GREG GIBELING
GDGIB@EECS.BERKELEY.EDU
UC BERKELEY
11/29/2005

The Art of Controlled Chaos
A Survey of Dataflow & Concurrent Programming

1.0 Introduction

There are currently a wide variety of systems and architecture projects attempting to design or program in a dataflow style. The fundamental reason for this is a desire to expose concurrency at various levels, and support a more natural style of programming both for modeling and implementation reasons.

Working with concurrency exposes the fundamental flaw in the von Neumann view that computers are sequential imperative machines: no sufficiently complex physically realistic system can or will do exactly one thing at a time. As such system designers have begun to realize the immense performance gains which could be achieved by exposing a more parallel, or at least a more realistic, model of a computer to the programmer.

Unfortunately there exist today no widely used general purpose languages for describing dataflow and concurrent programs. Most projects have their own style or language: WaveScalar [1] at Washington recompiles Alpha binaries, FLEET [2] by Ivan Sutherland is slowly giving rise to a very odd machine language (UC Berkeley CS294-4 is in the process of defining it by in-class argument), RAW [3] from MIT uses a custom encoding and the SCORE project here at Berkeley uses a streaming language called TDF [4].

While there have been suggestions that common languages can be developed to exploit these architectures [5] the full power and use of these languages remains an open area of research.

Worse than the lack of a single language, the biggest problems facing concurrent dataflow systems research is the missing connection between practical and theoretical work. The purpose of this paper is to define the problems and state of the art in dataflow and concurrent programming, in the hope of starting to bring the two together. This paper will reference both dataflow programming models and dataflow architectures. It should be understood that the two are not inevitably tied together, but will both benefit from cooperative research.

Sections 2.0 Applications and 3.0 Frameworks give several motivating applications and existing frameworks. Section 4.0 Dataflow & Concurrency discusses the foundations and relevant history of dataflow and concurrent systems, as well as describing their interaction. Section 5.0 Models presents some of the available formalisms and 6.0 Languages gives examples of specific, practical languages.

2.0 Applications

This section describes four major application areas of computer science all of which rely on concurrent execution

for high performance, and dataflow models to achieve it.

This section is not intended only to motivate interest in dataflow or concurrency, but to provide focus and constraints which the remaining sections of this paper, and major dataflow research, should address.

2.1 Hardware

Digital logic is perhaps the single most widely understood example of a concurrent, dataflow system. The fact is that a circuit, unlike an imperative command or an ISA instruction, physically exists at all times, and will be constantly performing some operation. In this sense, concurrency in hardware is inevitable.

There are two general classes of digital hardware: synchronous and asynchronous [6] [7]. The difference is that while synchronous systems rely on a global clock to orchestrate computation and data movement, asynchronous systems rely directly on the dataflow firing rule (see section 4.1 Dataflow).

However, given that synchronous systems have grown beyond designers' ability to reason about the complete design at once, large synchronous systems are generally designed with large components connected in a dataflow style.

The primary constraint associated with hardware design is that the computation structures must be entirely static; indirection and higher order concepts are impractical. This highlights some of the problems with the original dataflow work at MIT as described in [8].

2.2 Databases

With a history of over 25 years databases are perhaps the longest standing successful example of a dataflow system. Since the relatively early days of SystemR [9] at IBM research, database query execution has relied on the composition of iterative or streaming operators. Each operator provides or operates individually on each row of a relational database table, or "tuple."

The need for high performance has long since led to parallel execution of query plans, as in the Volcano [10] system and in more recent years, Mariposa [11], TelegraphCQ [12] and P2 [13] have renewed interest in this field for efficient and highly concurrent query execution.

Databases represent one of the biggest successes of dataflow programming, as well as an area where performance and efficiency improvements will always be needed. Exactly these properties make databases a valuable application to consider.

2.3 Networking

A single network, for example a segment of Ethernet, is already a good example of a concurrent dataflow system, each node takes independent action as the result of receiving a packet. However with the advent of the internet, it has come to the point where concurrent, but interacting systems might be separated by minutes and miles, rather than milliseconds and feet.

What's more networking components like routers are almost ideal dataflow components, operating on each packet or datagram semi-independently and often in concurrently, especially at 1Gbps line speeds.

Recent proposals for more advanced or flexible routing systems have often led to systems with dataflow architectures and declarative dataflow

languages for programming. Examples include Click [14] and P2 [13].

