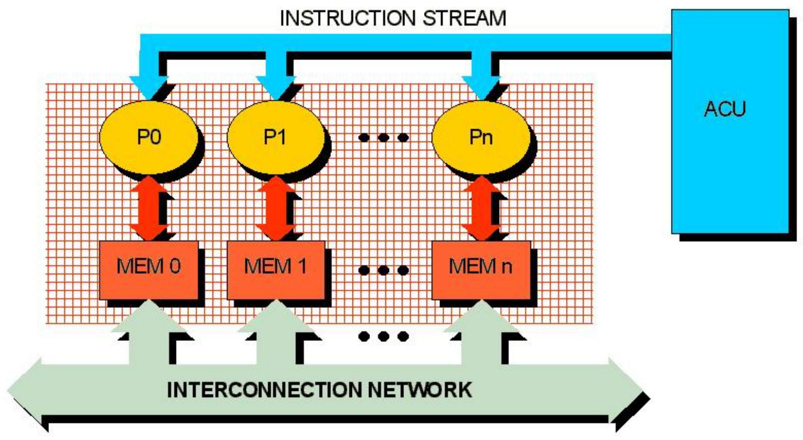

Figure 4 General architecture of the SIMPil system

Each PE is interfaced with a subset array of the thin film detectors stacked on top of the processing plane and an ADC (Analog to Digital Converter) to convert the light intensities incident on the detectors to a digital value. The microarchitecture of a PE consists of 16-bit datapath including an ALU, a barrel shifter, a multiply-accumulator unit, 16-word register file, and a local memory.

Overall, the SIMPil architecture provides the high computational throughput and data bandwidth required by modern multimedia applications while satisfying the low power and low cost constraints required of a portable processor. All tests done on the SIMPil system in this research was done through the SIMPil16 simulator.

3.2 SEP: A High Level Language for Programming on the SIMPil Architecture

In order to facilitate software development on any system, a high level language and a good compiler are essential, and SIMPil is no exception. The SEP language and its compiler, the SIMcc, were first developed by Gunther R. Costas for the SIMPil system with this in mind [14].

The SEP language was designed to make programming for the SIMPil system as easy as possible. The language constructs used in SEP were kept to only those that are necessary in creating common image processing applications and syntax similar to that of the widely used C language was adopted. SEP instructions include the WHILE loop, IF condition, arithmetic and logic instructions, and comparison operators. Also, there are special SIMPil specific instructions for communication and instructions for retrieving data sampled by the detector. An overview of instructions supported by SEP is shown in Figure 5.

SINGULAR	Singular variable declaration
PLURAL	Plural variable declaration
IF/ELSE	If/else condition
WHILE	While loop
GET	Communication instruction
LOAD	Retrieve data sampled by the detectors
SAMPLE	Sample image using detector
SHOW_IMAGE	Display image stored in memory
SET	Set architectural parameters
+, -, *, /, =	Arithmetic operators
&&, ||, ==, !=, <=, >=, <, >	Logical operators

Figure 5 An overview of instructions supported by SEP

SEP supports two types of variables: singular and plural variables. Singular variables are integer variables that are declared only on the ACU, and plural variables are those that are declared on every PE. Plural variables can also be declared as an array. Most SEP instructions can be used with either a singular variable or a plural variable. Operations between variables of the same data type evaluate to a variable of the same data type and operations between a singular variable and a plural variable will evaluate to a plural variable.

When the IF condition is used in conjunction with a condition that evaluates to a singular variable, the instruction is executed only by the ACU, and when it is used in conjunction with condition that evaluates to a plural variable, it is executed by all the PEs and only the PEs whose condition evaluates to true will execute the instructions within the IF condition.

The WHILE loop is similar to the IF condition. When the condition evaluates to a singular variable, the ACU executes the instruction like a normal WHILE loop. However, when the condition evaluates to a plural variable, the ACU sends instructions to the PEs and each PE executes the instructions within the WHILE loop as long as the condition evaluates to be true. However, instead of stopping here, the ACU repeatedly sends the instructions to the PEs until none of the PEs evaluate the condition to be true. A diagram of how a WHILE loop is executed by the PEs is shown in Figure 6.

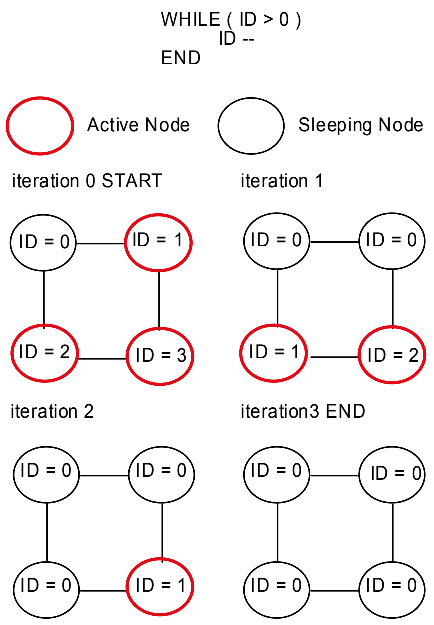

Figure 6 Execution of a SEP WHILE loop on SIMPil

The GET instruction is used to send data across the mesh network in any of the four NEWS directions. The LOAD instruction is used to load pixel data taken by the detectors to a variable. The various logical, arithmetic, and comparison operators used in SEP are identical to those used by the C language.

3.3 SIMcc: An Optimizing SEP Compiler for SIMPil

The original SIMcc compiler designed by Gunther R. Costas was a simple single stage compiler which generated SIMPil assembly instructions directly from a SEP program. Such simple compilers are often used for compiling simple languages like SEP as this design makes the compiler easier to implement. However, it also makes the compiler less efficient as it does not allow any form of optimization. Therefore, the original SIMcc compiler was redesigned from the parser level up to include the use of IR, optimizations and more efficient register allocation which was necessary to implement optimizations for communication. A diagram showing the various components of different compilers is shown in Figure 7.

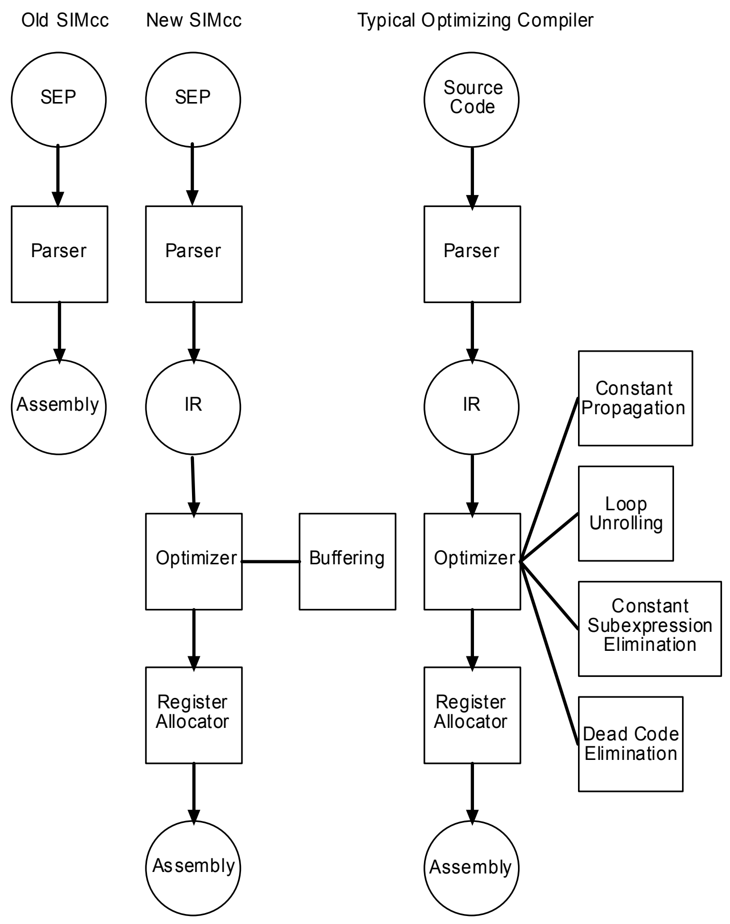

Figure 7 Comparison of various components of different compilers

The parser in the newly designed SIMcc generates IR instead of real SIMPil assembly instructions. The IR is often instructions that are representative of the assembly instructions but hides some of the architectural details that are unnecessary for optimizations. This way, after optimizations, the IR can be then translated to assembly instructions for the target system, and the optimizations won't have to be repeated when being compiled on a different architecture.

