A large/fine-grain parallel dataflow model and its performance evaluation*

by BEHROOZ SHIRAZI
Southern Methodist University
Dallas, Texas

and

ALI R. HURSON
Pennsylvania State University
University Park, Pennsylvania

ABSTRACT

Data driven architecture has been widely proposed in literature as an alternative to the von-Neumann design to handle real time and fifth generation applications. However, the network delays at the fine-grain dataflow level and handling of large arrays are some of the problems which should be addressed in these architectures. In this paper, we introduce a new model for dataflow computation which yields itself to an efficient realization of both static and dynamic dataflow architectures. Furthermore, the proposed model provides grounds for efficient handling of arrays in an SIMD fashion. Some implementation issues, the VLSI constraints, and architectural support for the model are discussed. The proposed organization achieves parallelism at the program block level (large-grain parallelism), instruction level (fine-grain parallelism), and data level (array processing). The system behavior is studied through a probabilistic simulation model and the conclusions are presented.

*This research was supported in part by the Department of Defense under Contract 5-25089-310
INTRODUCTION AND BACKGROUND

The recent real time applications demand a computational power of the order of 1 billion operations per second. However, it has been proven that neither the conventional von-Neumann type computers nor the traditional concurrent systems can offer such a computational power. This computation gap is a result of factors such as: (1) the technological constraints imposed by the physical laws, (2) the sequential nature of the von-Neumann architecture, and (3) the software/hardware complexities introduced by the traditional concurrent systems. These deficiencies encourage the design and implementation of systems which are inherently parallel. The data driven computations provide the basis for such an organization. In this environment, a program is represented as an acyclic directed graph in which the nodes are operations to be performed and the arcs direct the data among the nodes. The concept of asynchrony embedded in the definition of a data driven architecture provides grounds for a high degree of implicit parallelism. In addition, the data driven organization eliminates the need for an updatable storage, use of identifiers, and all of their associated by-products such as global side-effects and aliasing. Such a radical departure from the sequential von-Neumann type organization has eliminated familiar concepts such as the program counter, addressing schemes, central memory, etc., with an eye towards increasing the degree of parallelism.

However, the traditional fine-grain (instruction level) data driven computation leads to the increased cost and complexity of the network, or otherwise erecting a potential bottleneck due to delays of the network for token transmission. Therefore, the large-grain data driven architectures have recently been investigated as an alternative for their fine-grain counterparts. This has led to the development of the block driven systems which explore the parallelism at the program block level.

In addition to the network problem, the elimination of the global variables and addressing mechanisms enforces the tokens (instruction and data) to be self contained (i.e., they must carry a large volume of information in order to efficiently utilize the processing power). At the implementation phase, this violates the pin limitation constraints imposed by the current technology. Finally, the lack of updatable storage and asynchrony in the operations requires special procedures and mechanisms for handling of the data structures. The proposed data driven architectures in the literature have attempted to overcome these problems by shortening the interconnection paths and by developing new algorithms for handling the data structures and I/O operations.

A viable data driven architecture should comply with the technological constraints, offer a better performance for inherently parallel problems, reduce fine-grain communication costs, and introduce a practical and effective solution for the manipulation of the data structures. These criteria have led us to the introduction of a new model for data driven computations capable of supporting array processing. This paper discusses the proposed model, addresses the architectural support for the model, and provides a performance evaluation of the system.

THE PROPOSED MODEL

The dynamic dataflow machines allow simultaneous execution of several activations of the same block. This provides a higher potential for parallelism at the expense of more complexity. On the other hand, the static dataflow architectures provide simplicity with less parallelism. The advantages of these two classes of dataflow machines have led us to the introduction of a new model for the data driven architectures. Our goals are simplicity of the design, reduction of communication requirements, and capability to handle arrays efficiently. These goals can be achieved by distribution of the processing power among the memory cells.

