Caching of Intermediate Results in Dataflow Environments

Eli Steenput, Yves Rolain
VUB Dienst ELEC
Pleinlaan 2
B-1050 Brussels (Belgium)
Phone +(32 2) 629 28 44
Fax +(32 2) 629 28 50
Email esteenpu@vub.ac.be

ABSTRACT
A measurement is usually repeatedly executed with slightly different settings. Caching of intermediate results can prevent redundant execution of operations whose arguments remain unchanged between consecutive runs. Dataflow environments lend themselves well to such an optimisation, because data dependency information can make comparisons of cache values largely unnecessary. This paper discusses and compares several caching strategies for both the datadriven and the demanddriven dataflow execution scheme. Simulated performance measures are presented to illustrate the benefits of the various schemes.

1. Introduction

The introduction of the computer has dramatically increased the complexity of the measurement environment. The interactive use of instruments is relatively straightforward and efficient for the laboratory scientist, technician, or engineer, but the automation of measurements is not. As a rather unique characteristic of scientific measurement and modelling, the user is often directly involved in the design and implementation of the programs to automate his tests or experiments.

During the verification and modification phase, the developer will repeatedly change some of the measurement settings to observe their effect on the measurement result. However, the measurement itself and the processing of the measured data are often very time-consuming. It is a natural reaction to try to minimise the resulting idle time for the developer.

If only a few of the settings are modified, it is tempting to execute only the operations which depend on the changed settings. It is unreasonable to , for example, repeat a lengthy measurement just to use a different windowing algorithm. The developer could naievely attempt to make the execution of various parts of the algorithm conditional, depending on the state of the settings.

This approach has a huge drawback. The partitioning of the program into conditional parts must reflect the data dependency between the parts and the measurement settings. However, any change to the program is liable to change these relations. Keeping the conditions consistent with the program dependencies places an excessive burden on the developer and compromises program readability.

With parts of the code being skipped during execution, the relationship between the settings and the results will become obtuse and indecipherable. If the user doubts whether or not the results correspond to the latest settings, he will be tempted to run the program more often than needed, possibly nullifying the intended time savings.

1.1 Caching Intermediate Results

To prevent redundant operations during repeated execution of a program, it is necessary to cache the intermediate results of the parts that might be skipped. Aditionnaly, a method is required to determine if a specific program part needs to be executed for a specific state of the program's inputs, or if a previous value is still valid. This can be implemented in various ways, such as by storing previous results in a table indexed by argument values, which is proposed in [AU77] and an example of its implementation can be found in the Maple mathematical software [C88]. The Maple remember option associates with each function a table of previous arguments and corresponding results. Before executing a function, the table is checked for the presence of the given combination of argument values. If the arguments are already in the table, the result is retrieved. Otherwise, the function is executed and the table is updated. If sufficient memory is available, this scheme will reduce the number of operations to a nearly optimum level.

There is a lookup or comparison overhead inherent in any caching scheme. This overhead will depend on the data size and the number of previous results stored in the cache.

1.2 Dataflow Environments

In a dataflow environment, the available data dependency information can be used to lower the lookup overhead of the argument values. The fact that a particular program part was skipped provides information about the need to execute those sections that use the output values of the skipped part as arguments.

A dataflow graph is a directed graph, where each node (or module, box, vertex) represents an operator or function (a task) and each directed arc (or link, wire, edge) represents a path (dependency) over which data flows. Data flow graphs were introduced to express data dependency between tasks, where a task stands for a data transformation. A dataflow representation of a program is a data structure that can be rapidly traversed to determine the data dependency information between tasks. By introducing a dataflow execution mechanism, this representation becomes a program in its own right, with a parallel, local model of execution.

Dataflow is a partially ordered model of computation. Contrary to control flow graphs, which tend to overspecify a computation by imposing total ordering on its operations, dataflow specifies a partial order based on data dependencies.

Therefor, all task dependencies that are usually implicit in other representations must be explicitly represented as data flow to ensure that data changing operations occur in the correct order. Pure dataflow graphs do not include the notion of variables, as there are no named memory cells holding values. Computation supposedly does not produce side effects. Flow control is not part of pure dataflow. Programming environments must provide extensions to overcome these limitations to be able to use dataflow as a computational model. Proper execution order of data changing operations, and the handling of conditional and iterative constructions are among the most important.

1.2.1 Data Driven Dataflow

Out of the two possible modes of execution for a dataflow program [H92], datadriven execution is used the most in dataflow environments. In data-driven execution schemes, a node (which represents an operation in a dataflow graph) will execute as soon as all of its inputs have received a data token (a packet of data). Upon execution, the node produces one data token on each of its output arcs. The data tokens on the input arcs are lost once the node has generated an output. Data tokens flow through the graph, initiating calculations, until each data token reaches an output terminal.

Assume each node has an associated cache that stores the value of the last passing data token. Instead of immediately re-executing when its inputs arrive, a node would first determine whether its inputs have changed. If they did not, it can simply retrieve its previous result from the data cache. If a change occurred, the node re-executes, placing its new result into the cache.

The input terminal or executing node that places a data token on an arc can compare it with the cached value, and add a tag to the token if the value did not change. A node that has only unchanged tokens on its input arcs does not have to execute, its cached data token can be re-issued with an 'unchanged' tag. The use of tags makes comparison with previous values only required for newly calculated results.

This simple scheme prevents needless execution of operations that do not depend on a changed input. It even stops execution on paths depending on a changed input, if the intermediate operations yield the same results. This has proven useful, as relational operators for example often produce the same result for different input values.

