CACHING IN DATAFLOW-BASED ENVIRONMENTS

Eli Steenput, Yves Rolain
Dept ELEC
Vrije Universiteit Brussel
Pleinlaan 2
1050 Brussel
Belgium

ABSTRACT
Caching of intermediate results can prevent redundant operations during repeated executions of a measurement set-up with slightly different settings. Dataflow information, as available in dataflow-based virtual instrumentation environments, can be used to reduce cache overhead and to limit the cache space required. Different methods are presented to determine the caching scheme best suited for a particular algorithm, based on dataflow graph properties.[1]

1. Introduction

A measurement set-up is often repeatedly executed with slightly different settings for the measurement, data processing and representation. In the development phase, these changes can include adjustments to the program. During repeated executions, some time-consuming operations may be executed more often than necessary, resulting in avoidable idle time for the developer.

Redundant operations can be avoided by making the execution of various parts of the algorithm conditional, depending on the state of the input settings. However, the partitioning of the program into conditional parts must reflect the data dependency between these parts and the inputs. Any change to the program is liable to change these relations. Manually keeping the conditions consistent with the program dependencies places an excessive burden on the developer and compromises program readability. If errors occur in these conditions, the user may start doubting whether the obtained results are based on the latest settings and be tempted to recalculate the results, thus defeating the purpose of avoiding redundant operations.

To prevent the redundant execution of an operation, its previous arguments and the corresponding results must be stored in a data cache. Before executing an operation, the cache is checked for the presence of the current combination of argument values. If the arguments are already in the cache, the corresponding result is retrieved. Otherwise, the operation is executed and the cache is updated. If sufficient memory is available, this scheme will reduce the number of operations to a nearly optimum level. Because data can be quite large in a measurement or modelling environment, caching all intermediate results is not always feasible.

Any caching scheme also has an inherent lookup or comparison overhead. This overhead will depend on the data size and the number of previous results stored in the cache.

[1] This work was supported by the Flemish Government (GOA-IMMI), and the Belgian Program on Interuniversity Poles of Attraction initiated by the Belgian State, Prime Minister's Office, Science Policy Programming (IUAP4/2).

Data dependency information can be used to reduce the necessary cache size and cache lookup overhead. In virtual instrumentation environments based on dataflow diagrams, this information is readily available.

2. Dataflow-based environments

The data dependencies in an algorithm are commonly represented by a dataflow graph, a directed acyclic graph, consisting of nodes, arcs and terminals. The nodes represent tasks to be performed, arcs are directed paths over which data is exchanged between nodes, and terminals are the connections to the external world. By introducing a dataflow execution mechanism, this representation can become a program in its own right.

Two possible modes of execution for a dataflow program exist. In data driven execution, a node will execute as soon as all of its inputs have received a data token (a packet of data). The node cannot execute until all of its inputs have a data token available. Upon execution, the data tokens on the input arcs are removed, and the node produces a data token on each of its output arcs.

Suppose each input terminal produces a new data token whenever a value is assigned to it. In a data driven system, a change at one input terminal will only affect an output, if all input terminals the output depends on produce a token simultaneously (whether their values have changed or not). In fact, all input terminals should always produce data tokens together, otherwise “old” data tokens could remain on the arcs, in which case, it would be unclear whether the obtained result is correct, even if the whole graph were recalculated.

The second (less prevalent) execution mode is demand driven dataflow, in which execution of a node happens when downstream nodes request data from the node’s output arcs. If the node requires input data to compute the value, it will send data requests to its arguments, then wait for these arguments to provide data. After execution, the node will send its results to the downstream nodes that issued the request. In a demand driven system, a change at an input terminal will not take effect until a demand for the output value is issued.

Since the pure dataflow paradigm does not include program control, programming environments must provide some extensions to use dataflow as a workable computational model.

We propose that there are several advantages to integrating caching in the execution scheme of a dataflow-based environment. If each node has a cache to store its last result, the cache look-up overhead is eliminated, since each cache contains just one value. A node in a dataflow based program must store its result anyway until its downstream neighbours are ready to execute. Additionally, there is no need to store previous arguments with the cached result for comparison, the validity of a node’s cached value can be determined from the validity of the nodes it depends on.

3. Caching schemes

Different ways to integrate caching in a dataflow environment are possible. The following schemes each compare new values to cached results for a different number of nodes, and require different amounts of cache space.