In this scheme, each memory cell holds an instruction and its input operands. The system has enough processing power to perform token matching for each instruction and to carry out the operation within the cell. It is noticeable that the simple, cellular architecture will be suitable for VLSI implementation. Also, the communication requirements are reduced since the tokens only need to go through one level of networking as opposed to two or three, as in the static and dynamic models. Furthermore, the cells can be viewed as an array of processors which can be programmed in an SIMD fashion for manipulation of array structures.

Figure 1 depicts a detailed version of the proposed Processing Module (PM). The input and output ports provide communication between the PM and the outside world. The instructions of a program block are assigned to the cells, one instruction per cell. Each cell independently detects its firing condition, executes the instruction, and routes the result token to its destination cell(s) via the Sub-net. It is obvious that fine-grain data driven computation is achieved with minimal communication requirements.

A PM operates as a static data driven machine. Thus, the tokens will not carry any coloring information related to the recognition of the block activation. However, a number of these PM's can be used to efficiently emulate the dynamic data driven model through code duplication. As depicted in Figure 2, the system has a large-grain block driven organization in which program blocks can be simultaneously executed by simply duplicating the code and assigning it to a free PM.
The coloring information need only be kept at the program block level and individual tokens need not carry this information.

It should be mentioned that the system supports enough parallelism to tolerate network delays. For this reason and because we are concentrating on the model, not implementation, we did not specify any particular interconnection scheme for the networks used in Figures 1 and 2. Naturally, one will take advantage of the advances in interconnection technology to use the most cost efficient network during the implementation. In addition, the choice of the type of application programs to run on the system can affect the communication requirements. We assume that the application programs processed by the proposed system will be functional and block oriented with a high degree of locality of effect and virtually no global variables. These characteristics imply that the independent program blocks can be executed in parallel and that the degree of communication among the instructions of a block is much higher than the inter-block communication. Therefore, we need a relatively high speed interconnection network for the sub-net (see Figure 1), while the speed requirements of the system network (see Figure 2) is not critical.

The Host Module

The host module holds the program blocks as they are generated by the compiler. It performs system management tasks such as: detection of the firing condition for a block, keeping track of the free processing modules, and allocation of the enabled program blocks to the processing modules. The task allocation and load balancing is performed dynamically based on the firing condition in a block driven environment. In other words, as soon as a task (function or block) is enabled (fired) for execution, it will be assigned to one of the processing modules chosen from a pool of available PM's. The firing condition consists of the availability of the input arguments for a function (block) in a functional programming environment. The assumption is that the functions are strict, requiring all the inputs to be available before firing of the block.

Context switching may be used to increase the processing power utilization. Thus, if an executing function becomes inactive (e.g., due to calling another function), its current image is stored in a high speed memory in the host and the free PM can be assigned to another enabled function. The inactive block will be reactivated whenever it receives its requested item. A completely different scheme may also be employed in which an inactive block remains idle in a PM until it receives the requested data. These two schemes are compared in our simulation and the result will be discussed later.

The system management tasks are facilitated by labeling (coloring) the program blocks. A block label is a tuple \((\alpha, \beta)\), where \(\alpha\) is the static part assigned to the block during the compilation and \(\beta\) is the dynamic part determined during execution. Obviously, \(\beta\) is used to recognize different activations of the same block during execution. For example, when a called function completes, the \(\beta\) part of the label of the calling function is used to determine to which activation of \(\alpha\) the value must be returned.

The Data Structure Module

This module is used to hold the data structures and provides a smooth interface for their manipulation. The model is flexible enough to allow implementation of heaps as in
structures as in, or array structures defined in. However, one can take advantage of the functional programming style and single assignment rule to increase opportunities for parallelism. A function consumes its input arguments and produces output results. Therefore, a data structure is duplicated with a proper label (e.g., appending the dynamic label of the block to the data structure identifier) for each function to which it is passed as an input argument. However, upon the completion of the function, the input structures are deleted. The functional programming style and block driven firing conditions ensure serialization of the dependent functions which update the same data structure.