Because of this, much of the latest research into practical applications of dataflow design has been in this area, which like databases, is both a source of research and a sink for useful results.

2.4 Summary of Applications

In this section we have presented hardware, databases and networks as both motivating and constraining applications for concurrent dataflow languages and architectures.

This list should by no means be taken as exhaustive however, as areas like scientific computing, multimedia, GUIs, and web servers, are also excellent applications for dataflow techniques. However the above applications represent difficult, well understood and actively research applications respectively with seemingly different requirements, that when taken in aggregate still effectively cover the application space.

The reader is invited throughout the remainder of this paper to periodically reconsider problems commonly seen from the imperative or control flow perspective in light of what they are reading.

3.0 Frameworks

This section presents a general rundown of three of the primary programming frameworks on top of which various projects are attempting to build dataflow models. This section is not a discussion of best practices or good ideas, but rather the existing practical experience and ideas currently in use for achieving dataflow like semantics at a high level.

3.1 Shared Memory

The shared memory programming framework is the most common in use today. Languages like C, C++ and Java all support this organization, as do most operating systems, compilers and multi-processor architectures. Shared memory is so prevalent that it is barely recognized to be one choice among many.

In fact the shared memory permeates computers to the level of a standard ISA. Normal MIPS or IA32 instructions can be considered an example of this, where the register-file constitutes the shared memory.

As such all of the problems which have in general retarded the development of standard ISAs and assembly languages apply to shared memory systems at a higher level of abstraction. This offers some insight as to why consistency, coherence and locking in imperative and object oriented languages, present such a complicated problem [15]: a programmer designing a large system in a language like C, C++ or Java with threads and shared memory faces almost exactly the same problems as a programmer writing a single function in assembly language, but with the added complexity of non-atomic operations.

Locking, aliasing, state management, and operation interleaving are all complicated and hard to understand in the shared memory model, which almost always forces a programmer to make conservative choices with respect to synchronization and optimization, exactly the area dataflow languages excel at simplifying.

3.2 Message Passing

Shared memory programming has the inherent drawback that

concurrency and data independence must be simultaneously managed explicitly by the programmer. This can be alleviated by message passing systems, which enforce data independence and concurrency by the same mechanism. In message passing systems data is divided into independent messages, which can then be processed independently.

Based on the above description message passing is a first order approximation to full dataflow as described in section 4.1 Dataflow. The main difference is that a message passing system will often include sequential execution of operations between the receptions of messages, leading to a model with concurrently executing processes which communicate by IPC messages.

In addition to web servers, and some object oriented languages like Smalltalk [16], [17] most highly parallel super computers make use of some kind of message passing system. In fact some of the most common message passing frameworks exist to implement distributed shared memory.

Almost all large concurrent systems will use either message passing or shared memory to manage concurrency and data sharing. While at first glance this appears to be a major classification of systems, [18] concludes that in fact these two structures are functional duals of each other, leading us to believe that framework specific languages will be duals of each other.

However, given that a message passing framework can communicate over large distributions and latencies, while still performing useful work, it should be clear that message passing frameworks are generally more powerful than shared memory.

3.3 Transaction Oriented

Building on the widespread success of concurrency management in database systems [19], many researchers have proposed systems and languages based on some concept of atomic, and optionally concurrent operations. Examples of active research in this are include languages [20], formalisms [21] and architectures [22].

However while there are many practical examples of work in this field, most of the theoretical basis lies in the success of this framework for use in databases, leaving somewhat of a void in the formal analysis of these systems.

Problems like structural congruence [23], [24] and performance [25] which are introduced by these “transaction oriented” systems remain poorly studied, especially in the case of projects like [22] where these ideas are being applied at the most fundamental levels of computer architecture and cost margins are razor thin.

Transaction systems remain a highly promising area of research, as they offer ways to connect practically realizable systems to a formal model which can be analyzed. In addition both shared memory and message passing frameworks fit within transaction frameworks.

3.4 Summary of Frameworks

In this section we have presented several existing frameworks for managing the flow of data and computation.

The point of this section is that while all of these frameworks are compatible, can be, and often are, implemented on top of more basic languages like C, there is considerable power to extending existing languages and systems to fully incorporate one or

more of these frameworks. Not the least of which is the resulting availability of formal and theoretical work which could then be applied to optimization and correctness proofs.

4.0 Dataflow & Concurrency

Taken together sections 2.0 Applications and 3.0 Frameworks provide a relatively complete outline of the motivations and requirements for dataflow programming or systems design.

