Fault Tolerance in Parallel Implementations of Functional Languages

R. Jagannathan
Computer Science Laboratory
SRI International
Menlo Park, California 94025

E.A. Ashcroft
Computer Science Department
Arizona State University
Tempe, Arizona 85287

Abstract

Computing models for functional language programs not only facilitate automatic exploitation of inherent parallelism, but they also provide for implicit tolerance to hardware faults through temporal and spatial redundancy. In this paper, we argue that fault-tolerance can be achieved more efficiently by using intensional computing models (eduction) rather than extensional computing models (reduction). While intensional computing models can be implemented by using either data-driven execution or demand-driven execution, we show that the latter is naturally suited.

1 Why Fault Tolerance in Computing Models?

Laprie [1] has defined fault tolerance as the ability, by redundancy, to provide dependable service in spite of faults occurring or having occurred. Fault tolerance in parallel implementations is important for two reasons. First, the occurrence of a fault is likely, given the multiplicity of processor, memory, and communication components in a multiprocessor. Second, users expect isolated faults in parallel implementations to be tolerated, preferably automatically, without undue loss of functionality or degradation in performance.

In both multiprocessors and uniprocessors, fault-tolerance techniques have been successfully and widely used at the hardware level. The use of error-detection codes (such as M-of-N codes, parity, cyclic codes) with memory components and the use of self-checking logic with processor components are commonplace [2]. With the increasing complexity of hardware components in uniprocessors, however, hardware fault-tolerance by itself is insufficient to guarantee a dependable system.

The SIFT effort [3] was among the first to provided for fault tolerance at a level higher than hardware, namely, at the architecture level. While architecture-level fault tolerance allows flexibility in hardware design, it is intricately tied to a particular architecture. In addition, with increasingly complex architectures, providing for hardware fault tolerance at the architecture level is difficult and expensive. Therefore, it has become necessary to provide fault coverage at even higher levels of abstraction. This is not only important in providing comprehensive fault tolerance but it allows flexibility in architecture and hardware design.

Techniques for software level fault tolerance, such as recovery blocks [4] and N-version programming [5], exist at a level of abstraction higher than architecture. However, these techniques require the user to *explicitly* program the fault tolerance. Such explicit techniques are only as successful as the user's ability to apply them correctly. Moreover, some software fault tolerance techniques, such as N-version programming, are really designed to tolerate only faults of the programmer, and not of the underlying hardware system.

Be that as it may, the increased complexity of software for multiprocessors places an enormous burden on a user: programs the user writes not only have to create and manage parallelism but also tolerate faults. As a case in point, a multiprocessor program needs to tolerate the effects of the loss of a processor, which is not a concern for a uniprocessor program. Therefore, we believe that it is important not only to provide fault tolerance at the computing-model level but also to render such fault tolerance application-transparent.

Faults of several types can occur at the computing model level of abstraction, including omission faults, corruption faults, timing faults, and malicious faults [6]. We will consider omission faults and corruption faults since they constitute a significant majority of the faults that occur in parallel implementations. An omission fault manifests itself as loss of information. It is due to failure of a component such as processor, communication link, or memory module. A corruption fault manifests itself as corruption of information. It is due to a malfunction such as bits in memory changing arbitrarily or a processor entering an incorrect state. Omission faults are tolerated by temporal redundancy, i.e., recomputing after information loss. Corruption faults are tolerated by spatial redundancy, i.e., simultaneous computation of the same information at separate places.

It turns out that functional languages provide a framework in which application-transparent fault-tolerance is possible in computing models that evaluate functional programs in parallel [7, 8].

The focus of the paper is twofold: First, we show that a certain property of computing models for functional language programs, namely intensionality, allows for efficient fault-tolerance. Second, we show that intensional computing models (which are a superset of dataflow computing models) when implemented using demand-driven execution naturally realize the fault-tolerance.