The cost of the structure duplication is justified by the reduced cost of the memory chips and the fact that the duplicated structures only exist temporarily. It should be noted that the dataflow concept has been traditionally applied to the scientific and numeric application domains in which the size of the data structures is often limited. For applications requiring the manipulation of very large data structures, the model yields itself naturally to a virtual memory organization, with the data structures stored on a secondary storage device.

Duplication of the data structures for the blocks provides opportunities for parallelism among them. However, within a block the single assignment rule is used to avoid the write-after-write problem. According to this rule, a data structure element may be assigned a value only once. The read-before-write problem can be avoided by using a tag bit associated to each element. If the tag is set, then the element is updated and the read can proceed. Otherwise, the read should be queued and checked for processing after each data structure update.

This method was first introduced in for handling of the I-

The Processing Modules

It is obvious that the implementation cost of the model will become prohibitively expensive if the cells (refer to Figure 1) are complex. Therefore, we envision the cells to be simple processors called Elementary processing Units (E-unit) which can perform operations such as addition, subtraction, AND, OR, etc. The more complex operations are to be routed to coprocessors which are a collection of Functional Units (F-units). Thus, the cells of a PM are divided into E-units and F-units. The E-units are our basic cells to which the instructions and their input data are assigned. The flow of operations in an E-unit is presented in the Petri-net of Figure 3. As shown
in this figure, an E-unit matches the input tokens, performs simple operations, and forms operation packets for complex operations to be routed to the sub-net. An F-unit is simply a hardware implemented functional unit which performs a specific operation on the input operation token and sends the result to the corresponding E-unit (indicated by the input token) through the sub-net.

Granted the removal of the technical problems in Wafer Scale Integration (WSI), we suggest the processing modules be implemented on silicon wafers. WSI uses the entire wafer, instead of dicing, to condense more functionality into a single device. The intra-block communication among E- and F-units is, therefore, improved by eliminating the delays and difficulties associated to the multichip systems. However, there are some problems with WSI (heat removal, yield, etc.) which require further research and development in this area.

As previously mentioned, the sub-net can be any \( n \times n \) interconnection network with a reasonable performance. We intend to investigate a variety of networks, beginning with an arbitration network. The choice of the arbitration network stems from performance and implementation issues. Dias and Jump have shown that arbitration networks with buffering capability between the stages can match the performance of an equivalent crossbar network. In addition, the simple structure of these networks allows their easy implementation by WSI technology (i.e., crossovers and long parallel communication lines are avoided).

The sub-net, as an arbitration network, funnels the output tokens from the E-units and F-units in a pipeline fashion. The funneled tokens are either sent to the outside world via the output ports or distributed among the E-units and F-units through a common data bus. The PM is loaded via the input port in a bit-parallel word-serial pipeline method.

The Host and Data Structure Modules

The main function of the host module is to dynamically manage (allocate/deallocate) the program blocks during the execution using the dataflow concept as the primary task allocation principle (i.e., a function is assigned to a free processing module when it receives its input arguments). Our assumption, at this point, is that the compiler decomposes the program into blocks which best match the size of a PM. Therefore, a large logical block will be decomposed into several sequentially executable smaller physical blocks. If experimentation indicates that this scheme serializes the program extensively, we can extend the model by providing local memories for each PM. In this case, large blocks will be stored in the local memory of a PM and are processed segment by segment through some paging scheme.

The system management functions could be facilitated by a set of hardware tables which determine the status of the system during its operations. For example, the block assignment table determines the status of a block (passive, active, or inactive), while the firing condition table keeps track of the input arguments to a block and detects the firing condition for it. To improve the performance, the operations on the status tables are performed in associative fashion. This has two advantages: (1) the search of the tables are performed in parallel, and (2) we can have overlap of operation for different system operations.

The host module’s memory has a high-order interleaved organization in which different program blocks are assigned to different memory modules. This allows simultaneous loading/unloading of blocks among several PM’s. In this particular architecture, the host and the PM’s are organized in a master-slave organization with direct connections between the host and each of the PM’s. This is because of the need for block transfers between the host and PM’s, and due to the limited intra-block communication in a functional programming environment. However, as we will discuss later, we are considering other protocols such as a distributed host environment.

