Flow - A Concurrent Methodology Manager

Y. Kashai

National Semiconductor (I.C.) Ltd.
P.O. Box 3007, Herzliya B 46104, Israel.

Abstract

The Flow system is used to define methodology for ULSI design tasks in a formal way. It provides a concurrent environment in which such tasks are carried out automatically. The automatic execution of tasks in the Flow system utilize a set of workstations as computing resources. This paper describes the motivation for the development of Flow, the semantics used for methodology definition and the Flow system that carries out the defined tasks. Results demonstrating the systems robustness and superior load balancing technique are presented.

1 Introduction

The availability of low cost workstations and the growing number of graphical interactive tools have motivated many design centers to move from a centralized, mainframe based computing environment to a distributed network of interconnected workstations. This transition is not without difficulties [1], specifically for big tasks that would use up most of the mainframes power. The distributed environment has some intrinsic shortcomings, due to the fragmentation of computing power:

- It is difficult to achieve high utilization of the total computing power, as there is no centralized control. Some workstations can be idle while others are loaded.
- Tasks that require much processing power can take prohibitively long time in a distributed environment.
- Running processes influence the response time for the originator more directly than in a mainframe environment.

Those problems can be answered partly by providing better control over the distributed environment. The complimentary answer is modifying the profile of the running processes. Flow is quite unique in attempting to make use of both attitudes.

There are quite a few systems that make use of idle workstations to run remote processes on them. Starting with Shoch and Hupp's "worm" program [2], through Hagmann [3] and Litzkow [4] ideas, to the highly utilized Butler system [5] of CMU and Sprite of UCB [6]. All those systems work on a single process entity, lacking the ability to manage higher level entities (i.e. a group of processes that relate to each other). This ability is required, however, to handle processes concurrently. There are systems that can handle concurrent processes on a group of workstations, though most of the work in this field was done on specialized hardware.

A task even more difficult than resource control, is modifying the profile of the executing processes. Traditional ULSI design tools are not concurrent and require immense effort in order to break them into concurrent processes. It is claimed, however, that users are interested in shortening the processing time of design chunks, rather than improving the time of any specific process. The Flow approach is based upon the above assumption, taking into account the design methodology which allows running many tools in parallel. Flow is designed to work in the methodological level, with granularity of design tools. The Flow "program" codes the methodology and takes advantage of the existing parallelism to speed up design tasks, without actually speeding up any of the tools.

In order to code the design methodology for the tasks carried out by Flow, special semantics were needed. Those are presented in the second part. The architecture of the system that runs the tasks, manages resources and controls the operation is presented in the third part. Measured results are presented in the fourth part. Part five discuss related work and our conclusions.

2 Task definition

Throughout the design cycle of ULSI chips, there are stages where non-interactive tools are used in a sequence. Those stages are the main target of Flow. Usually some concurrency can be found in such stages. Sometimes the order of tool operation is not crucial. In other cases the hierarchy of the design can be used to fragment the task. Obviously, each design stage is repeated many times, for various parts of many projects. It is worthwhile then, to formulate the flow of work in a program. Whenever a part
is ready for processing, executing the program for it will ensure that the design stage is carried out just the way it was planned. This makes such tasks optimal in speed and resource consumption, fully complaint with the methodology in use, traceable and repeatable.

### 2.1 The Flow graph

Programs that code design methodology for Flow are called flow-graphs. Such programs are directed graphs $G(V,E)$, where $V$ is a set of nodes and $E$ is a set of directed edges. Each vertex $V(i, O, p) \in V$ is defined by a triple consisting of $I$, a vector of formal input parameters, $O$, a vector of formal output parameters and $p$, a sequential process that maps $I$ onto $O$, possibly through a transformation with some side-effects. The input vector must have a size of one or more, but the output may have zero size. An edge $e(O_i^j, I_j^l) \in E$ links between the $i$-th output of vertex $v \in V$ and the $j$-th input of vertex $u \in V$.