Note that tokens are compared after each operation that was actually executed. If all input terminals have unchanged values, not a single comparison takes place during traversal of the graph beyond the input terminals. As there is a cost for comparing the data tokens, the usefulness of caching will depend on whether the savings achieved by performing fewer operations exceed the overhead incurred.

If the possibility of nodes with a different input yielding the same output is disregarded, no comparisons at all are needed downstream of the input terminals. Also, in this case, not all intermediate results need to be stored. A *thread* is defined as a number of connected nodes that all depend on the same set of input terminals. When one of the nodes in a thread has to be recalculated, all of them require recalculation. Under these assumptions, only the 'end' nodes of the thread (which deliver arguments to nodes not belonging to the thread) benefit from a cache.

It is assumed that a node that has just executed with a given set of input values, has no reason to do so again. The scheme is not limited by this assumption, though. The program still executes in a data-driven way, in that all nodes are 'visited' by data tokens. If required, an arc could be marked to execute the next node without regard for the 'newness' of its data.

One could worry that keeping the intermediate results will require excessive amounts of extra memory. But the data tokens are often stored anyway in dataflow environments. As a node generates a token, it dynamically allocates the memory needed. The memory is not deallocated until the next token is generated by the node. In such an implementation of a dataflow environment, the scheme would not increase the memory requirements of the system.

1.2.2 Demand Driven Dataflow

For demand driven execution, execution of a node happens when downstream nodes request data from the node's output arcs. If the requested value is available, it can be returned immediately. If the value is not immediately available, then it must be computed. If the node requires input data to execute, it will send data requests to its arguments. The node will wait for these arguments to send it data. After execution, it will send the results over its output arcs to the downstream nodes that made the request.

In demanddriven dataflow there is a two-way traffic in the communication lines. In one direction data flows in the usual way from input to output. In the other direction demands are sent, from the output terminals upstream to the inputs. To the authors' knowledge, the "natural" implementation where the nodes transmit demands to each other and send data in return, has hardly ever been used in any existing environment, due to its great complexity. The propagation of demands suggested in the theoretical model can be implemented, but is very awkward on extensions of pure dataflow like iteration constructions.

A practical implementation of "demand driven" execution (as in [DX97], for example) will walk through the network, starting from the requested result, to 'prime' the arcs encountered on the way. After this preparatory step, the 'primed' part of the graph will be executed using what is basically the datadriven execution scheme.

Since a node in the demanddriven scheme will not always execute when its input data becomes available, (because it was not in demand at the time) it is not sufficient to look at the 'unchanged' status of the arriving data tokens to determine if the node has to be updated. Assume a clock is incremented each time an input is changed, and every node or terminal is tagged with the time value of its last change. A node is up-to-date - and hence needs no recalculation - if its parent nodes are consistent and none are newer than the node itself.

Time stamps and 'unchanged' tags alone can effectively avoid many unneeded operations, with little overhead. Only the values at the input terminals have to be compared to their previous values. Because a new, changed argument can still lead to the same results as the previous value, additionally comparing the results of intermediate operations with their previous values is considered worthwhile.

1.2.3 Control Constructions

Because of the absence of control flow information, some program constructions can not be represented by standard dataflow graphs. Sequential execution, procedural abstraction and iteration are deemed necessary to tackle complex problems. Every dataflow-based environment must therefore provide some extensions to implement these constructions.

Among the constructions not supported by pure dataflow are: functions/procedures, sequential execution, recursion, iteration, and conditional execution.

These can be implemented in many different ways. Most implementations can be recognised as belonging to one of two different approaches to implement control. The first approach isolates a part of the graph (a subgraph), so a control mechanism can operate on the subgraph. The subgraph is sometimes represented as being enclosed by the controlling node. An example of this would be a subgraph contained in a "while-structure" node. The control node will operate on the subgraph in ways that can not be expressed in the pure dataflow formalism, but will behave like any other node to the rest of the graph.

The second approach uses the dataflow execution mechanism to provide control. Iteration and conditional execution can be implemented by introducing selector (or merge) and distributor (or switch) nodes (and cycles in the dataflow graph for iteration). A selector accepts a true or false control token to decide which of two inputs should be propagated to its output. A distributor uses the control token to pick an output arc to put its single input value on. These nodes can act as 'valves', cycling a token repeatedly through some part of the graph. This mechanism sometimes requires certain arcs to be 'initialised' with a starter token.

An advantage of the second approach is the more consistent model, whereas in the first different execution mechanisms are mixed. The second approach offers the virtues of a uniform data model and provides for sufficient control of operations to build realistic applications. However, it operates at a low level with very basic computational primitives. The greater flexibility comes at the cost of a larger learning curve and greater effort to build complex operations. The first approach produces programs that are easier to understand, and most environments prefer this approach for clarity.

2. Performance Evaluation of Dataflow Execution Schemes

The performance and overhead of a specific scheme will not only depend on the properties of the dataflow graphs (such as the distribution of compute times and data sizes, the branching factor of the arcs and the graph connectivity), but also on the unpredictable sequence of user interactions. The following section studies the feasibility of evaluating the effect of different execution schemes and buffering on a given algorithm.