2 Benefits of the Functional Programming Model

Functional languages have many attractive properties such as programmability and potential for parallelism [9, 10, 11]. Programmability comes from the fact that functional programs are high level and programs solving complex programs can be clearly, elegantly, and succinctly expressed. There is potential for parallelism because the lack of side effects and the absence of imperative "flow of control" mean that explicit sequencing is minimal, and is restricted to situations where it is forced to occur because of data dependencies. Both properties make it attractive to attempt to implement functional languages on multiprocessor architectures, or even to attempt to design custom multiprocessor architectures that can execute functional languages directly.

As soon as a parallel implementation is contemplated for which the number of components is in the hundreds, the possibility, or probability, of hardware and other faults must be considered, and their effects ameliorated. While standard techniques for fault tolerance at the hardware level and software level can be used, we propose that the inherent fault tolerance of functional language programs be harnessed at the computing-model level. This is desirable as it has the advantages of hierarchical fault-coverage, application-transparency, and flexibility in architecture and hardware design.

The benefit of inherent fault-tolerance of functional language programs has been observed by Grit in [12] and by Lin and Keller in [13]. What we show is that the inherent fault-tolerance capability of a functional language program, due to its referential transparency property, can be efficiently harnessed if an *intensional* computing model is used [14, 15]. Moreover, this efficient fault-tolerance can be naturally and effectively realized using a demand-driven implementation [16].

The one main property of functional languages that seems to give opportunity for both spatial and temporal redundancy is referential transparency. Referential transparency ensures that, in general, two occurrences of the same subexpression in a program denote, or will evaluate to, the same value. This property ensures that evaluating an expression several times, even if it really occurs only once, will always yield the same value. (It is easy to see that features that destroy referential transparency, such as side effects and aliasing, destroy this multiple evaluability property.) These multiple evaluations can either occur simultaneously (spatial redundancy) or at different times (temporal redundancy). Spatial redundancy guards against corruption faults, i.e., changes that occur in a value while it is being transferred from one place to another or incorrect values produced by a bad instance of an operation while temporal redundancy guards against omission faults, i.e., complete loss of a value while it is being transferred or the death of an instance of an operation.

This observation about referential transparency is not new. What is original in this paper is the observation that the degree of computing-model fault tolerance can depend crucially on the type of computing model. In particular, computing models that have a property called *intensionality* allow for efficient fault tolerance. In fact, we claim that intensional models, when implemented using demand-driven execution, are *naturally* fault-tolerant. (This claim will be substantiated later.)

To clarify the issues, we present a simple first-order functional language, and describe two computing models for the language, one of which is graph-reduction and the other of which is based upon intensionality. We then show the efficiency with which fault tolerance can be achieved with the intensional computing model, and we indicate the problems faced by graph reduction.

A Simple Functional Language

We describe a simple functional language for the purposes of this paper. Like other functional languages, it is based on Lambda-Calculus [17]. It is actually a simple form of the language Lucid[10].

The language is specified by a simple abstract syntax. The language is an expression language: the programs are simply expressions (*terms*); there are no statements in the language. The language is definitional, and, apart from terms, the other main syntactic category is *definitions*.

Terms are built up using *constants*, *variables*, *operation symbols*, and user-defined *function symbols*. Constants are really nullary operation symbols and variables are really nullary user-defined function symbols. (There are no procedures in the language.) Compound terms, called *where clauses* (or simply *clauses*) are terms with subsidiary definitions. The language is typeless.

Variables are defined as the values of terms, and function symbols are defined in the same way, but they also have *formal parameters* which are variables. The definitions are written as equations.

A *program* is simply a term in which every function symbol that is used is defined in some clause. Variables may be used that are not defined in some clause. Such variables are called the *input variables* of the program. (The value of the term is called the *output* of the program.)

One of the important benefits of a functional program is that it can be given mathematical semantics independent of its various operational semantics. In fact, the mathematical semantics has primacy over all possible operational semantics. For this simple functional language we will give a mathematical semantics based on sequences. This is done in such a way that the first-order language is essentially second-order[18]. Therefore, although the language is a special case chosen to illustrate the link between intensionality and hardware fault-tolerance, it is general enough to represent most functional languages.