The node program $p_n$ may exist in two levels: on a meta-level within the flow-graph and as a real computer process. The meta-level program contains code that deals with the flow-graph control, as well as constructing the command line for the real process. The real process is created by Flow on any of its server computers to carry out a design task, as a side effect of the graph interpretation. The meta-level is connected to the computer process by a parameter passing mechanism, that enables sending and receiving data. The computer process can be regarded as a procedure called by the meta-level process.

The flow-graph is excited by dynamic entities called tokens. A token $t(j,d)$ is built from a job identifier $j$ and a datum $d$. A token can stand for a formal parameter in a nodes input or output vector, and it is passed along edges from one nodes output to another nodes input. Formally speaking, a job with identifier $j^*$ is the dynamically changing set of tokens $T_j$ where $t(j,d) \in T_j \Rightarrow j = j^*$. A job is initiated by creating the first such token, and it ends when $T = \emptyset$.

### 2.2 Node types

In order to allow more flexibility in the flow-graph there are three types of nodes. The node types are decided upon by the author of the flow-graph. They differ in the way they behave during execution:

**Simple Nodes:** Simple nodes, $v(I,O,p)$ create active instances $v^m(I,O,p)$ that spring into existence with a unique index $m$, called magic number, by the arrival of the first token. They acquire the tokens job identifier and become transparent to tokens of other jobs. When a token arrives for a parameter which is already occupied by another token, a new instance of the same node is created. The number of co-existing instances not bounded.

**Exclusive Nodes:** Exclusive nodes create a single active instance when they get the first token, much like simple nodes. However, nodes that attempt to send a token to an exclusive node with an active instance are blocked until that instance terminates.

**Source Nodes:** Source nodes have a single, ever existing instance, which is visible to all tokens. They have input and output vectors of a unit size, and no corresponding real process. In case the output edge is blocked, input tokens are accumulated within the source node.

The combination of the three node types provide semantics that are well suited for design task description.

### 2.3 The Flow-graph dynamics

The graph dynamics is much like a Data-Flow machine simulation [7]: A node instance $v^m(I,O,p)$ is said to be satisfied when each formal input parameter has a corresponding token assigned. Whenever a node is satisfied the mapping $p: I \rightarrow O$ takes place, possibly including a real computer process to provide part of the mapping. The output is thus produced in form of tokens that are placed within the output vector. When the process is terminated the output vector tokens are passed along their edges to the input vectors of some other nodes. This process is called firing. It is important to note that the tokens need not be the actual input or output of the computer process, but rather a symbol that represents such data (which may exist in a file, for instance). This mechanism is generic enough to model time dependencies and other relations.

![Figure 1: A simple flow-graph.](image-url)

The example given in figure 1 presents a simple flow-graph that gets a token representing a design block in the input source node, extracts its sub-blocks in the node called "extract", while firing the parts into the source node called "parts". Then all the sub-blocks are processed in parallel, in instances of the simple node called "process". Finally the outputs are combined within the exclusive node called "combine". The "combine" node must be exclusive, in order to prevent the creation of multiple instances, each with partial information. This scheme works for variable number of sub-blocks, so the level of concurrency is determined dynamically during the run.
Also, the fact that node instances are runnable does not force them to execute concurrently. This depends upon the resources available to carry out their corresponding processes.

3 The Flow system

The Flow system receives flow-graphs, coded in a special purpose language, and establishes controller processes for those flow-graphs. It gets a list of computers that can serve as the legitimate "hunting field" for computing resources. Every such computer has a server process running on it. Whenever required, a job can be sent to an appropriate controller, via some interface program. It will start executing by running the flow-graph for the job and sending its processes over to the server pool for execution. The operation of the system can be monitored and controlled by a set of interface programs.

3.1 The Flow daemons

The Flow system is built of two major daemons: controller and server. The Flow controller has 3 major tasks:

- Interpretation and execution of a flow-graph. All meta-level actions are carried out within the controller.
- Resource allocation and centralized load balancing for the computer resources.
- Information gathering and control functions for the system operation. This is done by exchanging information with interface programs.