There are different formats for IR, but for SIMcc, quadruples were chosen as they most closely resembled the SIMPil assembly instructions, and this makes translating to assembly instructions much easier. Format of the quadruples and a list of IR instructions used in SIMcc are shown in Figure 8.

Quadruple Format

instruction	destination temporary	source temporary 1	source temporary 2

Quadruple List

add	subtract	multiply	divide	bra	beq
bne	bgt	bge	blt	ble	label
assign	move	assigntoarray	loadfromarray	sgt	sge
slt	sle	seq	sne	wakeup	and
or	raisehand	set	xfer	pload	ploads
sample	show_image	os_load_id	vectorize		

Figure 8 Format and a list of quadruples used in SIMcc

Once the parser generates IR, it can then be optimized. Typically optimizations such as constant propagation, constant subexpression elimination, dead code elimination, and loop unrolling are done before allocating registers and generating the final assembly instructions. No optimizations were done on the IR generated by SIMcc as they do not generally affect communication between PEs.

Once the IR and temporaries for the intermediate instructions were generated, the IR was converted to real assembly instructions. Then registers were allocated to the temporaries and the variables used in the program. Another major change that was made to the compiler was the use of a register allocator. In the old SIMcc, all variables were stored in the memory, and registers were used only when the data was processed. This is analogous to how the C compiler also keeps a copy of all variables in the memory, but in the case of SIMcc, it was unnecessary and it resulted in a very inefficient register usage. Therefore, in the new SIMcc compiler, G. Chaitin's graph coloring algorithm [1] was used to allocate registers and no variables were stored in the memory unless it was necessary. First, the IR was converted to basic blocks for liveness analysis, with each IR instruction being a basic block. Then, an interference graph was constructed from the result of the liveness analysis, and then the variables and the temporaries were allocated registers. If a register could not be found for a variable or a temporary, the variable was spilled to the memory, and the process was repeated from the beginning. Once the register allocation is complete, a set of assembly instructions that is executable on the SIMPil simulator is generated.

3.4 Optimizing Communications

3.4.1 Importance of Reducing Communication

Many common image and video processing applications on SIMD share data across nodes. In convolution a PE shares its data with all nodes surrounding it, with the radius of communication depending on the size of the mask. In DCT, nodes share data with PEs in the same row and with those on the same column. Other examples include DFT (Discrete Fourier Transform), matrix multiplications and median filtering. Some of these applications have such a large number of communications that a significant amount of time is spent in communicating rather than computing.

When these algorithms are simple, writing them to communicate efficiently is also simple whether it is in assembly language or in SEP. However, when they become more complicated, implementing them completely in assembly language becomes near impossible and even using SEP it becomes difficult to write efficiently.

For example, writing a convolution algorithm using a 3x3 mask is simple whether you're writing in assembly language or in SEP because there is only one way to communicate. Writing a convolution algorithm using a 5x5 mask to communicate efficiently becomes moderately difficult in assembly language and a little trickier in SEP. The real challenge begins when the size of the mask becomes 9x9. With such a large radius of communication, optimizing the communication becomes extremely complicated even in SEP. In this case, trying to minimize the communication by hand would take far too long for it to be practical and mistakes could occur too easily. It would be much simpler to just send each data from one source to one destination every time. However, doing so would greatly raise the number of communication instructions and in the cases of convolution and DCT where the percentage of communication is high, performance would be greatly affected. The most obvious solution to this problem is to allow users to write the algorithm however they want it, and have the compiler eliminate any redundant communication instructions. A diagram of communication paths before and after optimization for a 5x5 convolution is shown in Figure 9.

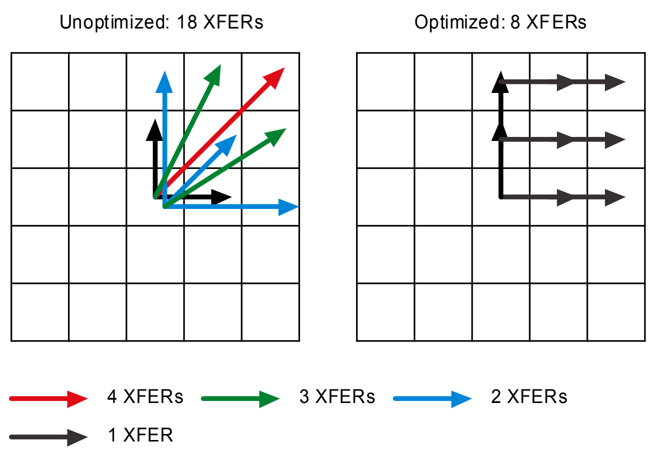

Figure 9 Communication paths before and after optimization

3.4.2 Algorithm for Tracking the Path of Data

The basic idea behind removing redundant communications is to track the path of data through the nodes and see which nodes are visited multiple times by the same data. By overlapping the paths of identical data and comparing the chronological order in which the communication instructions occur in the program, candidates for removal can be identified. Then, certain data items can be selected to be stored in the buffer to maximize the number of communication instructions that can be eliminated. Since all communications are executed simultaneously by all the nodes, data from all nodes travel in the same manner. Therefore, tracking the path of a variable by studying the instructions is identical to tracking the path of that variable on all nodes. A diagram showing the steps taken by the algorithm identify redundant communication is shown in Figure 10.

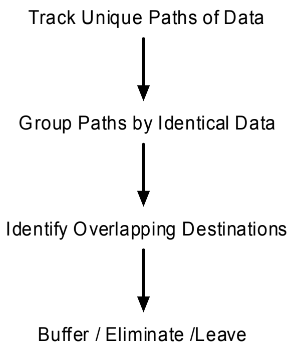
Track Unique Paths of Data
Group Paths by Identical Data
Identify Overlapping Destinations
Buffer / Eliminate /Leave

Figure 10 Steps taken by the algorithm to identify redundant communication

An array of linked list was first constructed to track the movement of data among the nodes using the (X, Y) co-ordinate system. Each linked list in the array is a collection of nodes tracing the movement of a particular variable. Each node in the linked list represents an XFER instruction that was used to send the variable to a neighbor. The northerly direction represents the positive Y axis, the easterly direction represents the positive X axis, the westerly direction represents the negative X axis and the southerly direction represents the negative Y axis. For example, if the first XFER instruction sends the variable to the north, the first node in the linked list would have the position (0, 1). If another XFER instruction sends the variable to the east, the new node added to the linked list would have the position (1, 1). A diagram of the co-ordinate system is shown in Figure 11.