This section formally defines both the terms "dataflow" and "concurrent." But more importantly it provides a survey of both the current and historical work in dataflow languages and architectures, followed by a discussion of how this applies to concurrent programming.

4.1 Dataflow

There are two obstacles to infinitely fast execution of a program. First, processing takes time. Second, each operation requires its' inputs to be ready before it can begin computing.

Dataflow programming opens up interesting and efficient implementation options by accurately capturing the true data dependencies between computations. Because the exact flow of data elements, often called "tokens," is precisely laid out in these languages, the delays which are not the result of the above two limitations can be controlled or removed.

The term "dataflow" is used in this paper to refer to those systems, architectures, models and languages in which computation is driven by the availability and motion of data. The term "dataflow" is used in contrast to "control flow" in which computation is driven by some form of imperative controls such as in structured languages like C.

Dataflow languages and systems are generally characterized by the "dataflow firing rule" which states that an operation may take place as soon as all of its' inputs are present.

Those familiar with Petri-Nets (see section 5.2 Petri-Nets) will recognize this as the transition firing rule. In fact some of the original work by Dennis [26] describes a dataflow architecture which is essentially a simple hardware realization of a colored marked graph, Petri-Net. That is, tokens have values which can be operated upon, and no more than one token may appear at each place. This is the basis of the view that dataflow programming is a concrete implementation of abstract computation graphs [27].

The best known history of the term "dataflow" begins with Dennis's work at MIT, which later blossomed into projects like ETS [8] and Monsoon including a series of papers by D.E. Culler [28] [29] [30].

Most early work, such as ETS, suggests an architecture and language, which in more current terms would be called "push dataflow." Each instruction includes an operation and a continuation, i.e. the instruction(s) to which its' result(s) should be delivered. This is a stark contrast to "pull dataflow" where an instruction is specified as a list of instructions which generate its' inputs and an operation to produce a single output. Recently systems like Click [14] and P2 [13] have arisen which are careful to include both push and pull elements, suggesting that formal analysis of this space is wanting.

Another characteristic of the early dataflow architectures is their reliance on dynamic program structure,

often realized as a tree of activation records rather than a stack. These projects did not significantly exploit compile time techniques for flattening this tree or transforming it to a static structure, instead relying entirely on hardware to do this at run time.

Newer projects, like Score [4], RAMP [31] and P2 [13], generally convert this dynamic structure to a static one, as much as possible in order to simplify implementation and increase performance. In fact in very low level hardware systems like Score, RAMP and most DSP systems, one of the key requirements for a practical implementation is the elimination of any dynamic structure through compile time transformations such as those suggested by [27]. This is one of the primary reasons for including section 2.1 Hardware in this paper.

As a final historical note, it is interesting that Dennis and Misunas describe at the end of [26], a possible distributed architecture, which is surprisingly similar to WaveScalar [1] and FLEET [2], both of which are being built and explored nearly 30 years later. As with any long dormant research, this connection should suggest some caution in modern research. While it is likely that the ideas are now ready for new work, it is also possible that the same dead ends will be met as 25 years ago.

One of the main points of this survey is to attempt to tie recent and historical, theoretical and practical research together, in order to avoid past problems.

4.2 Concurrency

In this paper, two events or computations are "concurrent" if they have an unknown temporal relationship. That is to say they may be simultaneous, parallel or sequential: no relationship is known.

As stated at the beginning of this paper, a sequential von Neumann model of computers, while easy to understand, is insufficient for high performance systems, and is inadequate to capture the issues in operating systems and architecture design. As such concurrent models are required, and it is the combination of this requirement and the existence of program analysis techniques such as in [27] and [32] which drives our interest in dataflow programming.

Though [27] spends a lot of time talking about translating from imperative shared memory programs to dataflow (similar to basic block analysis techniques), when a system in specified in a dataflow language, the programmer is given the opportunity to explicitly create only the required sequencing. This increases the allowable flexibility of the compiler and eventual run time system to modify the program for efficiency reasons without affecting its' correctness.

The key to increasing concurrency in a dataflow language is the existence of a very large, inexpensive synchronization namespace, as described in [8]. They state "To obtain high performance on a [concurrent] machine lacking these features, a program must be partitioned into … processes that operate … on local data and rarely interact." This statement embodies the main reason for the pervasive interest in dataflow languages and systems: they represent an easily understood way to transfer the responsibility of the time/space tradeoff from the programmer to the run time system.