Each controller gets a flow-graph which corresponds to its task during its startup and establishes a service of some sort. (There could be a simulation controller, a layout production controller, a layout verification controller and so on.) Such services are publicized among the user community, allowing them simple activation of tasks which used to require much specialized knowledge in the past.

The Flow server process is a daemon running on all the computers that can be used by Flow. Its main goal is to receive atomic processes and carry them out on the host computer. Servers participate in the parameter transfer between the controller and the local process. They implement control functions that are initiated by the controllers, and collect information about the local processes and the local host computer, for the controllers.

3.2 Resource scheduling

Each Flow controller establishes a stand-alone service. It is quite possible to have a Flow system with a single controller. If there is more than one, however, all the controllers use the same pool of resources. Flow-graphs will present resource requirements for each process. Those servers that match the requirements are the potential targets of that process. Out of this group, the least loaded server is supposed to get the job. The controller has some notion of the load of each server, as well as a load limit for each of them. This information is based upon the history of the local controller, with possible contribution of fresh load data from some servers. The decision, therefore, is based upon partial information, and is done in a centralized manner. This can be done, knowing that the servers have a notion of local load, and in case the allocation violates the maximal load, the job will be either forwarded or rejected. This way the scheduling is very fast, as it is done in a single process, without any information gathering phase. The quality of the scheduling is dependent upon the quality of the available information.

As more controllers are added to the system, each controller gets to see a smaller part of the total work done by the whole system. Because of that, the quality of its resource allocation might deteriorate. The quality of resource allocation is affected by the global state data latency (GSDL) and the global rate of events (GROE).

Definition: Global state data latency (GSDL) is the maximal latency between the occurrence of any event and its registration by all controllers.

Definition: Global rate of events (GROE) is the average amount of events in the system per unit of time.

The probability of making a bad allocation grows drastically when GSDL >> 1 / GROE. In this situation the controllers are using stale global data most of the time. This is solved by providing an updated status report of all the active servers, limiting the GSDL to a known value. For that task another daemon named event-manager is added. Event managers can live on some of the computers. Each server reports its status transitions to one of the event managers. All the event managers send periodical updates to all the controllers.

4 Results

The Flow system, in various versions, has been operational at some National Semiconductor sites for over 3 years. It successfully handled huge designs, running sometimes thousands of jobs a day. The methodology coded into Flow covers many aspects of the design process, including cell characterization, layout synthesis and layout verification. In order to demonstrate the performance of Flow runtime system, a typical synthesis job was selected. It was carried out many times on a varying Flow system to allow statistically reliable measurements.

4.1 The test case

A block called DX52M was delivered to the synthesis system as an HDL file. Producing that kind of a block manually (i.e. running by hand all the required programs for the production of layout) typically takes a whole work...
day. The block was produced automatically a total number of 20 times, divided into 4 groups: Working as a single job, competing with other jobs on the same controller, competing with jobs of other controllers both with and without the presence of event-managers.

In the first case the management overhead is minimal and the ability of Flow to shorten design steps is most evident, due to lack of competition. The other cases demonstrate the deterioration of the system in face of varying knowledge about the global state of the system. In the second case, GSDL is zero as all work is being done by the same controller. In the third case GSDL is 30 seconds, as provided by the event-managers. The forth case represents the worst case operation, where the information on the spontaneous actions of other controllers is not available, yielding GSDL = ∞.

Three measures are used: the total time of actual execution, the average time for process allocation and the average load on the server computers.