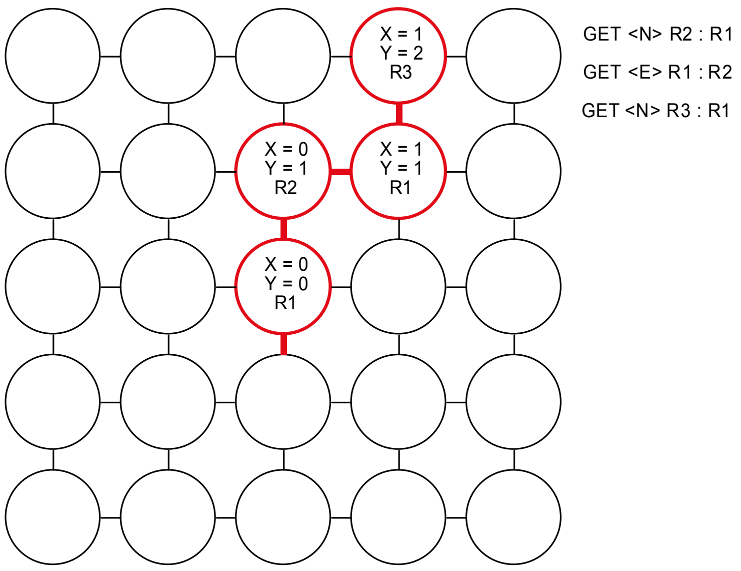

Figure 11 Co-ordinate system with an example trace

The algorithm creates a new linked list whenever a new variable that is not identical to a variable that is already at the end of an existing path is transferred. The algorithm keeps track of these variables that are at the end of any paths in a queue. Any copy instruction that copies a variable in the queue adds the copy to the queue and if any instruction overwrites a variable in the queue, the variable is removed. Copy instructions may be an ADDI instruction with immediate value of 0 or a XFER instruction that copies a variable to different variable in a neighboring node. Therefore, there may be multiple variables in the queue that continues the same path and a path may spread out like a tree. If any path does not have a variable in the queue, that path is closed and will not be extended any further. An example of how this is done is shown in Figure 12.

GET <N> R1 : R2

GET <N> R3 : R1

GET <S> R1 : R4

GET <E> R5 : R3

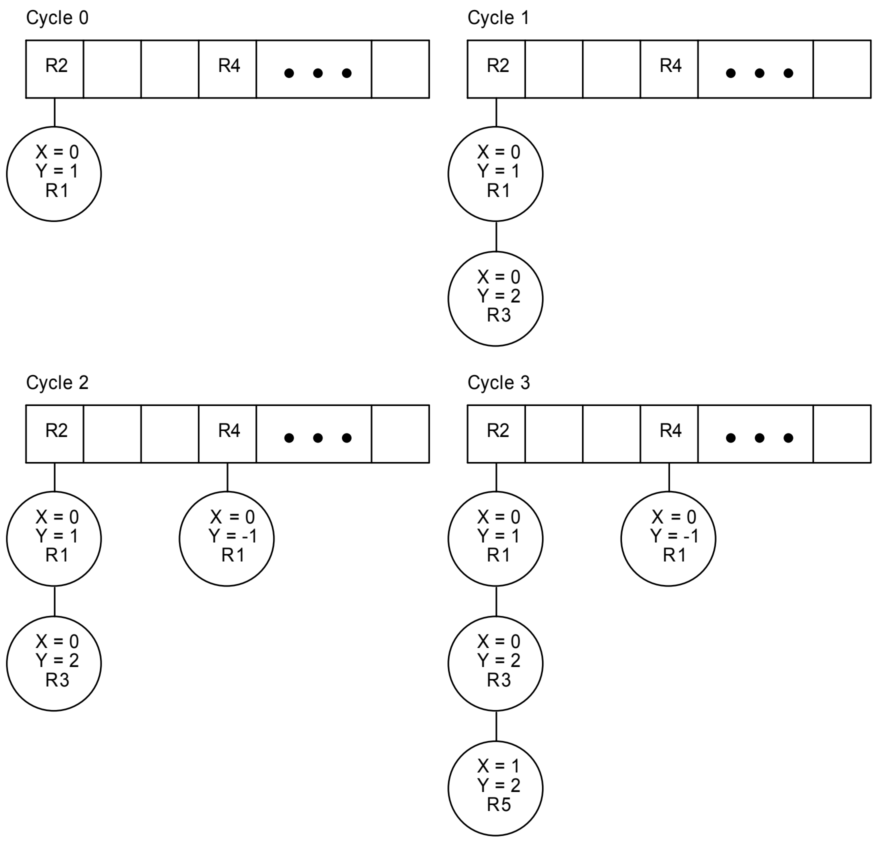

Figure 12 Example of how data movement is mapped by the algorithm

Once all the paths have been mapped out, the source variables of all the linked lists are compared to see which are identical. The algorithm that was adopted for this task was a simple one that checks each variable against another to see if one is a copy of the other or if they are copied from the same variable. A much better, but also much more complicated algorithm such as value numbering would return a smaller number of sets of paths as value numbering identifies not just copies of variables but also variables that have the same value. This algorithm was used as it would suffice for the test programs that would be used to measure the performance of SIMcc. In a future update of SIMcc, a value numbering algorithm will most likely replace the current algorithm to improve performance.

Once the sets of paths that are of variables with the same data are established, the paths are then re-organized to identify the overlapping PEs more easily. The algorithm takes each set of paths and creates a 2-D linked list. Each 1-D linked list represents a particular destination that any of the paths in the set traverses and each node in the 1-D linked list represents a particular XFER instruction that sends the data to that PE. The nodes are arranged chronologically, that is in an ascending order with respect to the instruction number. The algorithm constructs this data structure by going through every PE in all the paths in the set and checking its (X, Y) co-ordinates. If the same destination already exists in the data structure, then a new node is created and added to the appropriate place in the corresponding 1-D linked list. If that particular destination has not been encountered, a new 1-D linked list is created for that destination. A diagram depicting the data structure is shown in Figure 13.

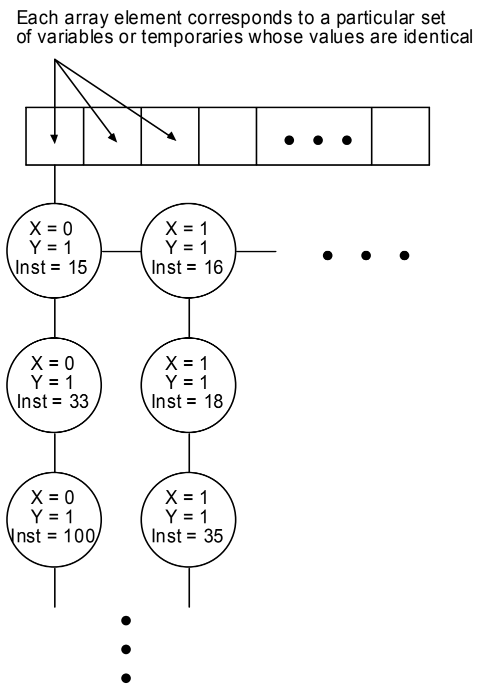

Figure 13 Data structure containing information about overlapping PEs in the paths

Once all these data structures have been established, the IR can be traversed again to see which data items would be stored in the buffer and which would not be in order to minimize the size of the buffer while maximizing the number of XFER instructions eliminated

3.4.3 Algorithm for Buffering and Replacement

Since every XFER instruction can potentially be buffered and the buffer will often be full, a decision will need to be made at every XFER instruction on which data to replace, if any at all. The total number of ways in which to allocate data to the buffer for an entire program is so large that it is not feasible to check through them all to see which yields the largest number of XFER instructions eliminated. However, the different decisions that can be made at every XFER instruction can be formed in to a decision tree, where the tree branches out whenever a XFER instruction is encountered and a choice of which data item in the buffer to replace is made. The problem can then be presented as a search problem where the tree can be traversed or searched in a particular way to find a solution. A small part of the decision tree that can be constructed is shown in Figure 14.