Once the algorithm is fixed, the validity of a node ultimately depends on the status of the input terminals, and comparing new input settings with the previous values is sufficient to determine the validity of each cached result in the graph. Suppose each input terminal compares a new setting with the previous value. If the new setting is the same as the previous value in the cache, the data is marked "unchanged" before it is passed to other nodes of the dataflow graph. A node that receives only unchanged data on its input arcs does not have to execute. Its cached result is still valid and can be re-issued, with an "unchanged" tag added to it. This simple scheme prevents needless execution of operations that do not depend on a changed input terminal.

Not every node needs to cache its result in this scheme:

A ***thread*** is defined as a set of the largest number of connected nodes depending on an identical sets of input terminals. Each node in the graph belongs to one thread or constitutes a thread by itself.

Nodes in the same thread all depend on the same input terminals. Thus when one of the nodes in a thread has to be recalculated, all of them require recalculation. Only the 'end' nodes of the thread (which deliver arguments to nodes not belonging to the thread) benefit from caching. This permits a reduction of the number of cached values, without reducing caching efficiency. The number of threads in a graph will determine the number of results that require caching, and the required cache space. We call this the ***MinCache*** scheme, referring to the use of the minimum amount of cache space required to avoid the execution of operations depending on unchanged inputs.

The cost reduction that can be achieved by the MinCache scheme, is calculated as follows:

If a node n depends on p input terminals, and assuming all input states are equally likely, each input has a probability of ½ to change, so the probability that none of the p input terminals changed is 2^{-p}, and thus the probability P_n that a node is affected by a change to the input terminals equals $P_n = 1 - 2^{-p}$.

The MinCache calculation cost is obtained by multiplying the cost C_n of each node n with the probability P_n of its execution, and summing the obtained values for all nodes: $C_{total} = \sum_n P_n \cdot C_n$.

Note that this calculation may be inapplicable to some extensions (implementations of program control etc.) of the pure dataflow paradigm.

The only overhead in the scheme consists of comparing new input settings with previous values (data fetching etc. are performed anyway in a dataflow environment).

If nodes sometimes produce the same output for different argument values, it can be useful to compare a newly calculated result with the previous value in the cache (relational operators for example often produce the same result for different input values). Suppose each node (or input terminal) caches its last value and compares a new result with the cached value on execution. If the new result is the same as the previous value in the cache, the result is marked as "unchanged". Nodes with unchanged argument don't have to execute and can return their cached result, also marked as "unchanged".

This scheme stops execution on paths depending on a changed input where intermediate operations yield the same results. Note that cache comparison is limited to newly calculated results. If all input terminals have unchanged values, not a single comparison takes place downstream of the input terminals. Since all intermediate results are cached, we call this the ***MaxCache*** scheme.

Caching all intermediate results is advisable while the algorithm is still being edited. The MaxCache scheme avoids recalculation of nodes that are not affected by a change to the program graph.

To calculate the MaxCache cost, for each node n the probability r_n of producing the same value for changed input arguments must be known.

The total calculation cost is again the sum, for all nodes, of the cost of each node n multiplied by the probability P_n that the node will be executed.

An input terminal has no associated calculation cost, and is not "executed". It can only be assigned a new (changed) value. A node that has input terminals as arguments, must execute if one of the input terminals changed. So for an input i $r_i = 0$, and $P_i = ½$.

A node n is executed if any of its arguments a was executed (probability P_a), and, upon execution, produced a new (changed) value (probability $1-r_a$).

This gives for a node n (input terminals excluded):

$$\overset{\text{Probability that n will be executed}}{\boxed{P_n}} = 1 - \overset{\text{Probability that each argument of n is unchanged}}{\boxed{\prod_{\text{arguments}(n)} \Big(\underset{\text{Probability that argument is executed but unchanged}}{\boxed{P_{\text{argument}} \cdot r_{\text{argument}}}} + \big(\underset{\text{Probability that argument is not executed}}{\boxed{1 - P_{\text{argument}}}} \big) \Big)}}$$

Note that P_n depends on the structure of the subgraph leading to n, and the probabilities r will have to be estimated.

The ***MedCache*** scheme is a hybrid, that caches only the end nodes of threads, like the MinCache scheme, but compares new results for these nodes with the cached values, as in the MaxCache scheme. This way execution on paths depending on a changed input where some of the intermediate operations yield the same results is stopped at the first thread end node. The scheme needs only the minimum required cache space.