4.2 Results and discussion

The results are summarized in the following tables. The elapsed time for the serial execution of the job on one of the server computers (Sun SPARC 1) is approximately 2:48 hours. Table 1 shows the execution time and process allocation time for all cases:

**TABLE 1. Execution Time and Allocation Time**

<table>
<thead>
<tr>
<th>Measure</th>
<th>Case 1</th>
<th>Case 2</th>
<th>Case 3</th>
<th>Case 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>GSDL (sec)</td>
<td>0</td>
<td>0</td>
<td>30</td>
<td>∞</td>
</tr>
<tr>
<td>GROE (min⁻¹)</td>
<td>2.2</td>
<td>21.9</td>
<td>22.1</td>
<td>23.7</td>
</tr>
<tr>
<td>Execution Time (h)</td>
<td>1:32:04</td>
<td>1:32:52</td>
<td>1:34:16</td>
<td>1:55:12</td>
</tr>
<tr>
<td>Normalized Time</td>
<td>1</td>
<td>1.009</td>
<td>1.024</td>
<td>1.251</td>
</tr>
<tr>
<td>Allocation Time(min)</td>
<td>0.01</td>
<td>0.01</td>
<td>0.04</td>
<td>0.42</td>
</tr>
</tbody>
</table>

Table 2 shows the average load and standard deviation on the computers used during the test. (Load is measured as number of active processes per CPU.)

**TABLE 2. Average Load**

<table>
<thead>
<tr>
<th>Measure</th>
<th>Case 1</th>
<th>Case 2</th>
<th>Case 3</th>
<th>Case 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>0.18</td>
<td>1.14</td>
<td>1.15</td>
<td>1.02</td>
</tr>
</tbody>
</table>

The run-time results demonstrate the robustness of Flow in face of partial global state information. It implies that Flow usage pays off even when the system is working without any cross-talk between controllers. Figure 2 displays the normalized runtime and management overhead vs. the global rate of events.

**Figure 2 Normalized runtime and management overhead**

There is an obvious turning point in the graph, when the event-managers are missing. On the left side of this point is the region where centralized allocation wins a hit in high probability, yielding allocation time of a few seconds. On the right side centralized allocation usually fails, and the distributed scheme takes over. This scheme is much more sensitive to the actual system load and takes much longer.

**Figure 3 Load average**

The quality of the load balancing is demonstrated by the second table. It gives only a rough estimate, as a mean
value over a long period of time. It shows that the quality of load balancing improves as the overall load of the system grows. However, when the state data latency is too big, the tendency is reversed.

5 Summary

5.1 Related work

There are a few systems that capture and code the design methodology. One such tool is Flowmap of the Nelsis framework, developed at the university of Delft [8], featuring both methodology coding and design state tracing. Activation of each of the steps is manual, however, and parallel execution is not supported.

A system which somewhat resembles the design modeling used in Flow, is the one suggested by van den Hammer [9]. The design is turned into a datamflow graph and it can supply information about the state of the design. The system supports versioning by keeping the evolution path of each datum. It supports automatic activation, but does not attempt concurrent execution.

A system which emphasizes the capturing of emergent working methods (rather than rigorously defined methodology), is VOV of Berkeley [10]. VOV allows repeating methods that where defined simply by example. It does not support sophisticated parametric methods, and though it supports automatic parallel execution, the control mechanisms suggested are very basic.

The system that resembles Flow most is MCCs Methodology Management System[11]. It features both methodology definition and concurrent execution control. The major differences are in the semantics used for methodology definition - which is procedural, rather than data-flow oriented, and the load balancing tactics, which is fully distributed, in contrast to the mixed centralized-distributed scheme used by Flow.

5.2 Conclusions

In this paper, the Flow system was discussed both as a methodology definition tool and a parallel execution system. Some of our less formal observations about the system follow.

Flow does have a major problem with very long processes. Some applications use tools that can run for many hours. During that period, there is no effective way to reclaim the resource. This would be solved best, by migrating the process around. Unfortunately, process migration is not very practical for jobs that take up big virtual space. Another shortcome of Flow is the loss of job execution data when a controller crashes. This has not been a major problem in our experience, but this is definitely the weakest point in the system, in terms of fault tolerance. Further work will be aimed at distributing the control work throughout the servers.

The fact that Flow encapsulates design tasks and make them into "services" which are simple to use has proven to be advantageous. Introducing new users to the design environment has become easier, while the quality of the work is ensured, in the sense that there is no ill-usage of the tools in the design parts covered by Flow.

6 References


