A Reconfigurable Data Flow Machine for Implementing Functional Programming Languages

Christophe Giraud-Carrier
Brigham Young University, Department of Computer Science, Provo, UT 84602
cgc@axon.cs.byu.edu

Abstract
Functional languages do away with the current state paradigm and achieve referential transparency. They also exhibit inherent parallelism. These qualities fit very well on top of a data-driven architecture such as a data flow machine. In this paper, we propose a fully reconfigurable data flow machine for implementing functional programming languages. The design is based on smart memories and nodes interconnected via a hypercube. Important aspects of the proposed model are described and compared with other similar attempts. Advantages of our system include massive parallelism, reconfigurability, and amenability to higher-level, graphical programming. Current limitations are identified and extensions are suggested.

1. Introduction
Most current computation schemes, programming paradigms, and computer architectures are inspired by the pioneering work of Von Neumann. However, this computational model has failed to satisfy the ever-increasing need for greater efficiency in very large scale programs. Recent research has focused on overcoming what has become known as the Von Neumann Bottleneck.

The Von Neumann machine is a current state machine, where the current state is defined as the set of all memory values (including registers) at any given time. Any operation performed on the machine (including NOP) alters its state by modifying its memory content (e.g., increment the value of the PC register). This behavior is sometimes referred to as a form of programming with side-effects, where the side-effects are the changes in state caused by the execution of an instruction. Another concern with the Von Neumann scheme is the physical bottleneck caused by the physical separation of memory from processing units, that results in constant transfer of data from memory to the processor and back to the memory. Improving the current model implies finding an alternative to these important drawbacks.

Functional programming and data flow machines seem to offer such an alternative. In this paper, we propose a fully reconfigurable, massively parallel data flow architecture for implementing functional programming languages. After a brief review of the basic concepts of functional languages and data flow computing, we present the architecture of our system. We then address the issue of mapping program structure (processes) onto the architecture (processors), and describe how programs execute. Important issues raised by the development of our system are discussed. Insights are provided as to how it compares with other proposed models. Examples of applications are given and limitations discussed.

2. Functional Languages and Data Flow Computing
Functional languages abstract a program as a black box, or a function, that takes some inputs, and from them produces an output. There are no variables. Consequently, functional languages are free of side-effects and do not have any explicit notion of current state. All computed values are intermediate results that serve as inputs to other functions. The results of functions are time-independent, that is, a function's value is only contingent on the values of its arguments, and not on when it executes. The only constraint on computation is that a function's arguments must be evaluated before the function itself can be evaluated. However, there is no definite order in which the arguments are to be evaluated. This property is sometimes referred to as referential transparency. It gives rise to very natural parallelism since all the arguments of a function can potentially be evaluated in parallel. Much work has been done to develop functional languages that make easy the automatic extraction of inherent parallelism. In [3] for example, Friedman and Wise propose a functional language whose semantics is aimed at using massive parallelism. The programmer is provided high-level constructs that can be automatically recognized by the compiler and translated into efficient parallel code. An excellent discussion of functional programming languages concepts and evolution is available in [6].

Data flow computing is based upon data flow graphs. Elements of data are transformed as they flow through the graph. Nodes in the graph represent computation units which, from their inputs, produce an output. The output of a node serves as an input to subsequent nodes in the graph. A node fires as soon as all of its inputs are available. Data flow computing is therefore data-driven (i.e., functional). A thorough study of issues in data flow machines is in [13].

One of the challenges with data flow computing is its need for reconfigurability. Each program executed on a data flow machine requires its own configuration of nodes. If no reconfiguration is possible, the data flow machine is bound to be fully dedicated to a given application (possibly a small set of applications matching the graph), and
almost every application would then require its own machine. More flexibility is usually expected and the trend is towards general-purpose machines capable of handling a wide variety of applications without having to alter the hardware structure.

The underlying philosophy of functional languages based on black box computing maps nicely onto the data flow structure in which each node can be thought of as one black box implementing one of the functions in the program. The architecture we propose is motivated by this close relationship between functional programming and data flow computing.

3. A Reconfigurable Data Flow Machine

In this section, we give an architectural view of our machine. In the following sections, we justify it and evaluate it in more details. The main issues addressed in the design of the architecture are: massive parallelism, recursion, reconfigurability, and recomputing vs. caching.