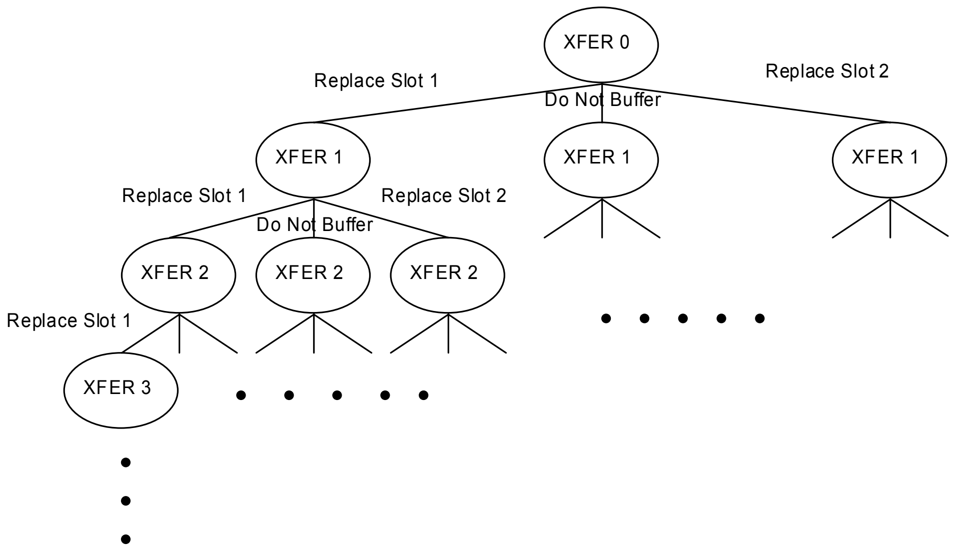

Figure 14 Part of the decision tree that can be constructed for buffer allocation

The tree that is formed from any program would have varying depths at different terminal nodes as some sets of decisions would yield more elimination of XFER instructions than the others. This would suggest a breadth first search to be the best method of finding the solution as the path with the smallest depth would be optimal. However, since the size of the buffer determines the number of paths at each node, even if the buffer size is as small as 4, the number of nodes that must be checked at each level increases to such large numbers that checking all of them would be too time consuming to be practical after 10 or so XFER instructions.

Since every path leads to a solution a depth first search can also be used. However, since only one particular path would be explored with a depth first search, determining whether the path that was taken is the optimal path would be more difficult. To find a good solution to the problem using a depth first search algorithm, factors that influence the number of XFER instructions that can be eliminated must be found and decisions must be made based upon those factors. The solution can now be found using a hill climbing algorithm which is a depth first search with a heuristic measurement for ordering the choices and taking the choice with the best chance of yielding the optimal solution [18].

In order to maximize the number of XFER instructions eliminated, each data that is stored in the buffer in a node must be received by the node as many times as possible before the data is replaced. Since the program has already constructed a data structure that keeps track of all the XFER instructions that send a particular data to a particular destination, it can easily determine how many XFER instructions can potentially be eliminated by storing that data in the buffer for a particular number of cycles. The objective is to eliminate as many XFER instructions as possible and therefore storing data that can eliminate more instructions is desirable. Therefore the algorithm ‘looks ahead’ to see how many instructions can be eliminated by storing a particular data in the buffer.

Whenever the algorithm used in SIMcc for buffering and replacing data encounters a XFER instruction, it checks the buffer to see if that data is already in it. If the data already exists in the node, the XFER instruction is not needed, and the data in the buffer can be used instead. Otherwise, the algorithm checks the buffer to see if there is any empty slot that can be used. If no such slot is available, it checks the buffer to see which data eliminates the fewest number of XFER instructions from that point in time onwards. If the data item in question eliminates more instructions than the data in the buffer that was found to eliminate the fewest instructions, then that data replaces the one in the buffer, otherwise, nothing is replaced and the XFER instruction is executed.

Whenever a piece of data is stored in the buffer, it is merely copied to a newly created temporary variable. If a XFER instruction is eliminated by that data in the buffer, all occurrences of the destination variable of the XFER instruction between the XFER instruction and the instruction that next kills it, is replaced by the temporary variable. After all modifications to the IR has been made by the buffering algorithm, the temporaries that were used as a buffer slot is allocated by the register allocator. In essence, each buffer slot is a common register that is not being used. This method is more efficient than storing them in a specified location like the memory or in pre-determined registers, as most programs do not use many registers simultaneously and accessing the registers is much faster than accessing the memory.

SIMcc also has an algorithm that replaces data items randomly and an algorithm that replaces the oldest data item, thereby using the buffer as a FIFO queue. These algorithms were implemented to see how well the hill climbing algorithm performs compared to them.

CHAPTER 4

RESULTS

Several small applications including 8x8 DCT, 5x5 convolution and 9x9 convolution algorithms were created in SEP in order to measure how well SIMcc eliminates communication instructions using the buffer. The sizes of the three programs in SEP and their sizes in assembly language after being compiled by SIMcc without buffering, and their execution cycles are shown in Table 1. The sizes of the programs in SEP and assembly language are those after loop unrolling has been performed on loops with communication instructions.

Table 1 Program sizes and execution times for various applications

	SEP (lines)	Assembly (lines)	Execution Cycles
5x5 Convolution	131	348	800
9x9 Convolution	543	1205	2721
8x8 DCT	456	1431	4173

Then, the programs were compiled with buffering in effect, with three different heuristics for buffer replacement: look-ahead, random and FIFO. The compiled assembly program was executed on the SIMPil simulator to measure the effectiveness of the heuristics used to eliminate the communication instructions.

4.1 Percentage of Eliminable Communication Eliminated

Each of the example programs executes a certain number of communication instructions. Among them, certain instructions are absolutely necessary in order to distribute the data to all the destinations that need the data. For example, in a 5x5 convolution, each node needs to send its data to 24 other nodes surrounding it. Therefore, the first 24 XFER instructions that send the data to the 24 surrounding nodes are compulsory, and all others are eliminable, that is, not necessary and therefore could be eliminated if buffered perfectly. Percentage of XFERs eliminated as compared to the total number of eliminable XFERs for varying buffer sizes for 5x5 convolution, 9x9 convolution, and 8x8 DCT are shown in Figure 15, Figure 16, and Figure 17 respectively. Each graph compares the performance of the three different heuristics that were used for replacement.

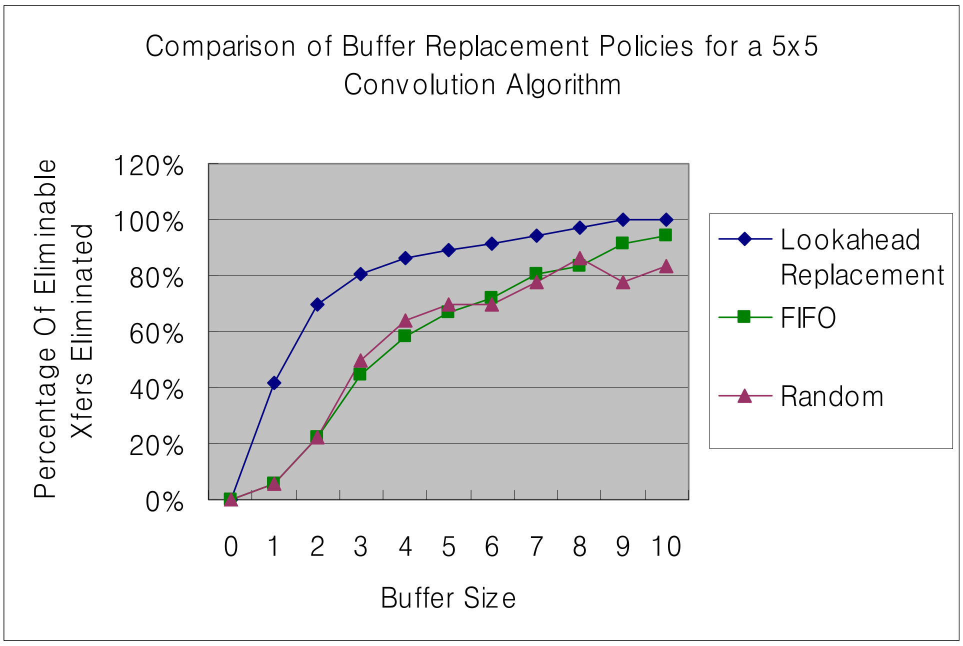

Figure 15 Effect of buffer size on percentage of eliminable XFERs eliminated for a 5x5 convolution algorithm