To specify the mathematical semantics, we must have meanings for all the constants and all the operation symbols. The meaning of the program is then specified as the minimal fixpoint of all the equations (definitions). A few minor points must be taken care of, such as the fact that the nested clause structure requires a recursive definition of the semantics (the minimal fixpoint solution for a clause requires the minimal fixpoint solutions for the nested clauses), but doing so is completely straightforward. We take the domain or universe (of values for the constants and individual variables) to be the set of finite and infinite sequences of values from the domain of some more primitive or basic algebra. The only unusual feature of the semantics is that most of the operation symbols denote *pointwise extensions of operations on the basic domain*. (By the basic domain we mean the domain of the basic algebra.) For example, if the basic domain includes numerical values and there is an operation symbol for addition, it actually denotes pointwise addition, where the i-th value of the result is the addition of the i-th values of the arguments. (If either argument has no i-th value, neither does the result.) If one of the arguments is a constant symbol, its i-th value is always the same, namely the value that the constant denotes in the basic algebra.

There are a few operation symbols that do not denote such pointwise extensions. These symbols denote operations that can be thought of as manipulating whole sequences. For example the operation symbol *next* denotes the operation that gives the tail of its argument sequence and an operation symbol *fby* (followed by) denotes the sequence formed by putting the first value of its first argument in front of its second argument. These operations allow us to write definitions like

$$x \; = \; 1 \, fby \, (next \; x \; fby \; x + 1) \qquad (1)$$

which defines x to be a sequence whose odd elements are 1, 2, 3, etc., and whose even elements are not specified.

An operational semantics (or a computing model) tells us how the values of programs can actually be calculated. The example definition above should indicate that the computing model has to be able to handle partially defined objects and infinite objects.

3 Extensional and Intensional Computing Models

Computing models for evaluating functional programs can be divided into two categories: extensional and intensional. Examples of extensional computing models are parallel graph reduction [19, 20] and term rewriting [21]. Examples of intensional computing models are eduction [16, 22] and dataflow [23, 24, 25].

Extensional Computing Models

The most commonly used extensional computing model is reduction. While we restrict the following discussion to reduction, it is equally valid for other extensional computing models. Reduction considers the nonpointwise operations as sequence-manipulating operations, such as *cons* for *fby* and *tail* for *next*. Using reduction, programs are viewed as expressions, and the program is evaluated by repeatedly transforming, or reducing, the expression into a possibly simpler expression. The original expression must include any data that the program is to work on, so that at every stage we are manipulating both program and data, and the two become intimately intermingled. The process stops when no further transformation can be applied. At each stage, several transformations may be possible, but it doesn't matter which we apply. We always get the same answer, if we get an answer, but it is possible to make choices so that we do not arrive at the answer. There usually are simple tests to apply, when making choices, to ensure that we are making choices that are leading in the right direction.

Graph reduction is a form of reduction where the expressions at each stage are represented by graphs. This makes sense: for example, if a variable is used (not defined) at several places in a program (and thus at several places in the expression), in the graph representation all these uses can be linked, or point, to one place. Changing what is being pointed to has the effect of making several changes in the expression simultaneously. The graph representation is much more efficient. For our discussion, the important thing to realize is that graph reduction involves building and changing a generally large and complicated data structure.

In the lazy graph reduction computing model, Definition 1 would be considered as

$$x = cons(1,\ cons(head\ tail\ x,\ x{+}1))$$

with $+$ being a lazy coercion, to the sequence domain, of the addition operator on the basic domain. The operation *cons* is completely lazy, and evaluates neither argument until its value is needed. Thus, if a program containing Definition 1, for example, is to give a defined result, the even elements of x must never be needed, but if the odd elements *are* needed, a linear list structure or graph is built up in which every other element (the second, fourth, sixth, etc.) is a subgraph that corresponds to first argument of the second call of *cons* in the program (i.e., *head tail x*). These subgraphs are like suspended computations; they are never developed. The details need not concern us here, but it is important for later discussions to realize that a large connected data structure is built up for x, with much of it consisting of suspended computations.