2.1 Analytic evaluation of a simple Network Analyser example

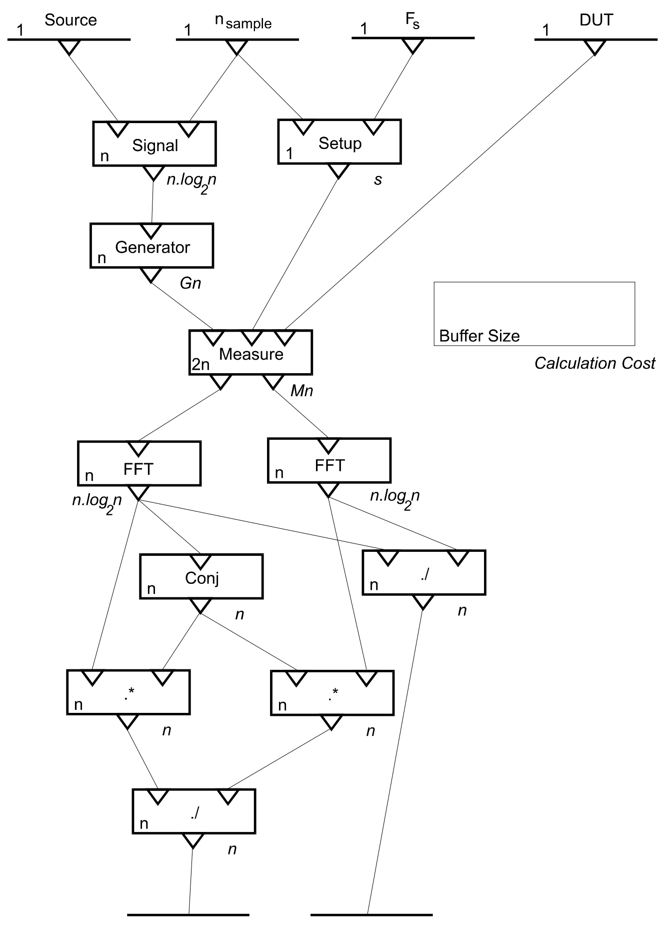

Figure 1 A Simple Network Analyser

This simple network analyser will produce the transfer function or power spectrum of a device under test (DUT). This example immediately illustrates the difficulty of comparing datadriven and demanddriven execution. The use of two outputs is only sensible in a demanddriven environment. For datadriven execution, the programmer will most likely insert a condition immediately following the FFT nodes to select one of the outputs.

This 15 node graph has 4 input terminals (inputfraction 27%). There are 7 threads, which makes the average threadlength 2.1. If we break up this total, we find 5 1-node threads, 1 2-node thread and 1 8-node thread. There is a distinct disproportion of threadlength before and after the measurement. The average number of output arcs is 1.4.

The calculation costs and buffer sizes are indicated on Figure 1. The cost *s* for set-up of the measurement equipment, *Gn* for loading the signal into the generator and *Mn* for the measurement pose a problem to computing a performance, as these will depend on such factors as the hardware and the sample period.

There are 15 possible combinations of input status.

- Ten lead to complete recalculation, the total cost is $Gn+3n\log_2 n+Mn+s+2.5n\ (+2.5n)_{datadriven}$, with n the number of samples,
- one has a cost of $2n\log_2 n+Mn+2.5n\ (+2.5n)_{datadriven}$,
- two a cost of $2n\log_2 n+Mn+s+2.5n\ (+2.5n)_{datadriven}$, and
- two a cost of $Gn+3n\log_2 n+Mn+2.5n\ (+2.5n)_{datadriven}$.

If the input changes are stochastically independent, the average required calculation cost is $0.8Gn+2.8n\log_2 n+Mn+0.8s+2.5n\ (+2.5n)_{datadriven}$.

None of the nodes are likely to produce the same result given a changed argument, so comparing the input terminal values (all scalar) will avoid any redundant operation.

As an example, assume $n = 10^5$, $M = 10$, $G = 0.5$ and $s = 10^4$.
This yields the following costs (overhead is ignored):
- total cost is 6292892 demanddriven, 6542892 datadriven
- buffered cost is 5948699 demanddriven, 6198699 datadriven

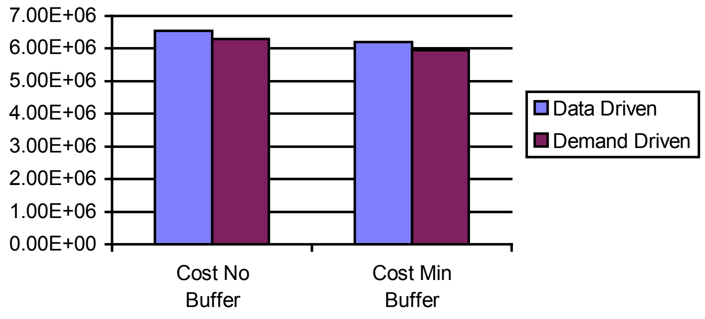

Figure 2 Overview of average costs

This example illustrates some of the difficulties encountered and the assumptions required when attempting to calculate the performance of an execution scheme for a specific algorithm.

The apparently small benefit gained from caching (or from demand driven execution) can be attributed to the disparity in threadlength in the graph structure. Due to the simplicity of the example, the input settings affect only the measurement itself, not the data processing,. More than half of the nodes depend on all input terminals, and are thus executed for each input change. Caching is nevertheless found useful in data visualisation environments [DX97]. This suggests input terminals should be distributed more or less evenly over the graph to make caching profitable.

2.2 Simulations

Related research [CG92, SCNPW93, WS95] suggests the use of simulations, using random generated graphs and user interactions, to obtain performance measures. This method was adapted to several dataflow execution schemes, including those from [SR96, SR97, SR97bis].

A simulator was created to calculate the performance and overhead for the following caching schemes:

1. no caching, no comparisons to previous values
2. caching the end nodes of threads, comparing values assigned to input terminals with their previous value
3. caching and comparing values at the end nodes of threads
4. caching and comparing values at every node

In the previous discussion only the last results are stored in the cache. A node could cache more than just the last value (as in [DX97]), thus increasing the probability of finding a result in the cache, but this method was not explicitly considered in the simulation.