The data structure module is composed of a Data Structure Memory (DSM) and a Data Structure Processor (DSP). The DSM holds the data structures used in the data flow programs as well as the temporary data structures which are created to ensure the parallelism among different blocks. The interleaved organization of the DSM allows simultaneous access to many elements. The DSP provides smooth access to the data structures, manages the data structure memory, and initiates the execution of vector processing instructions.

**The System VLSI Complexity**

In order to comply with the current technology in the design of the proposed system, modularity has been explored at three levels (3-dimensional modular system). First, the architecture is composed of a few building blocks which are duplicated many times across the system (e.g., processing modules). Second, each processing module is composed of a group of basic building blocks (e.g., E-units and F-units). Third, the organization of each E-unit and F-unit is composed of a number of basic cells which are duplicated in a two dimensional space (see Hurson and Shirazi, for example). The VLSI time and space complexities of the proposed system have been evaluated through the analysis of its major components. A detailed discussion of such analysis is out of the scope of this paper, but can be found in Hurson and Shirazi. Table I summarizes the geometry area and the timing delay of some of the system major components.

**THE SYSTEM PERFORMANCE EVALUATION**

The purpose of this performance evaluation is to study the behavior of the system and to recognize the weak points and the bottlenecks. It is not our intention to run complete programs on the simulator and compare it against the existing data driven machines; although such a study is currently underway and the results will soon be available. As such, the system simulation is based on an event-driven model which passes tokens among different resources. The token generation and routing is based on a probabilistic model reflecting the program characteristics such as the degree of parallelism in a block, execution delay of a block, number of function calls, and the type of data structure operations.
TABLE I—The VLSI characteristics of the proposed architecture

<table>
<thead>
<tr>
<th>Unit</th>
<th>Geometry Area</th>
<th>Timing (ps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>E-unit</td>
<td>4mm×4mm</td>
<td>250</td>
</tr>
<tr>
<td>Sub-net</td>
<td>40mm×20mm</td>
<td>150</td>
</tr>
<tr>
<td>F-unit (Multiplier)</td>
<td>10mm×10mm</td>
<td>97</td>
</tr>
<tr>
<td>Host Module (Arithmetic Memory)</td>
<td>10mm×5mm</td>
<td>10</td>
</tr>
</tbody>
</table>

1. λ=2.5 μm.
2. Average inverter delay=0.3 ps.
3. 32-input, 4-output arbitration network, routing 64 bits in parallel.
4. Assuming no conflicts while going through the network.
5. Reference 18.

The timing delays are based on either the VLSI timing analysis of the designed components or timing information obtained for the existing units. For example, loading of a block takes \( \log_2 n \Delta t \) to fill the arbitration network pipeline plus \((n-1)\Delta t\) to go through the pipe, where \(n\) is the number of cells in a PM or the block size, whichever is less, and \(\Delta t\) is the delay of an arbitration switch in the sub-net.

In order to obtain realistic statistics about the execution delay of a function and its degree of parallelism, we could not rely on a probabilistic simulation model. Therefore, an emulator was written which could mimic the operations of a processing module. We were then able to write actual dataflow programs (small hand compiled functions) and run them on the emulator. The results are presented in Table II.

The system simulator would then build a complete program, running in parallel on different PM's, from these functions. The program blocks are generated by randomly selecting one of the functions and augmenting it with some global program characteristics such as function calls, data structure operations, and enabling other functions at the completion of the current function. For example, in one experiment, the number of function calls in a block is uniformly distributed between 0 and 4, while the block may enable from 0 to 3 other blocks at the end of its completion (called functions are treated differently; they only return a value to the calling block).

Figure 4 depicts a plot of the MIPS (Millions of Instructions Per Second) performance of the system against the number of processing modules. The number of E-units in a PM is fixed at 32, while the MIPS figures presented are the average of the results collected from a number of simulations. It is noticeable that the performance saturates relatively fast and addition of the PM's does not improve the MIPS. Although not presented here, the processor utilization study also reflected a similar behavior (i.e., the PM utilization decreased as the number of PM's was increased).