3.1. General Overview

Three essential building blocks constitute the underlying architecture of our machine: smart memories, nodes, and a complex interconnect. A pool of nodes is connected to a pool of smart memories via the complex interconnect, as sketched in Figure 1. The various data paths needed for program execution are established at compile-time, based on the program's data flow graph and the underlying hardware interconnection network. Each one of the building blocks is detailed in the following three sections.

3.2. Smart Memories

The purpose of the smart memories is to hold previously computed results (i.e., cache). In our system, we use a simplified version of the more general smart memories introduced in [8]. Their internal structure is described in Figure 2. The descriptor memory is associative and associative descriptor processing is used (see [8]). One smart memory is associated to each numeric function type in the program, and is used to cache the results of computation of that function. Each descriptor contains the values of the inputs and the corresponding output. The memory processor is programmed with microcode to efficiently manage the descriptor memory. In particular, it contains instructions for fast (associative) searches.

Smart memories are chosen (over conventional caching mechanisms) because their encapsulation of processing at the memory level gives us the flexibility to fit the search algorithm to the properties of the function whose results are being cached. For example, if the function associated to the smart memory is commutative, the memory processor, when asked to search for an entry (a,b) should also do a search on the entry (b,a) in case (a,b) is not found. This allows to store only one form of the input thus saves space in the descriptor memory.

3.3. Nodes

A node (or set of nodes) is bound to the invocation of a function. Nodes can connect to one smart memory and to a number of successor nodes. The internal structure of a node is shown in Figure 3.
Inputs are queued to accommodate the data-driven computational model. A node detects when new inputs are available to it, via an extra E (i.e., Enable) bit for each input register. The E bit is set when a value is written to the corresponding input register, and cleared as soon as the node fires and uses that value. As soon as a set of inputs is available (i.e., all E bits are set), the node is enabled and fires. Section 4 describes the details of firing.

The node's processing unit is the actual computational unit of the model. Each node can perform an atomic or built-in function of its inputs and write the result to its buffer. The processing unit is fully programmable, that is, it has the ability to store instructions and execute them on demand. It can be thought of as a very small CPU embedded in the node structure.

The connection to the smart memory (to/from SM) is for caching purposes, and the incoming result arrow is used when the node is a controller-node as described in Section 4. Each node also holds the addresses of its successors. An extra bit is associated with each address to allow controller-nodes to distinguish between successors in the program's data flow graph (to whom computed outputs are sent) and successors in the substructure implementing the user-defined function (to whom inputs are forwarded).

3.4. Interconnect

The interconnect consists of a hypercube, in which nodes and smart memories are interspersed. Because of the great flexibility of connection and communication available in the hypercube, any one of the smart memories can potentially be logically connected to any one of the nodes. The same is true of logical connections between nodes. There are as many smart memories as there are nodes and they are spread across the hypercube in such a way that every node has a smart memory reachable with a path involving at most two physical links (this configuration is critical to caching efficiency).

We arbitrarily limit the number of inputs per node to four, thus limiting the width of the physical data paths. This decision seemingly forces any function to have at most four arguments, which may appear as a restriction. However, most commonly used functions (e.g., car, +, etc.) require only two arguments and any user-defined functions that would require more than four arguments can easily be decomposed in two or more functions of at most four arguments. Providing more inputs would most likely be an overkill, result in extensive waste of register space at the node level, and make unreasonable (in terms of implementation) the width of the data paths in the hypercube. There is a definite trade-off between logical flexibility and physical requirements. We feel that four inputs is a reasonable compromise.

Nodes communicate via a simple rendez-vous mechanism. Since nodes hold the addresses of their successors, it is straightforward for a node to send the value of its output to all of its successors in the program's data flow graph.

3.5. Data Tagging

An important issue raised by functional programming is that of recursion. Most functional programs make heavy use of recursion in their computations. This results in the same function being invoked many times from its inside with different arguments at each stage of the recursion. For the data flow machine to handle recursion, a mechanism is provided for a node to know how to use its inputs so that its computation is performed on arguments belonging to the same function invocation.

Each piece of data that flows through the system is tagged with a value reflecting the invocation it belongs to. Watson and Gurd [14] proposed a similar approach on the Manchester Data Flow Prototype to allow re-entrant code structures. However, their system is based upon a ring architecture with a global matching store which holds all the tokens (i.e., data items). Tokens with the same label must be matched, and once all of them are found, the next instruction is fired. We opt for having the tag be part of the data item flowing through the system, allowing the search for a match to be local to every node rather than global. This local search is done on fewer data elements and is therefore faster, thus improving the overall speed of computation.