2.2.1 Creating the Graphs

Graphs are created randomly with a predetermined number of nodes and number of input terminals. For each node a size and a calculation cost is chosen randomly from a given range. Conditional and iterative functions are not modelled, but each node has a randomly associated probability of yielding the same result on execution (this could alternatively be thought of as a cache hit). The maximum value of this probability, the ChanceFactor, is chosen on graph level.

From the input terminals, links are made to randomly selected graphs. Each node has at least one output, the exact number of outputs is random due to the graph construction process.

Then nodes that already have an input are randomly connected to nodes that do not yet have outputs - to avoid loops - until all nodes (except one) have at least one output. The number of links per node is again determined randomly based on the BranchingFactor. The last remaining node is removed, and its arguments are chosen as the output nodes of the graph.

2.2.2 Creating an Interaction

Simulating data driven dataflow is relatively straightforward. Some of the inputs are marked as changed, all the inputs get an activation token, and the tokens are propagated through the graph. To evaluate the effect of caching, the simulator must determine which nodes can make use of a cached value. In a datadriven scheme, these are the nodes whose arguments are the same as before.

The simulator compares datadriven and demanddriven execution of the same graphs, with the same pattern of changed input terminals. It can be argued though that a demanddriven environment will prompt the use of a different 'programming style' than a datadriven environment. However, this can hardly be taken into account in a simulation.

A demanddriven scheme is only relevant when the system has more than one output node. To obtain graphs with several outputs, it was sufficient to discard the single output node of the graphs produced by the graph generation algorithm, and use its arguments as output nodes. It must be noted, however, that some of the outputs thus obtained may require very little calculation. Without statistical information on typical application graphs, it is impossible to determine how this compares to a realistic output cost distribution.

For each interaction, one of the outputs is selected at random, and all its downstream arcs are 'primed'. Only activation tokens passing through primed arcs will influence the performance data collected on demanddriven execution. Arcs are 'unprimed' once an activation token has passed through them.

In the demanddriven mode, only part of the graph will reflect the current situation of the input terminals at a given time. The nodes that do not contribute to the selected result are not updated when their arguments change. The 'unchanged' status of the arguments is therefore no longer sufficient to determine whether a node can reuse its previous value or has to be recomputed. The simulator handles this by having a node mark each of its output arcs when its value changes, and unmark each of its input arcs when it is passed by an activation token (this works because there are no loops in the simulated graphs). That way a node can determine whether the value of an argument changed since its last use by the node (instead of since its last calculation, which is reflected by the 'changed' tag).

As with datadriven execution, several comparison strategies are possible. If only the values assigned to input terminals are checked for change, the required cache space can be decreased. The only nodes that need to cache their value are those who are argument to a node depending on a larger set of input terminals as its individual arguments. Those nodes, forming the outputs of a 'thread' depending on the same inputs, are marked during the graph generation phase by compiling and comparing the set of input terminals each node depends on.

The effects of several graph properties were studied: graph size, relative number of inputs, the branching factor, the threadlength and the chancefactor. The values under study are the calculation cost and the comparison overhead. The calculation cost is subdivided in CostNoBuffer (the cost for executing the entire graph with no caching), CostMinBuffer (the cost for executing the graph when taking into account which input terminals have new values), CostMaxBuffer (the cost for executing the graph when each intermediate result is compared with its previous value), and CostMedBuffer (compares results at the end of a thread). The simulated costs are corrected for graph size and average graph cost.

The exact overhead is difficult to determine, and the data size is taken as a worst case estimate. OverheadMinBuffer is the accumulated data size of the input terminals, OverheadMaxBuffer is the accumulated data size of all nodes that are executed, and OverheadMedBuffer is the compare overhead for results on the end of threads.

2.2.3 Simulation Results

2.2.3.1 Datadriven Scheme

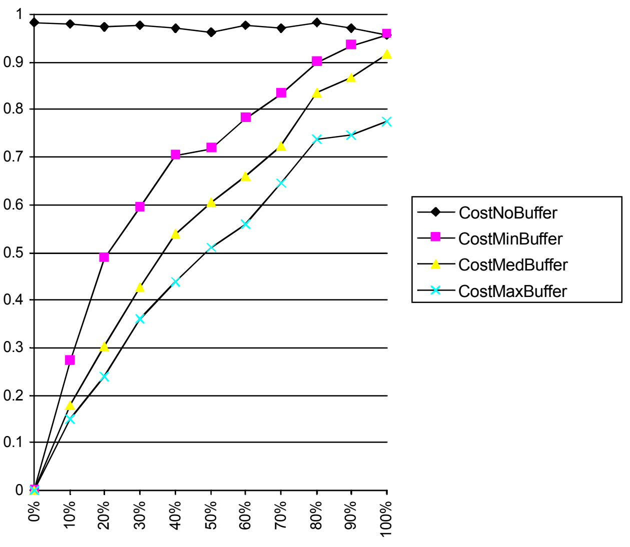

Figure 3 Cost vs. Changedfraction

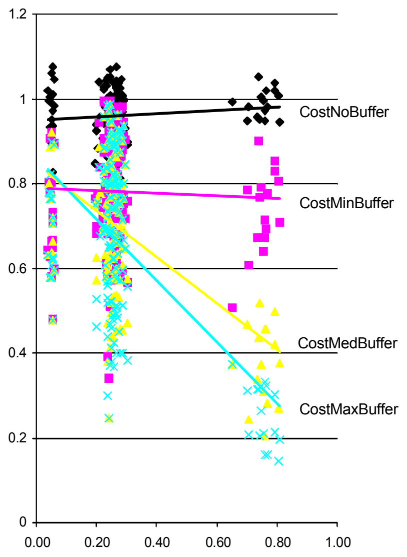

Figure 4 Cost vs. Chancefactor

The first obvious parameter affecting the cost is the fraction of changed input terminals (Figure 3). CostMinBuffer varies from 0% of CostNoBuffer when no inputs have changed to 100% when all inputs have changed, and is not very sensitive to the graph structure. CostMaxBuffer will be equal to CostMinBuffer in the worst case, but is usually smaller. CostMaxBuffer can be smaller than CostNoBuffer even when all inputs have changed.