Intensional Computing Models

The simple functional language described above can be viewed as an intensionally modeled langauage [14] where the word "intensional" is meant to contrast with "extensional." Both words are used in intensional logic [26] which was developed by logicians who were searching for a logic to enable them to reason formally while leaving many things implicit. Such reasoning takes place all the time in natural language where phrases are understood without the time and place of utterance having to be made explicit. An intensional computing model makes sense, in the same way, of programs that do not, at each point, make everything explicit. The model knows, at each point, what the appropriate implicit context is. In fact, in the intensional computing model, all that is known are the values of things in implicit contexts. When evaluating programs containing Definition 1, for example, the model will know the values of x in particular contexts (i.e., at various point) but it will not know what the whole of x is.

This means that an intensional computing model has no such thing as a connected data structure. For a program, containing definition 1, that has a defined result, the needed values of x will be calculated but they will be connected together only abstractly, not concretely. Each value that is calculated will be stored somewhere with a tag indicating which element of the sequence x it corresponds to. Nothing will be stored for the even elements of x, because they are not needed and thus no attempts will have been made to calculate them. The needed values of x can be stored anywhere, even on different machines if the implementation is distributed, as it might well be. There is no monolithic or contiguous structure.

One of the beauties of an intensional computing model is that it allows random access into sequences. If a particular value of the (conceptual) sequence x is required, the tag that is calculated is used to retrieve the value with that tag. There is no traversal of a list to reach the point we want. The tag itself will even indicate on which machine to look if the implementation is distributed. (At an earlier stage, the tag determined, in exactly the same way, on which machine the value would be stored.) No coercion is involved with operations such as "+" which, semantically, are pointwise extensions of basic operations. What we are storing are tagged elements of the basic domain and we just apply conventional addition to elements with the same tag (and return the result, with that tag).

It is characteristic of an intensional computing model that it is concerned with the elements and not the "big picture." That, as it happens, is what makes it more suitable for fault tolerance.

Adding Fault Tolerance to Computing Models

Consider adding temporal redundancy to the lazy graph reduction computing model. The values calculated are lists or graphs. If a value is lost due to an omission fault, the entire list or graph that the value denoted must be recomputed. The recovery time could be substantial depending on the size of the list or graph.

On the other hand, adding temporal redundancy (recomputation) to the intensional computing model is much simpler. There are no structures. There are only basic values with tags. If a value is lost due to an omission fault, only that value need be recomputed. The recovery time would be quite short since the value is basic.

Consider adding spatial redundancy to the lazy graph reduction computing model. If several copies of a variable are being maintained, and if faults are detected by making sure the copy values remain the same as each other, this will involve repeatedly comparing lists, and these may possibly be very long lists. Moreover, if any of the copies is corrupted, an arbitrarily long list may have to be constructed to reestablish the desired relationship between the copies.

Now consider adding spatial redundancy to an intensional computing model. If we are maintaining copies (perhaps the tag will indicate which copy it is) and a value goes bad, all we need do is replace the value by the correct one. If a tag is corrupted it means only that when we look for the copy with that tag we will not find it. (That copy is now garbage; it will never be discovered again except by accident. Collecting up such garbage is an interesting problem that we do not address in this paper.) In this case, a new copy, both value and tag, will be stored. In both cases (corrupted value or tag), the work to do to recover is bounded and is quite small.

The following table summarizes the above discussion.

	Extensional	Intensional
Temporal Redundancy	Inefficient	Efficient
Spatial Redundancy	Inefficient	Efficient

4 Implementation of Intensional Models

Given that intensional computing models lend themselves to fault tolerance, we consider how these models can be implemented to effectively realize their implicit and efficient fault tolerance.