4. Putting the System to Work

For programs to take advantage of the architecture, a compiler is developed that maps logical data flow graphs onto efficient physical implementations. In this section, we show how functions are mapped to nodes, give an example for a simple function, and briefly describe the execution of the machine.

4.1. Function-Node Mapping

The hypercube structure allows various communication paths to be established between nodes, thus supporting reconfiguration. An effective way of dynamically gaining nodes and adding them to a growing structure is found in the rendez-vous mechanism. An alternative guaranteeing more locality is to use a wave allocation mechanism. A complete discussion of node interconnect and allocation in hypercubes is available in the work of Hillis [5] on the Connection Machine.
Our system uses wavefronts to logically connect smart memories to nodes ensuring physical proximity, and consequently, fast transfer of data between node and smart memory. This locality is a necessary condition to the efficiency of the caching mechanism that we discuss presently. Locality for nodes is not so critical. However, we desire fast interconnection set up, as well as an equal-level performance regardless of how many nodes are currently being used and how many are free. The rendez-vous mechanism guarantees us both. To allow efficient exchanges between nodes and high parallelism in communications, nodes are able to handle the various routing mechanisms described in Hillis's Connection Machine [5], in particular, referral and forwarding.

The basic mapping of functions to nodes is handled by the compiler. As it scans the program's source code, the compiler performs the following:

1. Each time a new atomic (i.e., built-in) function is encountered:
   a. Allocate a node for the function and connect it to a smart memory
   b. Encode the function directly into the node's processing unit
   c. Program the smart memory's processor (use knowledge)
2. Each time a new user-defined function is encountered:
   a. Derive the function's data flow
   b. Set up a controller-node and connect it to a smart memory. (This controller-node does not perform any computation per se. Its role is to get the function's arguments and pass them down to the nodes responsible for the actual computation. It also holds the result of the function and referees the caching/recomputing mechanism described below.)
   c. Physically set up the data flow implementing the function and connect it to the controller-node. The output of the last node points back to the controller-node's buffer.
3. For each invocation of an existing atomic function:
   a. Allocate a node and copy the existing one's contents into it
   b. Connect the node to the associated smart memory (same as first node)
4. For each invocation of an existing user-defined function
   a. Make a copy of the initial controller-node
   b. Connect the controller-node to the associated smart memory
   c. Make a copy of the data flow implementing the function and connect it to the controller-node
5. Keep track of the overall structure of the complete program's data flow and establish the remaining connections between nodes, letting each node know of its successors.

Atomic (built-in) functions (e.g., addition, car, etc.) are assumed to be known to the compiler along with the code that implements them. Built-in functions will be located in single nodes while user-defined functions will cause several nodes to be organized into a data flow graph which connects to a controller-node. When setting up each smart memory, the compiler programs the memory processor so that it takes advantage of the knowledge it may have of the function whose values it holds, in order to implement efficient search through the descriptor memory. Note that knowledge about the function's properties could be made explicit in the program (e.g., use special keywords).

4.2. A Simple Example

Suppose we were to implement the simple program of Figure 4, written in Lisp-like language. Following the steps of the compiler as outlined in section 4.1 results in the interconnection network of Figure 5. The dotted lines mark the interconnections that the compiler derived from the program's data flow graph and the user-defined function structure.

(defun myfunc (a b c)
  ((+ a (* b c)))
  (+ (myfunc a b a) b))
4.3. Program Execution

Once the configuration of the application at hand has been set up, the computation can start. Nodes are at the heart of the computational scheme. When a node recognizes that all of its inputs are available, it fires. Inputs are sent to the associated smart memory to check whether or not the result has already been computed and is thus in the cache. In order to increase efficiency, the node also fires. This means that, if the node is a controller-node, it passes inputs down and starts the computation of the data flow graph computing the function. If it is a regular node, it launches the execution of the built-in function coded in its processing unit. It is assumed that cache look-up is faster than function evaluation (especially for user-defined functions). If the result is found in the cache, it is sent to the output buffer of the node and computation is halted. If the result is not in the cache, then the computation proceeds until it completes. Once the result has been computed, it is written to the output buffer of the node and sent to the smart memory along with the inputs that produced it, thus updating the cache. The output is also made available to all of the node's successors in the program's data flow graph (via rendez-vous).