The amount of performance increase depends on the probability of the nodes producing a changed result, and on the graph structure. A low chancefactor (corresponding to most nodes producing changed values) will yield a CostMaxBuffer as high as CostMinBuffer (Figure 4).

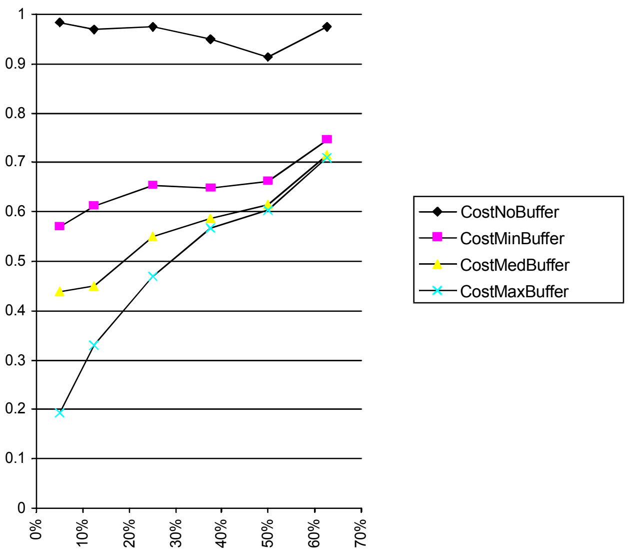

Figure 5 Cost vs. Inputfraction

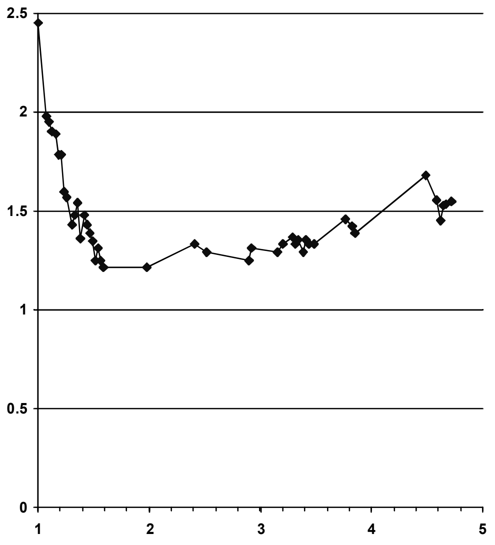

Figure 6 Threadlength vs. Branchingfactor

Contrary to expectation, a graph with only a few inputs still benefits from the maximum buffer scheme. In fact, the higher the inputfraction, the higher the cost (Figure 5). The determining factor here seems to be the threadlength, which is the average number of nodes per thread. Equally-sized graphs with fewer inputs tend to have longer threads, which makes the influence of one unchanged result more pronounced. The same effect, but to a lesser extent applies to CostMinBuffer. CostMaxBuffer can be dramatically low for low inputfractions.

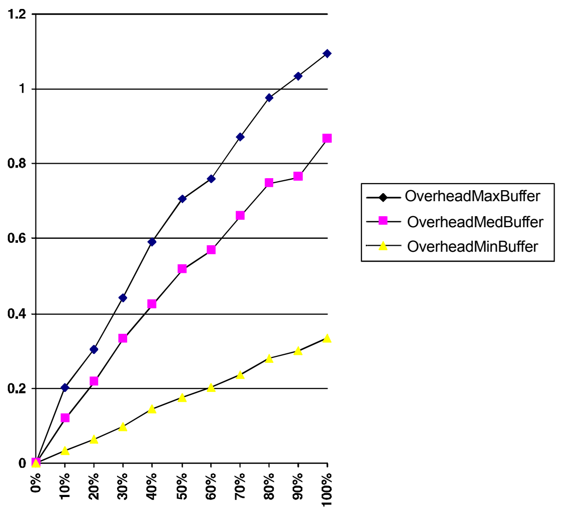

Figure 7 Overhead vs. Changedfraction

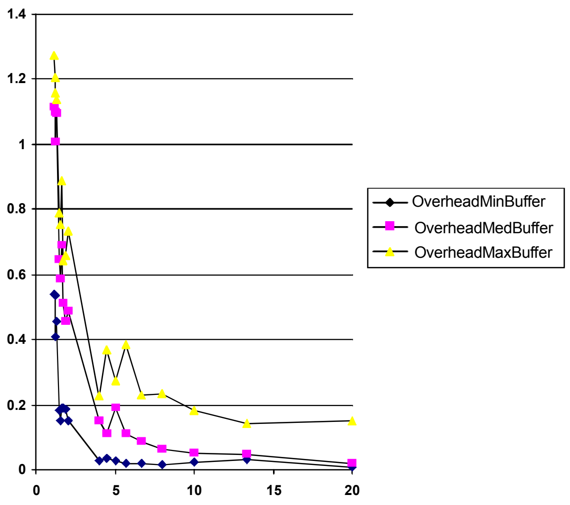

Figure 8 Overhead vs. Threadlength

The branchingfactor influences the cost through its influence on the threadlength (Figure 6). With increasing branchfactor, the threadlength decreases to a minimum, then increases again (presumably because due to the high level of interconnection, most nodes will ultimately depend on all input terminals).