The cost for the MedCache scheme can be calculated as in the MaxCache scheme, but since only nodes at the end of a thread can check their result against a cache, the term $P_{\text{argument}} \cdot r_{\text{argument}}$ should be omitted for all other nodes.

4. Performance comparison

Because only a limited number of algorithm graphs are available for analysis, performance measures of the caching schemes were obtained mainly by simulations on random generated graphs. Performance measures consist of calculation cost relative to the total calculation cost without caching, and the caching overhead relative to the caching overhead without using dataflow information.

To compare different graphs and to draw general conclusions, some easy to determine quantitative properties, representative of each graph, must be derived. The properties that affect performance can be divided into node properties, structural properties, and interaction properties.

The node cost and cache lookup overhead can be normalised relative to the graph averages, which leaves the probability to yield the same result for different inputs as the only node property influencing the effectiveness of the caching scheme. The graph average of this probability is called the

ChanceFactor. The ChanceFactor has no effect on the MinCache scheme, but a high ChanceFactor greatly increases performance of the MaxCache and MedCache scheme.

Performance measures can be normalised for the number of nodes. The structural properties of dataflow graphs can be represented by various parameters, and selecting parameters that promote understanding of the factors affecting performance is not obvious. A discussion of the possible parameters and how they affect performance falls beyond the scope of this article. In general, caching is more effective for graphs consisting of few, large threads that each depend on a small number of input terminals.

The user interaction will affect the performance through the number of input terminals that are changed in the interaction (only changes to the input settings were simulated as user interaction, since changes to the algorithm would also change the graph properties, making comparison impossible.). The fraction of changed inputs is called the ***ChangedFraction***. Caching is (predictably) more effective for a low ChangedFraction.

Simulations were compared for data driven and demand driven execution of the same graphs, with the same pattern of changed input terminals. It can be argued that a demand driven environment will prompt the use of a different "programming style" than a data driven environment does. However, this can hardly be taken into account in a simulation. It was found that the dataflow execution method (data driven or demand driven) doesn't significantly influence the relative performance of the caching schemes (it does influence overall performance), and therefore this discussion will be limited to data driven execution.

The simulations suggest that caching can significantly reduce the overall calculation cost. The use of dataflow information can dramatically reduce the caching overhead. Figure 1 shows the data driven calculation cost relative to the cost of total graph recalculation, and the data driven caching overhead relative to the total data size, (which represents the minimum caching overhead when dataflow information is unused).

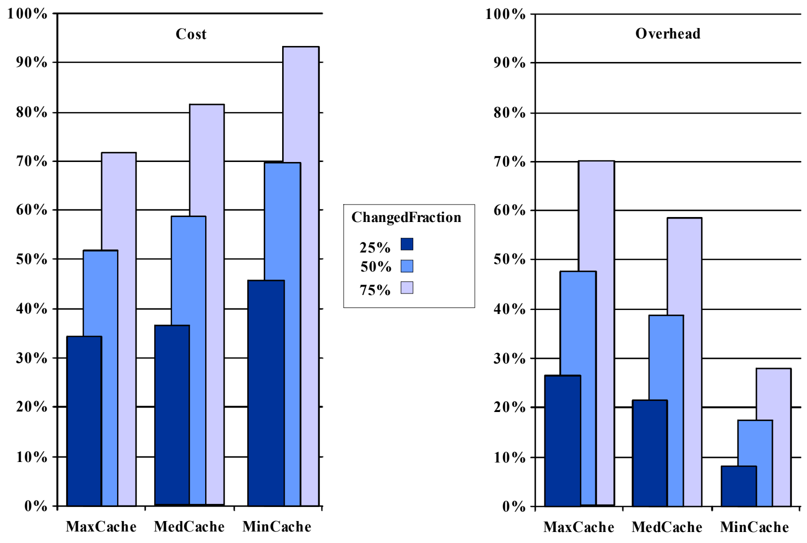

Figure 1 Cost and Overhead averages for different ChangedFraction values

The trade-off between calculation cost and comparison overhead will be determined by how these factors compare for a specific system. This depends on the implementation of comparison, on the average data size, and the order of complexity of the operations in the algorithm, as well as on the graph and interaction parameters. No caching scheme is always superior to all others, and general conclusions are not straightforward.