This transfer significantly increases the complexity of the run time

system, as evidenced by the problems encountered by projects like Monsoon, the ETS and TTDA at MIT, and it remains a problem for more modern systems. However as papers like [27] suggest, there exist simple yet very powerful compile time analysis techniques which should help to conquer these issues.

There are two canonical examples of the dangers inherent in a potentially unbounded synchronization space. Managing an excess of parallelism [28] exposed by dataflow architectures, and the difficulty of deciding congruence of concurrent systems as briefly outlined in [23].

The resource requirements problem inherent in dataflow architectures also resulted in problems in bounding storage requirements in the 1980s. This highlights the need for better formalisms to reason about concurrent dataflow systems. ETS as described in [8], attempts to explicitly capture state requirements, a definite positive step, which is none the less ignored by most of the models in section 5.0 Models. In fact the original WaveScalar specification includes unbounded buffers between dataflow elements, a clear problem in past architectures.

4.3 Concurrent Dataflow

Dataflow languages are an ideal way to capture the exact "meaning" of a programming, especially with respect to concurrency and ordering requirements. The use of the simple dataflow firing rule, combined with a large synchronization namespace for tokens allows a programmer to specify the desired behavior in a very precise manner. What's more, dataflow architectures allow these specifications to be directly executed.

However, problems such as bounded storage, practical implementation requirements, dynamic computation structures and under application of formal models have all hindered efforts in this direction.

5.0 Models

This section describes four possible formal models for reasoning about concurrency and dataflow. None of them is perfect, each has serious flaws, and as such they have been chosen not only for their relative fame, but also for the interesting lessons to be learned.

5.1 Process Networks

Process networks [33] represent one of the purest dataflow formalisms, in no small part because of work like in [34], where Lee gives a denotational semantics for standard "Dennis Dataflow" which equates it with process networks.

Process networks are characterized by a graph of statically connected dataflow operators which operate over lower order units of data.

Significant work in this field by Lee and others [35], has made it particularly amenable to resource constrained digital signal processing and hardware applications.

These areas are characterized by their hard real time constraints, static, numerically intensive computation structure and their need to handle infinite inputs.

It is the last requirement, infinite I/O, which forces a break from traditional programming models founded on λ-calculus, which is unusable in infinite I/O situations [34].

5.2 Petri-Nets

Petri-Nets [36] are perhaps the simplest model for concurrent and dataflow systems. In addition to the fact that they are an easily understood graphical model, standard Petri-Nets have exactly one rule for execution, which represents a highly simplified version of the dataflow firing rule.

Unfortunately with this simple one-rule version, Petri-Nets are not Turing complete. However with the addition of inhibitor arcs, Petri-Nets are become "extended Petri-Nets" which are Turing complete. This addition means that not only can Petri-Nets model both low and high level constructs, they are defined by exactly two rules.

On of the simplest shortcoming of Petri-Nets is the lack of a uniform easily understood textual representation for them. This somewhat limits their utility in formal analysis, as it is hard to write proofs with large graphics.

However the more deeply rooted problem with first order Petri-Nets is the exponentially lower expressivity, which seriously retards their applicability to larger problems. The fact of the matter is that even with techniques for managing partial nets, it remains very difficult to think about Petri-Nets. Even the simplest examples can be prohibitively complicated to write and verify.

5.3 π-Calculus

π-Calculus is one of the widest known and best studied formalisms for modeling concurrent, and potentially non-deterministic computation. It is based on the parallel composition and branching of sequential processes, which communicate channel names over channels. A more complete introduction can be found in [23] or [37].

One key fault with process calculi in general is their encoding of time and state. There is no reason in general to conflate the two issues, and many years of research have gone into separating the two in the form of out-of-order execution (Tomasulo's Algorithm [38], ROBs, etc). Unfortunately π-calculus, and many others implicitly model temporal and state progression with the same dotted prefix notation.

As a side effect of this conflation of state and time π-calculus may require unbounded state, and the current state is not explicit in the text of the model. This is because state, for each process is modeled as an environment in the programming language sense: new channels are always added to it, never subtracted. Of course, this makes it easy to write, but very hard to reason about, and effectively impossible to implement, never mind debug.

Furthermore the "!" operator in π -calculus models exactly the kind of deadlock inducing behavior which any practical implementation must shun. While we must be able to model this, another construct, such as pull dataflow, would be better suited to practical use.