These models can be implemented in two ways: using tagged data-driven execution [27, 28] or using tagged demand-driven execution [16]. With data, an operation can execute if and when its necessary operands are available. With demand-driven execution, an operation can execute if and when its result has been demanded and its necessary operands are available.

Consider the following program

$$z \; where \; z \;=\; h(f(x,y), g(x,y));\; end \qquad (2)$$

which is evaluated using an intensional computing model. (We assume that functions f, g, and h are strict and pointwise.

When the model is implemented using dynamic dataflow, the availability of x_i and y_i cause both $f(x,y)_i$ and $g(x,y)_i$ to be computed and, when these values are available, the value z_i is computed. Now consider what would have happened if the value of $f(x,y)_i$ were lost as a result of an omission fault. The value z_i would not be computed. Even if this fault were detected, recovery would involve the recomputation of x_i and y_i since they were discarded as soon as computation of $f(x,y)_i$ and $g(x,y)_i$ was initiated.

When the model is implemented using demand-driven execution, the evaluation proceeds as follows: First, value z_i is demanded which causes $f(x,y)_i$ and $g(x,y)_i$ to be simultaneously demanded causing x_i and y_i to be demanded twice (but computed only once). When the values x_i and y_i are available, the values $f(x,y)_i$ and $g(x,y)_i$ are computed, resulting in z_i being computed. Now consider, as before, the loss of value $f(x,y)_i$ as a result of an omission fault. The demand for $g(x,y)_i$ would be satisfied but the demand for $f(x,y)_i$ would remain unsatisfied. Simply redemanding the value $f(x,y)_i$ would cause its value to be correctly recomputed using the previously computed values of x_i and y_i. At such a time, the value z_i would also be computed.

Therefore, a demand-driven implementation supports tolerance of omission faults, which is not the case with a data-driven implementation of the intensional computing model. If the omission fault is because of loss of a shared value (such as x_i), the recovery is *naturally* supported by demand-driven execution as illustrated in the following scenario: Computing $f(x,y)_i$ and $g(x,y)_i$ would generate two independent demands for value x_i. If the value x_i disappears after satisfying one of the demands, the other demand, upon not finding value x_i, would cause x_i to be automatically computed from its definition. We discovered this serendipitous benefit when emulating the Eazyflow multiprocessor [29, 7].

Demand-driven execution can be easily augmented to detect corruption faults. Instead of simply demanding $f(x,y)_i$ and $g(x,y)_i$ when z_i is demanded, we can make three redundant demands for each of the two operands. The value of $f(x,y)_i$ is the value that is in the majority, of the values returned as a result of the three redundant demands that were made. (A similar process can be used for $g(x,y)_i$.)

The redundant demands consume resources that could otherwise be used to exploit parallelism. Therefore, perhaps redundant demands should not be made for each demand but done only for certain terms, depending on resource constraints and fault tolerance requirements.

The following table summarizes the above discussion.

	Data-driven Implementation	Demand-driven Implementation
Temporal Redundancy	Difficult	Straightforward
Spatial Redundancy	Difficult	Straightforward

Figure 1: GLU Abstract Architecture

5 A Practical Application

The above described technique to tolerate hardware faults using intensional computing models is being applied in a general-purpose parallel software system currently being developed at SRI [30]. The system, based on a coarse-grain intensional language called GLU [31], embodies an abstract architecture that implements the eduction computing model using demand-driven execution. Here, we focus on how the fault-tolerance techniques proposed in this paper are incorporated into the abstract architecture.

The basic component of the abstract architecture is a *GLU engine* which consists of a *generator* and one or more *executors*. Each GLU engine (GE) is capable of executing a GLU program by itself. It is connected to other GE's using a network whose topology is determined by the underlying concrete architecture. The generator in each GE is responsible for demand-propagation and the execution of fine-grain operators whereas each executor is responsible for executing coarse-grained functions. The abstract architecture is shown in Figure 1.

Each GE will propagate a demand to another GE in two cases:

1. The demanded value would be the result of executing a coarse-grain function (possibly after further demand propagation) at the other GE.