Michie [9] was one of the first to study this issue of caching and recomputing. He argued, based on intuitive arguments, that the use of caching improves function performance. In particular, he distinguished between evaluation by rule (i.e., computation) and evaluation by rote (i.e., look-up), and conjectured that a combination of the two would result in significant performance increase.

5. System Evaluation

In this section, we examine some of the issues raised by the design of our system, compare it to other similar models, and discuss its range of applications, as well as its limitations.

5.1. Important Issues

The most important aspects of our system include: massive parallelism, reconfigurability, support of recursion, and caching/recomputing combination. We discuss each in turn and conclude with a review of the key role played by the compiler and the system's amenability to graphical programming.

a. Massive Parallelism. The proposed system provides massive parallelism. Each function invocation produces its own copy of the function's implementation, thus allowing several unrelated invocations of the same function to be computed in parallel. Though parallelism is increased, the decision to maintain several copies of a function's implementation raises important issues.

In particular, since all copies of a function share the same smart memory, the potential for parallel execution of independent invocations of the same function is likely to be hindered by contention at the smart memory level. Moreover, as the number of nodes to interconnect increases, the hypercube runs into the real-time/real-space problem, and the physical proximity necessary to proper caching can no longer be guaranteed. As long as the hypercube can be kept significantly larger than any application (in terms of nodes), the system will not suffer much. However, the hypercube has a fairly expensive interconnect as is, namely $O(n\log n)$, where $n$ is the number of nodes in the hypercube, and it is not inductive.

Recursion is another important issue affecting the amount parallelism of the system. With a recursive function, the data flow graph loops onto itself, causing one functional unit (i.e., structure implementing a function) to handle more than a single invocation. However, unless we decide to flatten out all recursions into iterative calls with dynamic allocation of smart memories and nodes for each new invocation, this is inevitable. Flattening recursions is very costly and would quickly overrun the physical capacity of the architecture. It is not unusual, indeed, to have a recursive function working on a huge data structure go into great depths of recursion, thus tying up all the resources.

b. Recursion. As discussed above, recursion tends to reduce parallelism, but it is an indispensable feature of any system pretending to support functional languages. Recursion is implemented by the tagging mechanism discussed in previous sections. The originality, and resulting increase in efficiency, of our system lie in the distribution of the matching store among the nodes. High performance is achieved in tag-matching.
c. Reconfigurability. The issue here, is between dedicated and general-purpose machines. Snyder [11] provides an interesting discussion of the issues involved in dealing with reconfiguration and proposes some possible architectures to handle them. Most data flow machines that have been proposed have been based on fixed, non-reconfigurable architectures. Many of them use a circular pipelining mechanism with several matching stores and several functional units, associated together in pairs in a ring-like architecture [4, 13]. Not only is this approach inherently less parallel, it also introduces possible bottlenecks at the functional and matching units. Of course, one could argue that there is a potential communication bottleneck in more complex structures. However, in the case of a hypercube, this problem is tremendously reduced by the referral capability. Communications may require a little more time, but not much more since the range of re-routing possibilities is very wide. The bottleneck becomes almost a non-issue.

Our architecture, because it can be configured on demand, can also allow several programs to run at once, thus increasing the overall amount of parallelism. The concurrent processes could potentially share the same smart memories for identical functions. Of course, this requires a multitasking operating system (and a large hypercube).

d. Recomputing vs. Caching. It is more than likely that in a large program, a function will be invoked several times with the same arguments. If the function is complex and slow to execute, we might wish to have its result made available through caching. Keller and Sleep [7] discuss important issues introduced by caching in Functional Programming.

Counting on the fact that it is possible to design a cache (smart memory in the present system) such that the look-up (including data transfer) is faster than computing, we decided to cache each function type in the program. The cache on atomic functions can be argued to be an overkill since these functions are implemented very efficiently on small CPUs. One way to ensure that the best performance is achieved at all times, is to adopt a combination of computing and cache look-up. In effect, both are carried out at the same time and whoever "comes back" with a result first is the winner. Now, the only argument against caching certain functions is for the purpose of using less smart memories. This might be a valid concern. Yet, the compiler could be built with enough smarts to recognize such very fast functions (e.g., +, and, or, etc.) and skip the step at which a smart memory is connected to the node.

e. The Compiler. As can be seen from our previous discussion, much of our system's value hangs on the efficiency of its compiler. Granted that it is possible to design such a compiler, almost any functional programming language can be implemented on our machine. This may be a big grant.