The cost of course increases with graph size, but the relative costs are almost constant under size variation.

The overhead is taken to be the cost of comparing the new values with previous ones. MinCompareOverhead is the overhead for comparing only the values of the input terminals. This value increases with the fraction of changed inputs (Figure 7), and is higher on the average for graphs with higher input fraction (not depicted). MaxCompareOverhead is the overhead when after each operation, the new result is compared with the previous value. MeanCompareOverhead is the overhead from comparing values at the end of threads. All overhead decreases fast with increasing threadlength (Figure 8).

2.2.3.2 Demanddriven Scheme

The total cost is no longer constant in the demand driven scheme, and is always lower than in the datadriven scheme. The cost decreases with the number of input terminals, and with the number of outputs. Both factors probably reduce the relative number of nodes an average output depends on. The Maximum test cost increases with the inputfraction for low inputfractions, and follows the total cost for higher inputfractions (Figure 9).

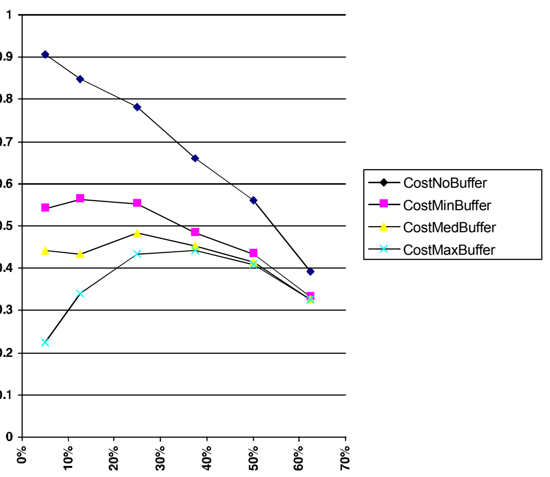

Figure 9 Demanddriven Cost vs. Inputfraction

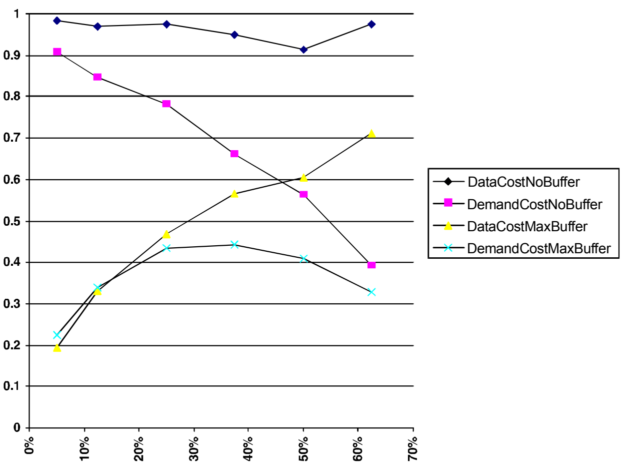

Figure 10 Datadriven and Demanddriven Cost

It is interesting to note that the datadriven MaxTestCost is lower than the demanddriven cost for low inputfractions and a not too low chancefactor (Figure 10). This seems counterintuitive and deserves an explanation. Indeed, one would assume that because the datadriven scheme always executes the entire graph, the cost must always be higher than the demanddriven execution cost. It appears however that under certain conditions the datadriven mode can make better use of nodes producing the same output for different arguments.

The demanddriven overhead is lower than the datadriven overhead, except for MaxCompareOverhead for low values of the inputfraction (Figure 12). The difference in overhead between datadriven and demanddriven execution increases with increasing inputfraction.

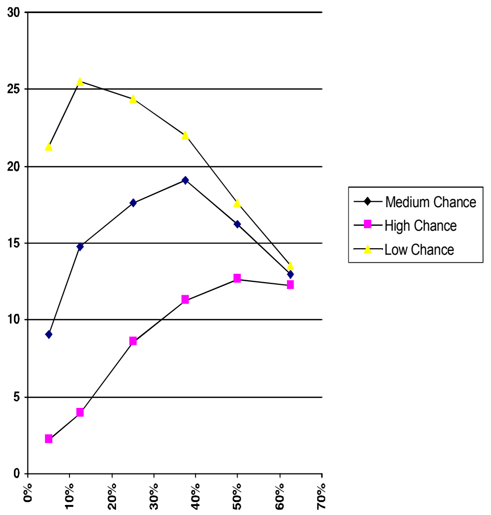

Figure 11 Extra Overhead vs. inputfraction

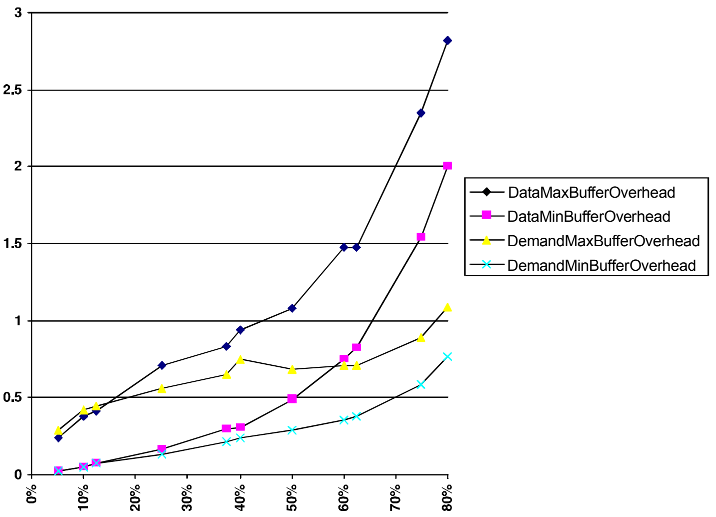

Figure 12 Datadriven and Demanddriven Overhead vs. Inputfraction

The extra overhead generated by the maximum test scheme as compared to the minimum test scheme (Figure 11) increases with inputfraction for a high chancefactor but decreases for low chancefactor. This is the same for both datadriven and demanddriven execution.