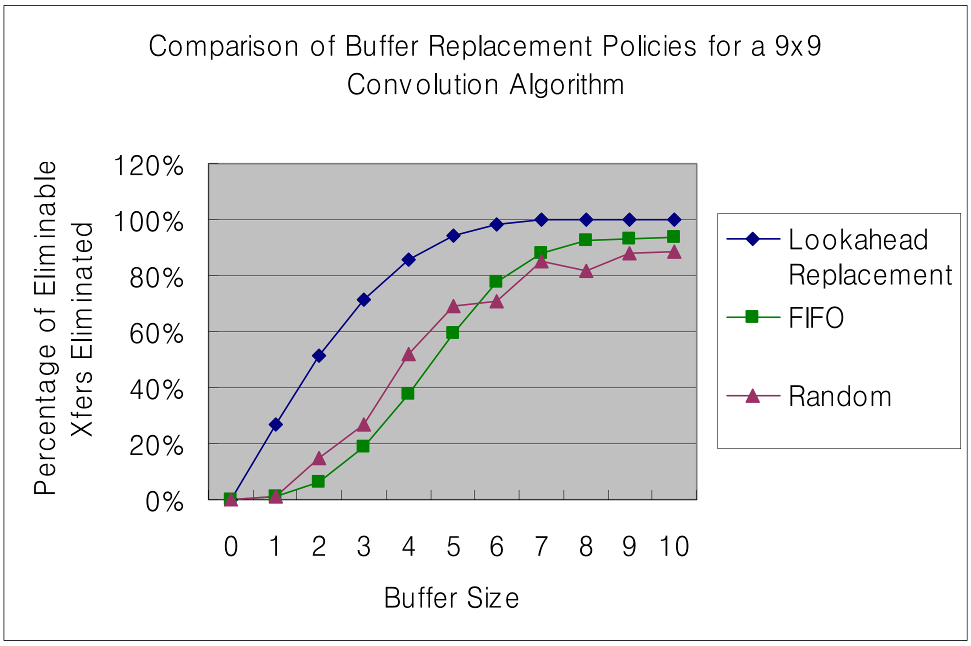

Figure 16 Effect of buffer size on percentage of eliminable XFERs eliminated for a 9x9 convolution algorithm

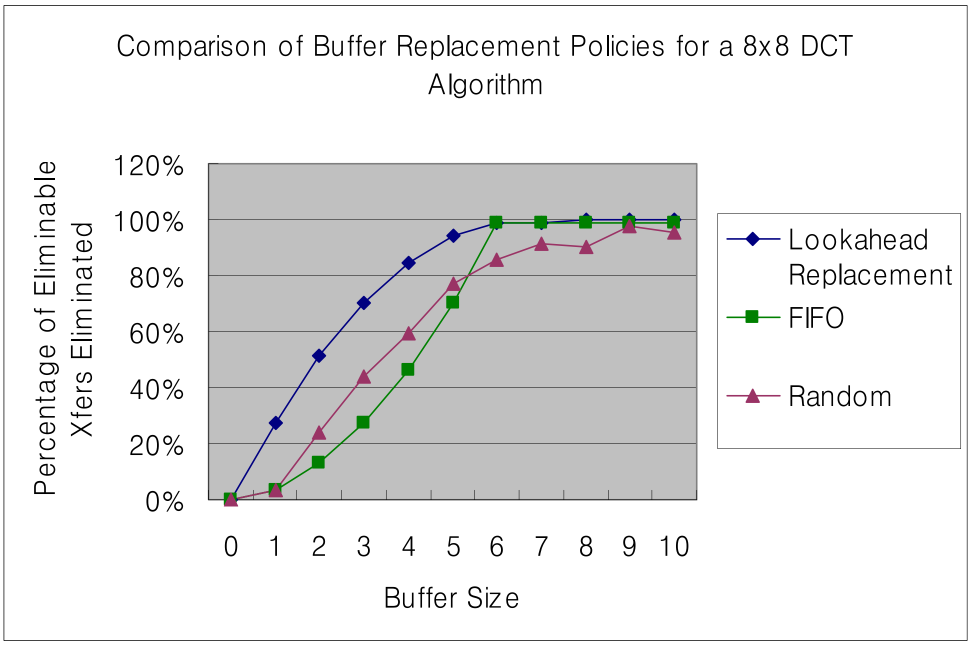

Figure 17 Effect of buffer size on percentage of eliminable XFERs eliminated for a 8x8 DCT algorithm

4.2 Percentage of Total Communication Eliminated

Percentage of XFERs eliminated as compared to the total number of XFERs, including the mandatory XFERs, for 5x5 convolution, 9x9 convolution, and 8x8 DCT are shown in Figure 18, Figure 19, and Figure 20 respectively. Again, each graph compares the different heuristics used for replacement.

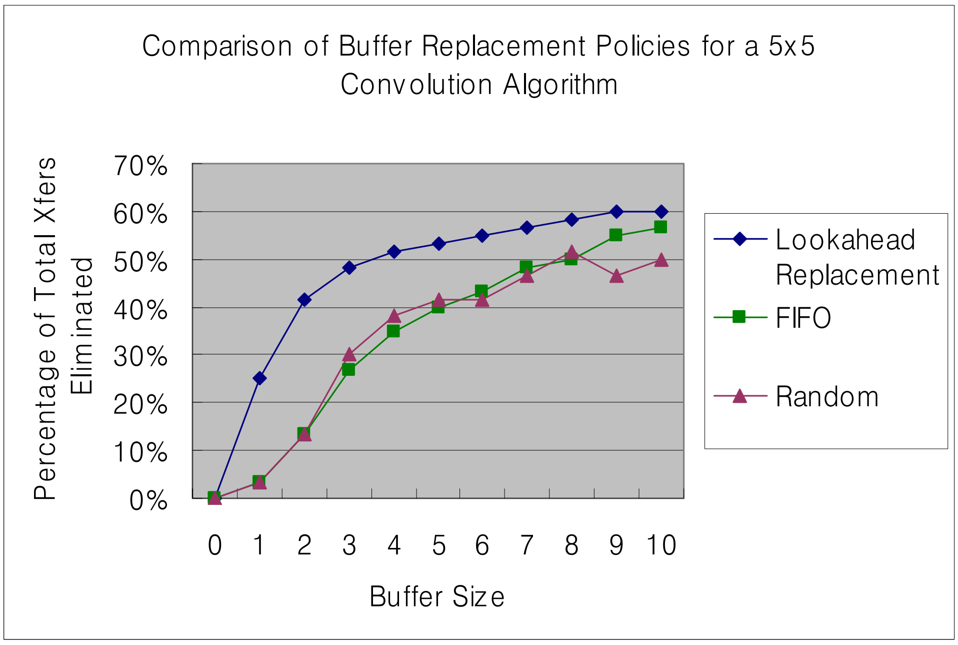

Figure 18 Effect of buffer size on percentage of total XFERs eliminated for a 5x5 convolution algorithm