However, much of the compiler's complexity can be alleviated and the system can provide a flexible platform for programming at a higher level of abstraction. Programs can conceivably be implemented directly from their flow graph, without ever having to write them in a specific programming language. Not only does this relieve the compiler from the difficult task of constructing the program's flow graph from its language representation (steps 2a and 5 in the above description), it also makes the system amenable to more graphical forms of programming. Instead of having to worry about translating a program into instructions in some language, the programmer only needs to graphically specify the program. The compiler then parses the graph and translates it into the appropriate configuration. This higher level of abstraction fits naturally in a functional programming environment.

5.2. Comparison with Other Models

Vegdahl [12] developed a set of criteria with which he meaningfully compared various architectures implementing functional languages. Rather than reproducing his results here, we refer the interested reader to his insightful paper and add our contribution to the list of proposed models in Table 1. Specific data flow architectures of particular interest for comparison with the one presented here have been proposed by Rumbaugh [10] and Dennis [1].

The main advantages of our system over other models are (1) its ability to reconfigure on demand, and (2) its amenability to higher-level, graphical forms of programming. Reconfigurability is achieved by taking advantage of the inherent flexibility (i.e., rich interconnect, referral capabilities, etc.) of the hypercube, and allows the system to be general-purpose, rather than fully dedicated. Graphical programming increases the level of abstraction at which programs may be specified. It is most desirable in a functional programming environment, since functional programs are best described by their flow graphs. In many cases, the programmer actually translates the program's flow graph into the corresponding language statements. Our system does away with this unneeded step, thus effectively (1) decreasing programming overhead, (2) reducing the potential for error (possibly introduced in the translation process), and (3) increasing usability.
### 5.3. Applications and Limitations

Because the system supports only atomic data, but offers a high degree of fine-grain parallelism by permitting all independent function invocations to execute in parallel, it is most suited for applications that require speed and/or are intensively number-crunching, such as mathematical software and real-time simulations. An example of global weather model was implemented on Dennis's architecture using the functional language VAL [2]. A 20-fold improvement in performance was reported for that weather simulation application. Since Dennis's model suffers some of the drawbacks discussed in this paper, and that our system attempts to overcome, we feel confident that our system can achieve, at least similar and probably higher-order of increases in performance.

No attempts have been made at this point to deal with data structures. The issue of handling data structures is in fact far from having found a general solution, as can be seen from Veen's remarks on the subject [13]. However, the caching mechanism and high degree of parallelism of our machine would be very beneficial in applications dealing with large data structures, such as matrices, that need to be extensively manipulated. Given the fine granularity of parallelism implemented, structure copying is too expensive. However, streams might offer a feasible alternative.

Since the architecture does not scale so well and the interconnect is expensive, changes may need to be made to accommodate very large programs. One possibility is to change the organization of the nodes so that a single functional unit handles more than one invocation of the function it implements. This could all be done at the compiler level and would probably make use of the tags. It would reduce the amount of parallelism slightly, but efficiently increase the capacity of the system to host larger programs without changes to the underlying architecture. There is obviously a limit to this process, since too much reuse would cause the system to behave as a sequential machine.

A formal complexity analysis, for both program configuration and execution, still needs to be performed, as a proof of the viability of the system. Also, a simulation program remains to be written, with the accompanying, possibly graphical compiler, to test the kind of performance actually achievable by the proposed architecture. Research in the area of compilers and operating systems is needed to find efficient ways of managing this new environment and taking advantage of it.

### 6. Conclusion

In this paper, we have proposed a reconfigurable data flow machine for implementing functional programming languages. Functional languages do away with the current state paradigm and achieve referential transparency. They also exhibit inherent parallelism. These qualities fit very well on top of a data-driven architecture such as a data flow machine. The proposed architecture attempts to merge these two complementary computing paradigms.

The design is based on smart memories and nodes interconnected via a hypercube. The particularities of our architecture are discussed at length and compared with other existing models. Recursion is supported by a distributed tagging mechanism. Massive parallelism is attained through fine-grain parallelism at the function invocation level. Reconfiguration is guaranteed by the flexibility of logical connections and communication in the hypercube. Improved performance is made possible by combining caching and recomputing. A completely graphical programming environment can be implemented.

Potential applications and limitations of the model are discussed. Even though no simulation is available yet, we feel confident, in the light of what other similar models have been able to realize, that a significant increase in computing performance can be achieved by our model, particularly in the area of numerical applications.

### 7. References