2.2.4 Summary

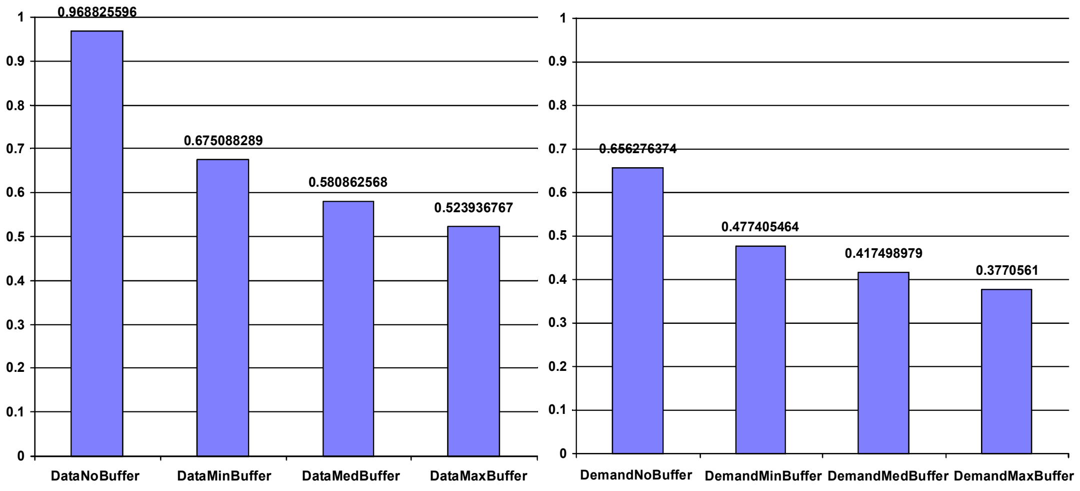

Figure 13 Comparison of average costs for datadriven and demanddriven execution

Averaging all simulations, demanddriven execution indeed displays a lower execution cost on the average than datadriven execution, for all studied modes of caching, though exceptions were encountered for specific conditions (Figure 13). Checking the input terminals for change reduces the cost to about the same extent. The cost reduction obtained by caching intermediate results appears less dramatic, but this can be partly caused by the unrealistically high average inputfraction.

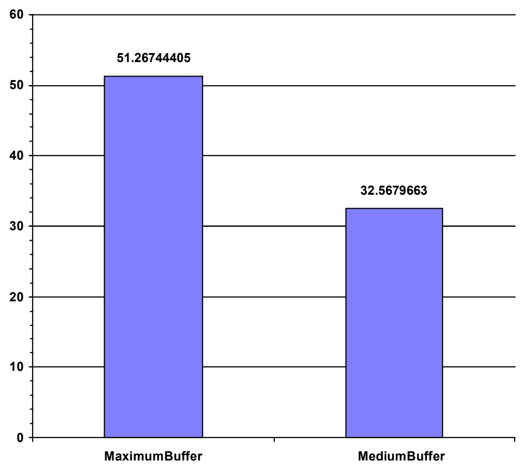

Figure 14 Cachesizes for maximum and medium buffer scheme

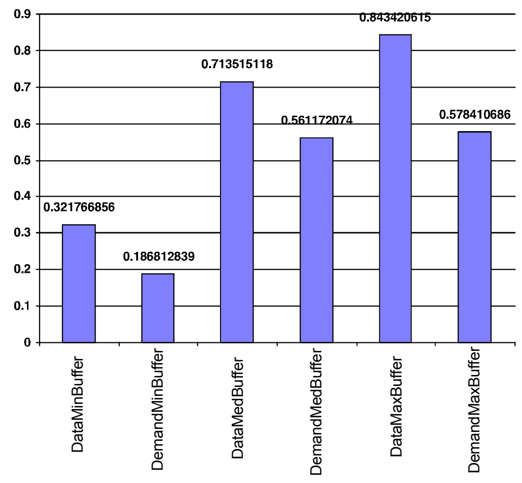

Figure 15 Comparison of average overhead for datadriven and demanddriven execution

The trade-off between cost reduction and comparison overhead will depend on how these factors compare for a specific system. As can be seen in Figure 16, general conclusions are not straightforward. It appears the medium buffer scheme is never cheaper than the maximum buffer scheme for demanddriven execution, and for datadriven only when caching is useless anyway.

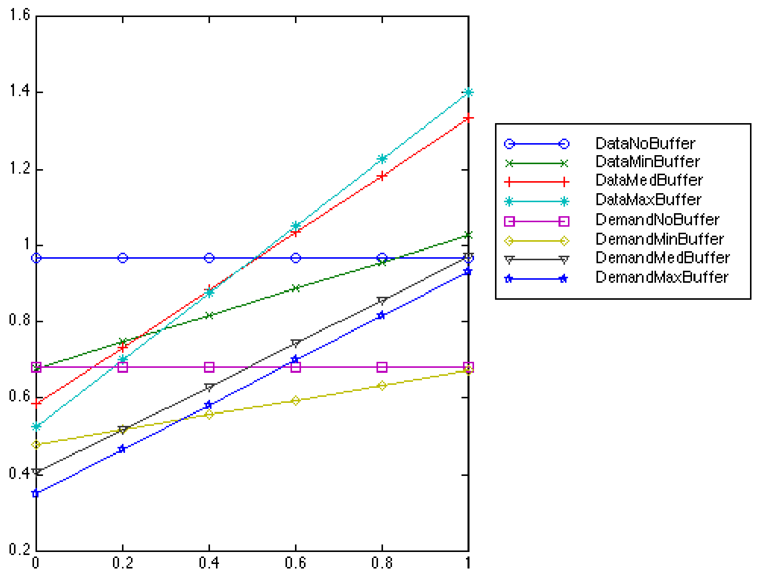

Figure 16 Total Cost (calculation + overhead) for increasing relative comparison cost.