5.4 Mobile Ambients

Mobile Ambients [39], [40] are a formalism proposed by Luca Cardelli for modeling mobility and security. As such they are very good for modeling ideal security situations.

Mobile Ambients add location and motion on top of a π-Calculus like base, without conflating issues or adding syntactic sugar, and the inherent hierarchy is well suited to modeling the physical world. However because of their heritage Mobile Ambients are bad

at representing state, the same as π-Calculus. Worse yet, they are quite unsuccessfully for modeling security, as perfect capability based systems are easily described, but security breaches and cryptographic cracking are ignored.

Despite the drawbacks of Mobile Ambients, the idea that locality should be a first order concept at the model level is of key importance and the primary reason for their inclusion in this paper. The fact is that models for dataflow and concurrency have often ignored the spatial dimension of the specification making it hard for a system or programmer to express a trade between space and time.

5.5 Summary of Models

Aside from the four formalisms presented in this section there are numerous others including guarded commands [41], CSP [42], CQP [43], CCS, the Actor Model [44], τ-Calculus [21], Session Types [45] and Specification Diagrams [46], some of which complement the above four, but all of which aim to model concurrency.

The goal of this section was to highlight two things: first there are a number of models for concurrent dataflow systems, and second, there are two basic flaws which can be introduced to a modeling formalism. The fundamental modeling flaw is a models inability to capture undesirable occurrences so as to be able to prove that they do not arise. The other flaw arises when a model captures some interesting feature of the system being modeled in an implicit way, making explicit reasoning difficult if not impossible.

6.0 Languages

This section discusses real world languages and systems, all of which are built on dataflow foundations, but none of which use the same terminology.

One of the key points of this paper is a survey of the following languages.

6.1 SQL

SQL is widely used and widely known, making it a perfect example of a successful dataflow language, despite the fact that it isn't always seen as one.

The key reason SQL is amenable to dataflow execution and optimization techniques is its declarative nature. SQL specifies what the answer should be, but gives surprisingly little information about how to compute it, representing an ideal tradeoff between programmer and run time responsibility.

The process of transforming a declarative SQL query to a dataflow query plan is called "query optimization" [47] and has been around for 30 years, but even in this field new ideas like Eddies [48] promise interesting results.

6.2 BlueSpec

BlueSpec is a hardware description language which started as a research project at first MIT [49] then CMU, and has since been turned into a corporate product [50].

BlueSpec is based on the concepts presented by Dijkstra in [41]. Commands are called "rules" and are grouped along functional lines, in an object oriented fashion. The compiler automatically determines which rules (commands) can be run concurrently, thereby resolving Dijkstra's choice operator at compile time.

Of course there are no loops, forcing the programmer to manage loop parallelism and state explicitly. Also, as this is a very practical language it

includes arbitration as syntactic sugar, allowing rules (commands) to have precedence.

The final and largest difference between BlueSpec and the guarded command formalism is that BlueSpec, like most hardware description languages, lacks facilities for sequencing. All rules are concurrent.

6.3 P2, Overlog & Click

The P2 project [13] began as a combination of work in distributed databases and P2P overlay networks. As such is has led to a declarative logic programming language mixed with dataflow, and named "Overlog." Overlog is an extension to Prolog with locations for logic rules. Rules now represent the creation or arrival of a relational tuple.

A more widely known system, with a less interesting language is The Click Modular Router [14]. While it is not really a router, it remains good example of a real world use of dataflow programming, and one of the first times the author of this paper has seen the explicit management of the dichotomy between push and pull dataflow.

6.4 Liberty & RDL

The Liberty project [51] at Princeton has applied standard programming language techniques to hardware design, including polymorphism, overloading and techniques from aspect oriented programming.

RDL [31], a language being developed by the author of this paper, is very similar to Liberty and others, in that it aims to capture a specification of digital logic as a series of interconnected dataflow like operators. RDL is part of the RAMP [52] project whose goal is to accelerate the development of multi-core architectures, operating systems and languages by providing a rapid simulation platform. Thus concurrency, networking and hardware issues are all highly relevant to this work.

7.0 Conclusion

This paper contains: a survey of applications and frameworks for concurrent dataflow programming, a brief history of dataflow languages and architectures, and their relation to concurrency, and several formalisms and practical languages for reasoning about and expressing dataflow programs and architectures.