Our first attempt to seek the origin of the problem was to study the loading/unloading of the blocks. The assumption was that the blocks can be loaded/unloaded in parallel from/to different modules of an interleaved memory in the host. However, instead of a bit-parallel word-serial scheme (as in the previous case), we employed a bit-serial word-parallel method. As depicted in Figure 5, this change did not improve the performance (i.e., there was no statistically significant difference). Therefore, we focused our attention toward the suspicious bottleneck, namely the host manager. In the next study, we reduced the host module overhead by a factor of 5. The results of this experiment are depicted in Figure 6. Notice that performance has improved by a factor of 2 to 5, but saturates around 64 PM due to overwhelming host overhead. As a result, we are currently studying a variation of the model in which the host tasks are distributed among a number of sub-hosts and each sub-host controls a number of PM's in a tree fashion. This model allows a better fault tolerance and scalability compared to the original model.

In the above experiments we used a context switching scheme to improve the processor utilization. In other words, an idle block is removed from a PM and reinstated whenever it receives the requested data such as function call or SIMD type array operation. If we relax the high processor utilization

<table>
<thead>
<tr>
<th>Program</th>
<th>Average degree of parallelism during execution</th>
<th>Program run time (GHz)</th>
<th>MIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quadratic equation</td>
<td>1.01</td>
<td>25.0</td>
<td>0.44</td>
</tr>
<tr>
<td>Standard deviation</td>
<td>1.30</td>
<td>29.2</td>
<td>0.53</td>
</tr>
<tr>
<td>Simple data transformation</td>
<td>1.48</td>
<td>33.4</td>
<td>4.74</td>
</tr>
<tr>
<td>( x^n, n=3 )</td>
<td>1.27</td>
<td>7.2</td>
<td>3.89</td>
</tr>
<tr>
<td>( x^n, n=10 )</td>
<td>1.24</td>
<td>23.7</td>
<td>4.47</td>
</tr>
<tr>
<td>( \int_0^1 x^2dx )</td>
<td>1.06</td>
<td>21.2</td>
<td>3.07</td>
</tr>
<tr>
<td>( \sum_{i=0}^{24} i )</td>
<td>0.68</td>
<td>9.7</td>
<td>2.47</td>
</tr>
</tbody>
</table>

TABLE II—The processing module emulation results, running actual dataflow programs (number of E-units = 32)
requirement, an idle block may remain in a PM until its re-activation conditions are satisfied. To avoid a deadlock problem, if all the PM’s are occupied, we will begin preemption of the inactive blocks in a PM in a first-in-last-out basis. The major advantage of this scheme is that it reduces the number of block transfers between the host and PM’s, and thus, reduces the host module overhead for setting up the blocks for transmission.

The simulator was modified to reflect the new policy “no context switching as much as possible” and the results were very encouraging. The MIPS performance was improved from 45% in case of 16 PM’s to 100% in case of 64 PM’s, reaching a performance of more than 200 MIPS. The processor utilization was reduced by about 40% in case of 16 PM’s to about 10% in case of 64 PM’s. Therefore, one can double the speed at virtually no processor utilization cost when the number of processing modules is large, more than 64.

CONCLUSION

This paper has introduced a new model for dataflow architectures based on the data driven (fine-grain parallelism) and block driven (large-grain parallelism) principles. The model is flexible enough to support both static and dynamic dataflow computation models as well as vector processing operations. It also provides opportunities for matching the underlying architecture with semantics of the functional dataflow languages. The model behavior was studied through the simulation of a system which represents an implementation of the model. The results indicated that a central host module can be the bottleneck and thus, a distributed control over the processing modules is more desirable. It was also concluded that given a large number of processors, in excess of 64, it is more advantageous to leave the inactive blocks in a processing module and only reallocate an idle module when there are no free processing modules available.

REFERENCES