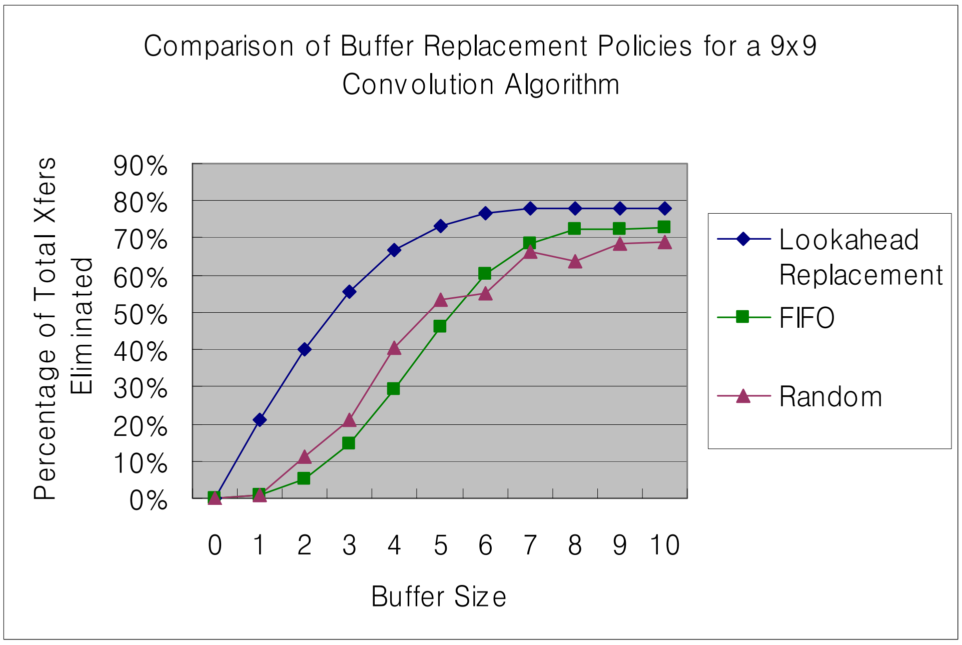

Figure 19 Effect of buffer size on percentage of total XFERs eliminated for a 9x9 convolution algorithm

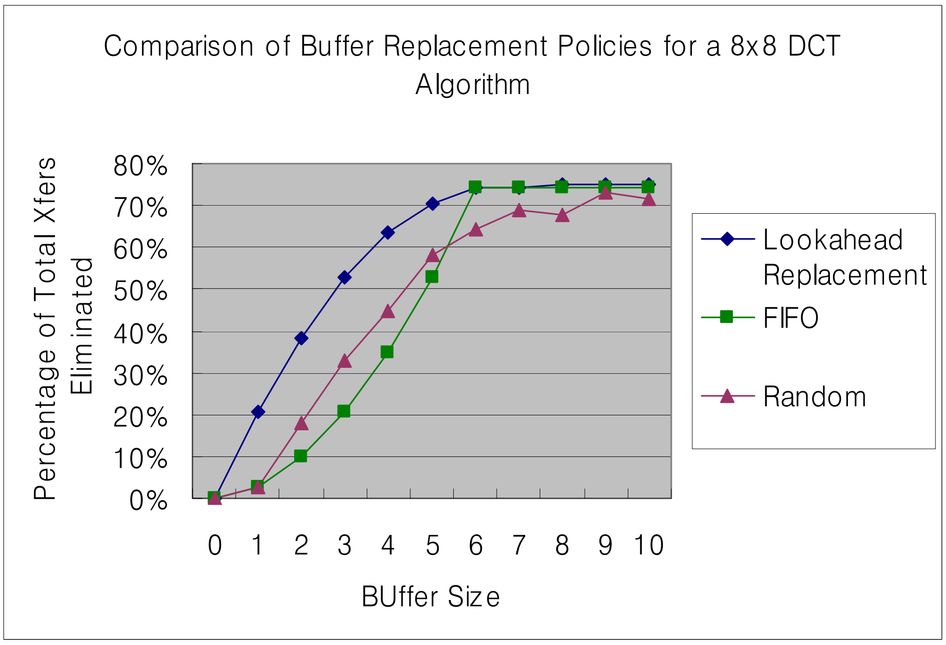

Figure 20 Effect of buffer size on percentage of total XFERs eliminated for a 8x8 DCT algorithm

It can be seen from the various graphs that using the look-ahead replacement method performs significantly better than random replacement or FIFO replacement. For the look-ahead replacement method, the number of XFERs eliminated grows quickly as the number of slots increases from 1 to 4, and then it starts to level, hitting the maximum at around buffer size of 9 for 5x5 convolution, and earlier at buffer size of 6 for 9x9 convolution and 8x8 DCT. As the buffer gets larger, enough slots are available to accommodate all data items to eliminate all eliminable XFERs.

By studying the graphs, it can be concluded that the optimal size for the buffer would be either 4 or 5. Although a buffer size of 6 or larger that maximizes the number of XFERs eliminated is desirable, the gain in performance is too small to warrant the use of the extra registers that would be required, as that could potentially result in very costly spills in to the memory.

4.3 Performance Speedup

The ability of the buffer on eliminating XFER instructions and the effect of the buffer size on performance has been studied so far. However, every time a piece of data is buffered, a copy instruction is created, increasing the instruction count. Therefore, the effect of the buffer size on the overall execution time of the program is slightly different. Every time data is placed in the buffer, at least one XFER must be eliminated to break even. The effect of buffering on the number of instruction count is smaller when considering the overall execution time. The effect of buffer size on execution time for 5x5 convolution, 9x9 convolution, and 8x8 DCT are shown in Figure 21, Figure 22, and Figure 23 respectively. Each graph compares the effectiveness of the three different heuristics used for replacement.

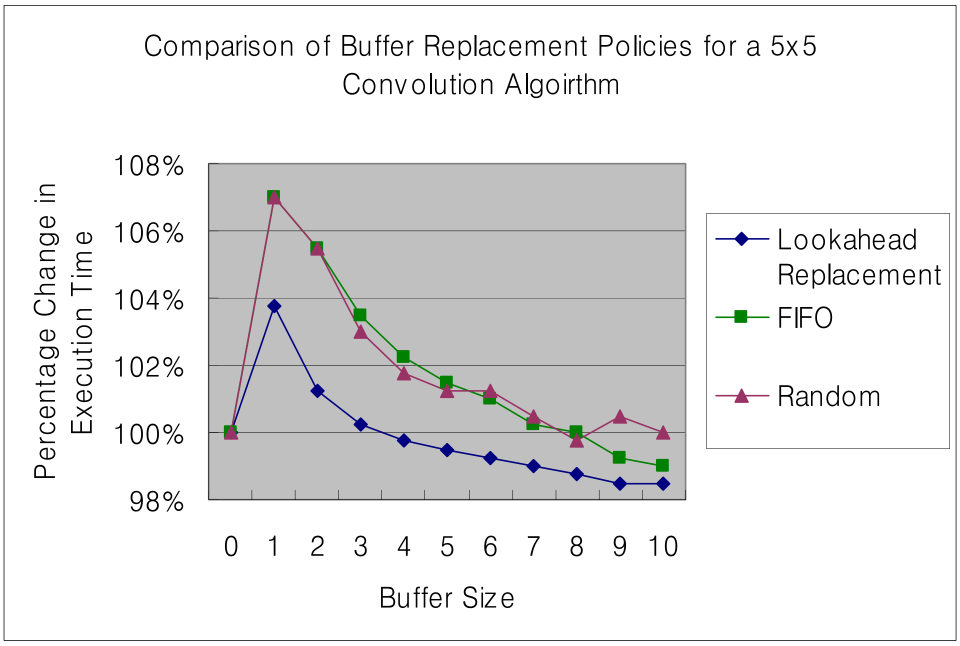

Figure 21 Effect of buffer size on execution time for a 5x5 convolution algorithm