This paper has outlined several problems and areas of open research for future work, as well as a number of existing projects in an effort to start closing the gap between the practical and theoretical work in this field.

8.0 Future Work

There remains significant future work in this field, including the development of better models for concurrent dataflow systems and their accompanying type systems. Both session types [45] and scheduling [35] might shed some light on this area.

The ideal end result would be the introduction of formal languages and models to real world applications at such a level as to allow subjects like dataflow programming and analysis and concurrency to be effectively taught at the undergraduate level, rather than remaining a "black art" as they are today.

9.0 References

1. Swanson, S., et al. *WaveScalar*. in *36th International Symposium on*

Microarchitecture. San Diego, CA. 2003.
2. Sutherland, I., *FLEET – A One-Instruction Computer*. 2005. p. 1-12.
3. Taylor, M.B., et al., *The Raw microprocessor: a computational fabric for software circuits and general-purpose programs.* IEEE Micro, 2002. **22**(2): p. 25-35.
4. Caspi, E., *Programming SCORE*. 2005.
5. Mattson, P., et al., *Stream Virtual Machine and Two-Level Compilation Model for Streaming Architectures and Languages*. 2004. p. 1-3.
6. Sutherland, I. and S. Fairbanks. *GasP: a minimal FIFO control*. in *Proceedings Seventh International Symposium on Asynchronous Circuits and Systems. ASYNC 2001. Salt Lake City, UT*. 2001.
7. Ebergen, J. *Squaring the FIFO in GasP*. in *Proceedings Seventh International Symposium on Asynchronous Circuits and Systems. ASYNC 2001. Salt Lake City, UT*. 2001.
8. Papadopoulos, G.M. and D.E. Culler. *Monsoon: an explicit token-store architecture*. in *Seattle, WA*. 1990.
9. Chamberlain, D.D., et al., *A history and evaluation of System R.* Communications of the ACM, 1981. **24**(10): p. 632-46.
10. Graefe, G. *Encapsulation of parallelism in the volcano query processing system*. in *1990 ACM SIGMOD International Conference on Management of Data. Atlantic City, NJ*. 1990.
11. Stonebraker, H., et al., *Mariposa: a wide-area distributed database system.* Vldb Journal, 1996. **5**(1): p. 48-63.
12. Chandrasekaran, S., et al. *TelegraphCQ: Continuous Dataflow Processing for an Uncertain World*. in *CIDR*. 2003.
13. Loo, B.T., et al., *Implementing Declarative Overlays*. 2005: UC Berkeley. p. 1-16.
14. Kohler, E., et al., *The Click modular router.* ACM Transactions on Computer Systems, 2000. **18**(3): p. 263-97.
15. Maessen, J.W., Arvind, and S. Xiaowei. *Improving the Java memory model using CRF*. in *OOPSLA 2000. Conference on Object-Oriented Programming Systems, Languages and Applications. Minneapolis, MN*. 2000.
16. Ingalls, D.H.H., *Design principles behind Smalltalk.* Byte, 1981. **6**(8): p. 286-98.
17. Ingalls, D., et al. *Fabrik: a visual programming environment*. in *ACM SIGPLAN 3rd Annual Conference on Object-Orientated Programming Systems, Languages, and Applications (OOPSLA 88). San Diego, CA*. 1988.
18. Lauer, H.C. and R.M. Needham. *On the duality of operating system structures*. in *Le Chesnay, France. 2-4 Oct. 1978.* 1978.
19. Franklin, M.J., *Concurrency Control and Recovery*, in *The Handbook of Computer Science and Engineering*, A.B. Tucker, Editor. 1997, CRC Press : Published in cooperation with ACM: Boca Raton, Fla.
20. Hoe, J.C., *Operation-Centric Hardware Description*
and Synthesis, in *Electrical Engineering and Computer Science*. 2000, Massachusetts Institute of Technology: Boston. p. 139.
21. Field, J. and C.A. Varela. *Transactors: a programming model for maintaining globally consistent distributed state in unreliable environments*. in *POLP 2005: 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. Long Beach, CA*. 2005.
22. Ananian, C.S., et al. *Unbounded transactional memory*. in *Proceedings. 11th International Symposium on High-Performance Computer Architecture. San Francisco, CA*. 2005.
23. Pierce, B., C., *Foundational Calculi for Programming Languages*, in *CRC Handbook of Computer Science and Engineering*. 1995.
24. Lohman, G.M. *Grammar-like functional rules for representing query optimization alternatives*. in *SIGMOD International Conference on Management of Data. Chicago, IL.* 1988.
25. Agrawal, R., M.J. Carey, and M. Livny, *Concurrency control performance modeling: alternatives and implications.* ACM Transactions on Database Systems, 1987. **12**(4): p. 609-54.
26. Dennis, J.B. and D.P. Misunas. *A preliminary architecture for a basic*