2. The demanded value would already have been computed and available at the other GE.

The demanded value, when computed, will be returned to the demanding GE.

Failure of a GE manifests as an omission fault. As a result, values in the failed GE demanded by other GEs will not be provided. A failed GE can be detected simply by the other GEs using the timeout mechanism. In the case when the demanded value was being obtained by function execution, the value can be redemanded and recomputed at another GE. The case when the demanded value was previously computed and stored in the failed GE, the alternate GE is chosen such that all demands for the value are, in some way, redirected to it. Fault-recovery using temporal redundancy is simple for two reasons. Firstly, only one value needs to be computed (and not the entire data structure that the value is part of) and secondly, coarse-grain functions are "functional" in the sense that a function's output only depends on its argument values.

A GE may not fail but may corrupt a demanded value resulting in a corruption fault. This can be dealt with using spatial redundancy. Essentially, each (or specific) demands for values can be redundantly replicated so that the same value is being computed at several GEs. The correct value for the demand can be decided by majority voting at the demanding GE. The values of the redundant demands should be identical in the absence of corruption faults because functions behave idempotently.

The abstract architecture, with its built-in fault-tolerance, can be realized on a variety of extant parallel computers from hypercube architectures [32] to workstation networks. We have realized the abstract architecture on a network of Unix workstations. We have been able to successfully demonstrate tolerance of omission faults by turning off a workstation while a GLU application is in progress and showing that the other workstations are able to gracefully recover from this fault without any loss in functionality.

6 Conclusions

We have suggested that fault tolerance at the computing-model level is desirable in multiprocessors and that computing models for inherently parallel functional language programs provide for implicit fault-tolerance through temporal and spatial redundancy. While both extensional and intensional computing models can achieve this, we have argued that intensional computing models are much more efficient in tolerating omission and corruption faults. We have further shown that demand-driven implementations (instead of data-driven implementations) of intensional computing model can naturally realize fault-tolerance. This is being implemented in a parallel software system based on an intensionally modeled language called GLU. An important practical benefit of the proposed fault-tolerance at the computing-model-level is that it is transparent to both the parallel applications programmer and the parallel computer system architect.

References

[1] J-C. Laprie. Dependable computing and fault tolerance: Concepts and terminology. In *Proceedings 15th International Symposium on Fault-Tolerant Computing*, Ann Arbor, Michigan, 1985. IEEE.

[2] D.P. Siewiorek and R.S. Swarz. *The Theory and Practice of Reliable System Design.* Digital Press, 1982.

[3] J.H. Wensley, L. Lamport, J. Goldberg, M.W. Green, K.N. Levitt, P.M. Melliar-Smith, R.E. Shostak, and C.B. Weinstock. SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control. *Proceedings of the IEEE*, 66(10):1240–1255, October 1978.

[4] B. Randell. System structure for software fault-tolerance. *IEEE Transactions on Software Engineering*, 1(2):220–232, 1975.

[5] L. Chen and A. Avizienis. N-version programming: a fault-tolerant approach to reliability of software operation. In *Proceedings 8th International Symposium on Fault-Tolerant Computing*, pages 3–9, 1982.

[6] R. Strong. Problems in fault tolerant distributed systems. In *Proceedings COMPCON Spring 85*, pages 300–306. IEEE, 1985.

[7] R. Jagannathan and E.A. Ashcroft. Fault-tolerant aspects of the eduction model and architecture. In *Proceedings of the 7th Digital Avionics Systems Conference*, Fort Worth, Texas, 1986.

[8] R. Harper, G. Nagle, and M.A. Serrano. Use of a functional programming model in fault tolerant parallel processing. In *Proceedings of the 21st Annual International Symposium on Fault-Tolerant Computing*, pages 20–26, 1989.

[9] P. Hudak. The conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21(3), 1989.

[10] W.W. Wadge and E.A. Ashcroft. *Lucid, the Dataflow Programming Language.* Academic Press U.K., 1985.