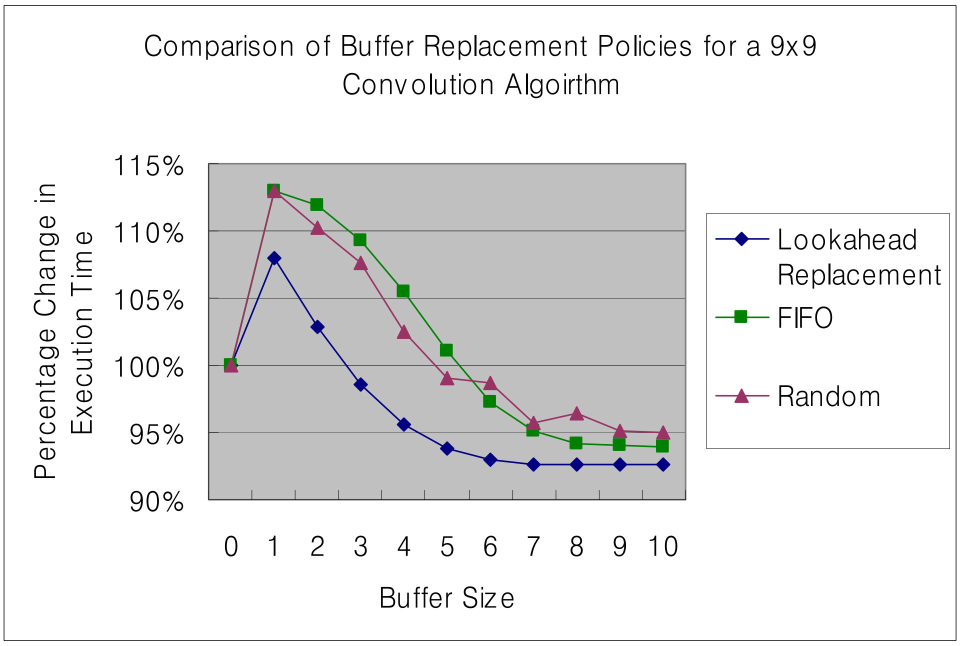

Figure 22 Effect of buffer size on execution time for a 9x9 convolution algorithm

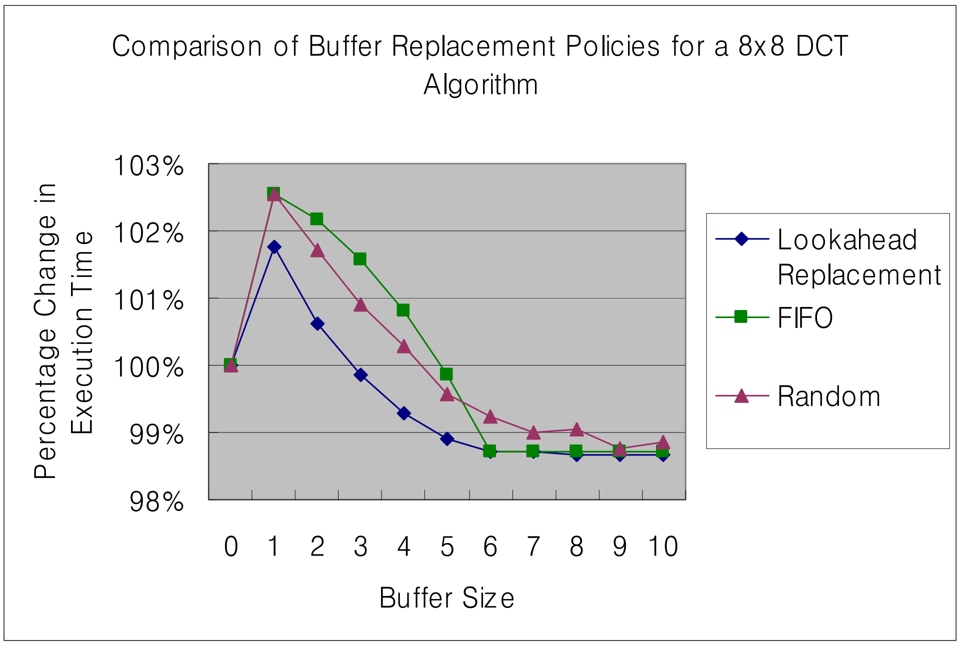

Figure 23 Effect of buffer size on execution time for a 8x8 DCT algorithm

The results indicates that more instructions are generated than are eliminated when the buffer size is small. Performance gain can be seen only after the buffer size exceeds three. For 5x5 convolution and 8x8 DCT, even after the buffer size exceeds seven or eight when the maximum number of eliminable XFERs is eliminated, the percentage of overall execution cycle saved barely exceeds 1%. Slightly better result can be seen from 9x9 convolution, where about 4% reduction in execution time can be observed with a buffer size of four, 6% at a size of five, and a maximum of 7% at a size of six.

The poor performance gains that can be observed from 5x5 convolution and DCT are due to high execution time as compared to the amount of communication cycles. This is mainly due to overhead instructions that were necessary. In the convolution programs, there are instructions that store the mask in each node and in the DCT program, more than half the cycles are spent dividing the mesh in to 8x8 grids. Therefore, when these algorithms are executed multiple times, as they are done in larger applications, the observed performance gains will be higher, as the overhead instructions will still only be executed once, increasing the percentage of communication cycles.

4.4 Effect of Look-ahead Distance on Performance

Another aspect of the buffer algorithm that was studied was the effect of the number of instructions the algorithm looked ahead to see how many XFERs could be eliminated by having that data in the buffer. The effect of the look-ahead distance on the total number of XFERs eliminated for different algorithms is shown in Figure 24. Buffer size of 4 was used.

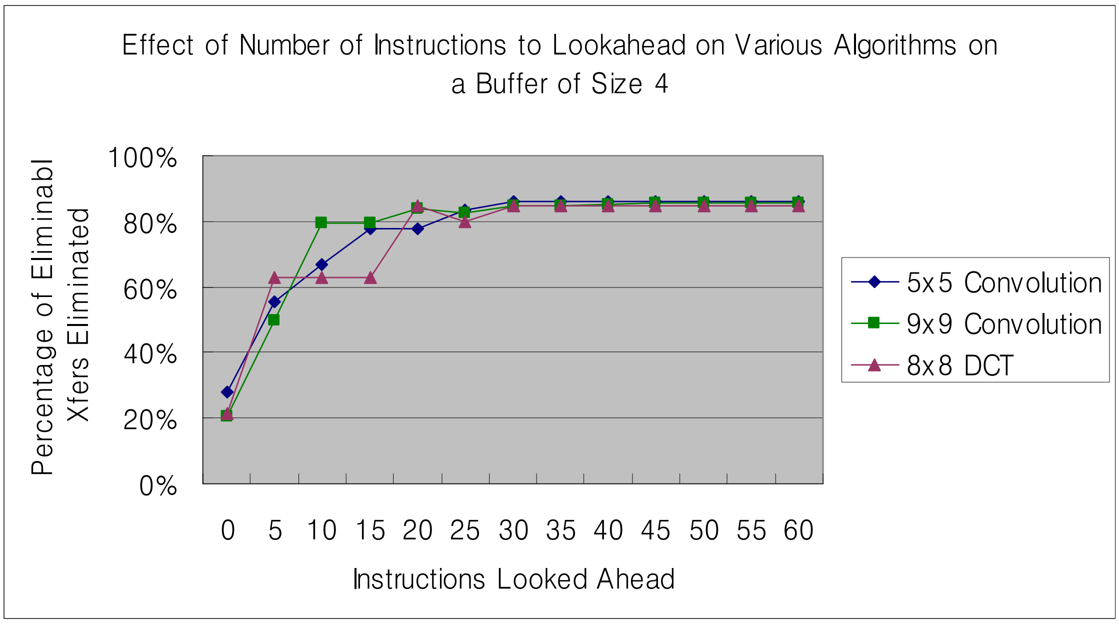

Figure 24 Effect of look-ahead distance on percentage of eliminable XFERs eliminated

It can be seen from the graph that although the performance does vary a little with smaller look-ahead distances, the performance generally increases by increasing look-ahead distance, and levels out at a maximum after a certain number. Therefore, it must generally be a good idea to just check the total number of XFERs a piece of data could potentially eliminate from that point in the program till the end of the program.