data-flow processor. in *Houston, TX*. 1975.
27. Beck, M., R. Johnson, and K. Pingali, *From control flow to dataflow.* Journal of Parallel & Distributed Computing, 1991. **12**(2): p. 118-29.
28. Culler, D.E. and Arvind. *Resource requirements of dataflow programs*. in *Honolulu, HI*. 1988.
29. Culler, D.E., et al. *Fine-grain parallelism with minimal hardware support: a compiler-controlled threaded abstract machine*. in *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. Santa Clara, CA*. 1991.
30. Culler, D.E., K.E. Schauser, and T. von Eicken. *Two fundamental limits on dataflow multiprocessing*. in *Architectures and Compilation Techniques for Fine and Medium Grain Parallelism. IFIP WG10.3 Working Conference. Orlando, FL*. 1993.
31. Gibeling, G., A. Schultz, and K. Asanovic, *RAMP Architecture & Description Language*. 2005, UC Berkeley.
32. Swanson, S., et al., *Dataflow: The Road Less Complex*. 2003. p. 13.
33. Lee, E.A. and T.M. Parks, *Dataflow process networks.* Proceedings of the IEEE, 1995. **83**(5): p. 773-801.
34. Lee, E.A., *A Denotational Semantics for Dataflow with Firing*. 1997: Berkeley, CA.
35. Buck, J.T. and E.A. Lee. *Scheduling dynamic dataflow graphs with bounded memory using the token flow model*.
36. Murata, T., *Petri nets: Properties, analysis and applications.* Proceedings of the IEEE, 1989. **77**(4): p. 541-80.
37. Milner, R., *The Polyadic pi-Calculus: a Tutorial*. 1991.
38. Tomasulo, R.M., *An Efficient Algorithm for Exploiting Multiple Arithmetic Units.pdf.* IBM Journal, 1967.
39. Cardelli, L. and A.D. Gordon, *Mobile Ambients*. 2003.
40. Cardelli, L. *Mobility and security*. in *Proceedings of Secure Computation. Marktoberdorf, Germany. 27 July-8 Aug. 1999.* 2000.
41. Dijkstra, E.W., *Guarded commands, nondeterminacy and formal derivation of programs.* Communications of the ACM, 1975. **18**(8): p. 453-7.
42. Hoare, C.A.R., *Communicating sequential processes.* Communications of the ACM, 1978. **21**(8): p. 666-77.
43. Gay, S.J. and R. Nagarajan. *Communicating Quantum Processes*. in *POLP 2005: 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. Long Beach, CA*. 2005.
44. Agha, G.A., *Actors: a Model of Concurrent Computation in Distributed Systems*. 1986, MIT Press.
45. Gay, S., V. Vasconcelos, and A. Ravara, *Session Types for Inter-Process Communication*. 2003, University of Glasgow: Glasgow, Scotland.
46. Smith, S.F. and C.L. Talcott, *Specification diagrams for actor systems.* Higher-Order & Symbolic Computation, 2002. **15**(4): p. 301-48.
47. Griffiths Selinger, P., et al., *Access path selection in a relational database management system.* 1 Aug. 1979, 1979: p. 59.
48. Avnur, R. and J.M. Hellerstein. *Eddies: continuously adaptive query processing*. in *2000 ACM SIGMOD. International Conference on Management of Data. Dallas, TX*. 2000.
49. Hoe, J.C. and Arvind, *Operation-centric hardware description and synthesis.* IEEE Transactions on Computer-Aided Design of Integrated Circuits & Systems, 2004. **23**(9): p. 1277-88.
50. BlueSpec Inc., *BlueSpec Overview*. 2005, BlueSpec Inc: Waltham, MA. p. 2.
51. Manish, V., N. Vachharajani, and D.I. August. *The Liberty structural specification language: a high-level modeling language for component reuse*. in *2004 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'04). Washington, DC*. 2004.
52. Wawrzynek, J., et al., *RAMP Research Accelerator for Multiple Processors.* 2005.