However, this result only represents the simulated population of graphs. By considering a subset of low chancefactor, high inputfraction (Figure 17) or high chancefactor, low inputfraction (Figure 18), very different results are obtained.

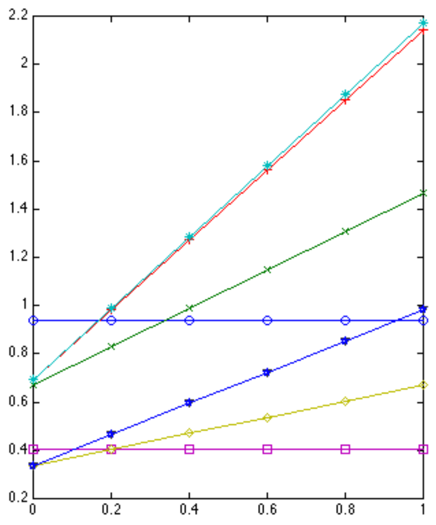

Figure 17 Total cost (low chance, high input)

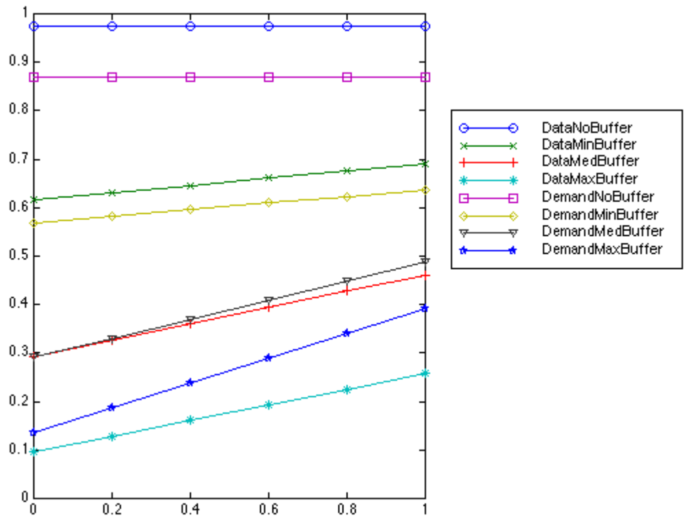

Figure 18 Total cost (high chance, low input)

3. Conclusion

Various execution schemes for dataflow environments are often presented without evidence concerning their alleged benefits. This paper demonstrated the use of simulated execution on large numbers of graphs as a possible remedy to this situation. It is shown that useful insights into the influence of graph properties can be gained from the simulation results. Demanddriven and datadriven execution with various caching strategies were compared. The simulations show that none of the scheme under study is consistently superior in all cases. The *thread length* of a graph was introduced as a useful concept to explain the influence of graph topology on the performance. The analysis of a simple example suggests that, to realise the full benefit of caching, user interactions should be evenly distributed over the graph.

References

[AU77] A. V. Aho, J. D. Ullman, Principles of Compiler Design, Addison-Wesley, 1977.

[C88] B.W. Char, Maple users guide. Waterloo, Ontario, WATCOM publications, 1988.

[CG92] V. K. Chaudhri, R. Greiner, "A Formal Analysis of Solution Caching", Proceedings of the Canadian Artificial Intelligence Conference, 11-15 May, 1992, Vancouver.

[DX97] IBM Visualisation Data Explorer User's Guide 3.1.4, IBM Corporation, May 1997.

[H92] D. D. Hils, “Visual Languages and Computing Survey: Data Flow Visual Programming Languages”, Journal of Visual Languages and Computing (1992) 3, 69-101.

[H95] How to Use HP VEE, Edition 1, Jan. 1995, Hewlett Packard.

[SCNPW93] M. Stonebraker, J. Chen, N. Nathan, C. Paxson, J. Wu, “Tioga: Providing Data Management Support for Scientific Visualisation Applications”, Proceedings 19th International Conference on Very Large Data Bases, Dublin, 1993, 25-38.

[SR96] E. Steenput, Y. Rolain, “Auto-Consistent Mathematical Environment for Measurement Software Development”, Proceedings of the IEEE Instrumentation and Measurement Technology Conference, Brussels, Belgium, June 4-6, 1996, Volume I, pp 21-26.

[SR97] E. Steenput, Y. Rolain, “Data Consistency and Redundant Operations in Measurement System Development”, Proceedings of the IEEE Workshop on Emergent Technologies & Virtual Systems for Instrumentation and Measurement, Niagara Falls, Ontario, Canada, May 15-16, 1997, pp 112-117.

[SR97bis] E. Steenput, Y. Rolain, “Auto-Consistent Environment for Measurement Software Development”, IEEE Transactions on Instrumentation and Measurement, Volume 46, number 4, August 1997, pp 742-746.

[WS95] A. Woodruff, M. Stonebraker, "Caching of Intermediate Results in Dataflow Diagrams," Proceedings of the 11th IEEE Symposium on Visual Languages, Darmstadt, Germany, September 1995.