[11] R. Milner. A proposal for standard ML. In *Proceedings 1984 ACM Conference on LISP and Functional Programming*, pages 184–197, 1984.

[12] D.H. Grit. Towards fault tolerance in a distributed applicative multiprocessor. In *Proceedings 14th International Conference on Fault-Tolerant Computing*, pages 272–277. IEEE, 1984.

[13] F.C.H. Lin and R.M. Keller. Distributed recovery in applicative systems. In *Proceedings 1986 International Conference on Parallel Processing*, 1986.

[14] A.A. Faustini and W.W. Wadge. Intensional programming. In J.C. Boudreaux, B.W. Hamill, and R. Jernigan, editors, *The Role of Languages in Problem Solving 2*. Elsevier Science Publishers B.V. (North-Holland), 1987.

[15] E.A. Ashcroft, A.A. Faustini, and R. Jagannathan. An intensional parallel processing language for applications programming. In B.K. Szymanski, editor, *Parallel Functional Programming Languages and Environments.* ACM Press, 1990.

[16] E.A. Ashcroft. Dataflow and eduction: Data-driven and demand-driven distributed computation. Technical Report SRI CSL-151, Computer Science Laboratory, SRI International, Menlo Park, California 94025, 1986.

[17] A. Church. The Calculi of Lambda-Conversion. *Annals of Mathematical Studies*, 6, 1951.

[18] E.A. Ashcroft. Lucid is Second-Order: Explaining Lucid and Intensionality to Functional

Programmers. In *Proceedings of the Fourth International Symposium on Lucid and Intensional Programming*, Computer Science Laboratory, SRI International, Menlo Park, California 94025, USA, 1991.

[19] J. Darlington and M. Reeve. ALICE: A multiprocessor reduction machine for the parallel evaluation of applicative languages. In *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture*. ACM, November 1981.

[20] S.L. Peyton-Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[21] J. Goguen, C. Kirchner, and J. Meseguer. Concurrent Term Rewriting as a model of computation. In *Proceedings of the Graph Reduction Workshop*, Santa Fe, New Mexico, USA, 1986.

[22] R. Jagannathan. A descriptive and prescriptive model for dataflow semantics. Technical Report CSL-88-5, Computer Science Laboratory, SRI International, Menlo Park, California 94025, May 1988.

[23] Arvind and K.P. Gostelow. Some relationships between asynchronous interpreters of a dataflow language. In Neuhold E.J., editor, *Formal Description of Programming Languages*. North-Holland, New York, 1977.

[24] J.B. Dennis, D.P. Misunas, and C.K.C. Leung. A highly parallel processor using a data flow machine language. Computation Structures Group Memo 134, Laboratory for Computer Science, MIT, January 1977.

[25] J.R. Gurd and I. Watson. A multilayered data flow computer architecture. In *Proceedings of the 1977 International Conference on Parallel Processing*, page 94. IEEE, August 1977.

[26] R. Montague. *Formal Philosophy*. Yale University Press, 1974.

[27] I. Watson and J. Gurd. A practical data flow computer. *Computer*, 15(2):51–57, February 1982.

[28] Arvind and K.P. Gostelow. The U-Interpreter. *COMPUTER*, 15(2), February 1982.

[29] E.A. Ashcroft. Eazyflow architecture. Technical Report SRI CSL-147, Computer Science Laboratory, SRI International, Menlo Park, California 94025, 1985.

[30] R. Jagannathan and A.A. Faustini. GLU: A system for scalable and resilient large-grain parallel processing. In *Proceedings of 24th Hawaii International Conference on System Sciences*, Kauai, Hawaii, January 1991.

[31] R. Jagannathan and A.A. Faustini. The GLU Programming Language. Technical Report SRI-CSL-90-11, Computer Science Laboratory, SRI International, Menlo Park, California 94025, USA, November 1990.

[32] C.L. Seitz. The Cosmic Cube. *Communications of the ACM*, 28(1):22–33, January 1985.