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

The SIMcc compiler is capable of eliminating a large percentage of redundant communication. However, performance improvement in execution cycles will not be seen unless a large percentage of the instructions are communication instructions. In fact, using buffering degrades performance if there are too few communication instructions or if the communication has already been fully optimized with no redundant instructions.

The look-ahead algorithm performed significantly better than either random or FIFO replacement policies, and the look-ahead algorithm was able to eliminate all redundant communication instructions at a smaller buffer size. The optimal size for the buffer was found to be 4 or 5 at which 86 and 89 percent of redundant communication was eliminated respectively for 5x5 convolution. For the same buffer sizes, 86 and 94 percent of communication instructions were eliminated for 9x9 convolution, and 85 and 94 percent were eliminated for 8x8 DCT. After 4 or 5, the buffer hit the point of diminishing return but still eliminated all communication by 7 or 9. No performance gain in terms of execution cycles was seen with 5x5 convolution or 8x8 DCT as the percentage of communication was too small to affect the performance, but with 9x9 convolution, four and six percent reduction in execution cycles was seen with buffer sizes of four and five respectively.

This version of SIMcc is basic and there is much room for improvement. The buffering mechanism could be improved further by computing the number of registers available at each point in time and using only the ones available to increase the buffer size without spilling into the memory. Better data grouping algorithms like value numbering can be used to minimize the data sets and maximize the number of instructions each buffered data can eliminate.

In future updates of the compiler, more common optimizations could be implemented for better performance and more language constructs can be added for wider programming options. Currently, the compiler assumes a PPE (Pixels per Element) of 1. In a real system this may not suffice as some images may be too large for the processor array to support a PPE of 1. The compiler can be re-designed or modified to hide the PPE from the programmer and automatically detect the optimal PPE to maximize processor usage and generate instructions accordingly.

Since energy is also an important issue for portable architectures, analysis on energy usage could be done in order to see if there are any advantages or disadvantages of using buffers, as buffering may reduce the network usage, but it does increase processor and register usage.

REFERENCES

[1] G. Chaitin, "Register Allocation and Spilling via Graph Coloring," ACM SIGPLAN Symposium on Compiler Constructions, 1982, pp. 98-101.

[2] A. Aho, R. Sethi, and J. Ullman, Compilers Principles, Techniques, and Tools, Addison-Wesley, 1988.

[3] L. Tucker and G. Robertson, "Architecture and Applications of the Connection Machine," Computer, Volume 21, Issue 8, August 1988, pp. 26-38.

[4] P. Christy, "Software to Support Massively Parallel Computing on the MasPar MP-1," Compcon Spring '90, 'Intellectual Leverage', Digest papers, Thirty-Fifth IEEE Computer Society International Conference, 26 Feb-2 Mar 1990, pp. 29-33.

[5] R. Bargrodia and K. Chandy, "Programming the Connection Machine," Computer Languages, 1988. Proceedings., International Conference on, 9-13 Oct 1988, pp.50-57.

[6] M. Gokhale and J. Schlesinger, "A Data Parallel C and its Platforms," Frontiers of Massively Parallel Computations, 1995. Proceedings. 'Frontiers 95'., Fifth Symposium on the, 6-9 Feb 1995, pp. 194-202.

[7] S. Ranka and S. Sahni, "Convolution on Mesh Connected Multicomputers," pattern Analysis and Machine Intelligence, IEEE Transactions on, volume 12, issue 3, Mar 1990, pp.315-318.

[8] A. Fisher, J. Leon, and P. Highnam, "Design and Performance of an Optimizing SIMD Compiler," Frontiers of Massively Parallel Computation, 1990, Proceedings., 3rd Symposium on the, 8-10 Oct 1990, pp. 507-510.

[9] V. Garg and D. Schimmel, "Hiding Communication Latency in Data Parallel Applications," Parallel Processing Symposium, 1998. 1998 IPPS/SPDP. Proceedings of the First Merged International…and Symposium on Parallel and Distributed Processing 1998, 30 Mar-3 Apr 1998, pp. 18-23.

[10] J. Choi, C. Lin, H. Kim, "Low Power Register Allocation Algorithm using Graph Coloring," TENCON 2000, Proceedings, volume 3, 2000, pp. 80-85.

[11] A. Watson, "Image Compression Using the Discrete Cosine Transform"

[12] J. Lee and K. Batcher, "Minimizing Communication in the Bitonic Sort," Parallel and Distributed Systems, IEEE Transactions on, volume 11, issue 5, May 2000, pp. 459-474.

[13] A. Gentile and D. S. Wills, “Portable Video Supercomputing,” to appear in *IEEE Transactions on Computers*, 29 pages, TC 114531, accepted February 2004.

[14] G. Costas, “Explicitly Parallel Compiling for the SIMPil Architecture,” Pica Group Technical Report, 1998.

[15] H. H. Cat, A. Gentile, J. C. Eble, M. Lee, O. Vendier, Y. J. Joo, D. S. Wills, M. Brooke, N. M. Jokerst, A. S. Brown, and R. Leavitt, “SIMPil: An OE Integrated SIMD Architecture for Focal Plane Processing Applications,” Proceedings of the Third International Conference on Massively Parallel Processing Using Optical Interconnections, pages 44-52, Maui, Hawaii, October 1996.

[16] B. Kahle and W. Hillis, “The Connection Machine Model CM-1 Architecture,” Systems, Man and Cybernetics, IEEE Transactions on, volume 19, issue 4, Jul/Aug 1989, pp. 707-713.

[17] J. Nickolls, “The Design of the MasPar MP-1: A Cost Effective Massively Parallel Computer,” Compcon Spring ’90. ‘Intellectual Leverage’. Digest of Papers. Thirty-Fifth IEEE Computer Society International Conference, 26 Feb-2 Mar 1990, pp. 25-28.

[18] Patrick H. Winston, Artificial Intelligence, Addison-Wesley, 1992

[19] International Technology Roadmap for Semiconductors 2003

[20] J. D. Meindl, J. A. Davis, P. Zarkesh-Ha, C. S. Patel, K. P. Martin, P. A. Kohl, “Interconnect Opportunities for Gigascale Integration,” IBM J. Res. & Dev. vol. 46, pp. 245-263, Mar/May 2002.

[21] J. W. Joyner, R. Venkatesan, P. Zarkesh-Ha, J. Davis, and J. D. Meindl, “Impact of Three-Dimensional Architectures on Interconnects in Gigascale Integration,” IEEE Trans. Very Large Scale Integration Systems, vol. 9, pp.922-928, Dec. 2001.