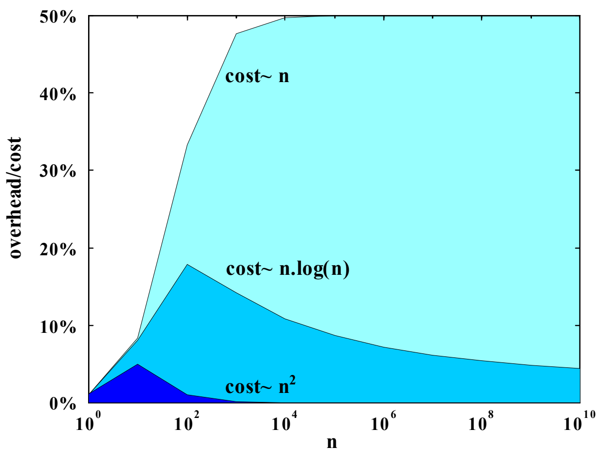

Figure 2 Overhead relative to cost for some typical complexities

Most likely the overhead is less important than the calculation cost, depending on the complexity of the operations and the data size. As Figure 2 demonstrates, the caching overhead quickly becomes insignificant compared to the calculation cost for algorithms with a calculation complexity of order n^2 or higher. In that case, whether or not dataflow information is used becomes unimportant from the overhead point of view.

Considering this, the data size and algorithm complexity will also determine areas where caching can be effective. Five different areas can be identified.

1. For simple operations on small data sets, caching is unnecessary.
2. For more complex operations, caching can be worthwhile to reduce calculation cost, even on small data sets. However, when the data size is small, the caching overhead and the required cache size are insignificant and the use of dataflow information to reduce these quantities is unnecessary.
3. For moderately complex algorithms executed on large amounts of data, caching will reduce the calculation cost while dataflow information can reduce the caching overhead.
4. For high complexity calculations, the caching overhead is insignificant relative to the calculation cost, even if the overhead is large in an absolute sense. The use of dataflow information to reduce the overhead will only marginally affect the execution time.
5. For large data sets, dataflow information can be useful to reduce the required cache size, even for complex algorithms where the caching overhead is relatively insignificant.

The resulting areas are summarised in Figure 3 (note that the scale and the relative proportions of the areas are drawn arbitrarily).

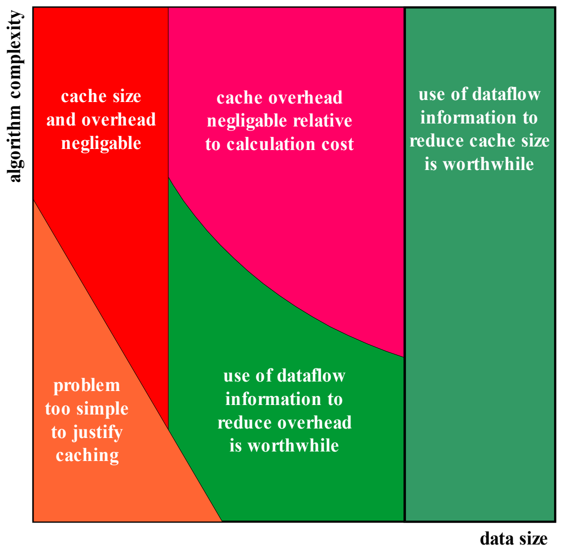
algorithm complexity
cache size and overhead negligable
cache overhead negligable relative to calculation cost
use of dataflow information to reduce cache size is worthwhile
problem too simple to justify caching
use of dataflow information to reduce overhead is worthwhile
data size

Figure 3 Effect of data size and algorithm complexity on the usefulness of caching

The performance of the caching schemes depends on numerous factors. In our simulations on random generated graphs, none of the schemes is consistently superior.

Figure 4 shows the density distribution of graph parameters for which each caching scheme offers the most economical solution. The elevation in the 3D mesh represents the fraction of graphs for which the scheme produces the lowest cost and overhead. The sum of the calculation cost and caching overhead is compared for the three caching methods, and for total recalculation without caching (***NoCache*** on the figures). The ChanceFactor represents the node properties, and the InputFraction (number of input terminals relative to the total number of nodes) is chosen here to represent the graph structure.

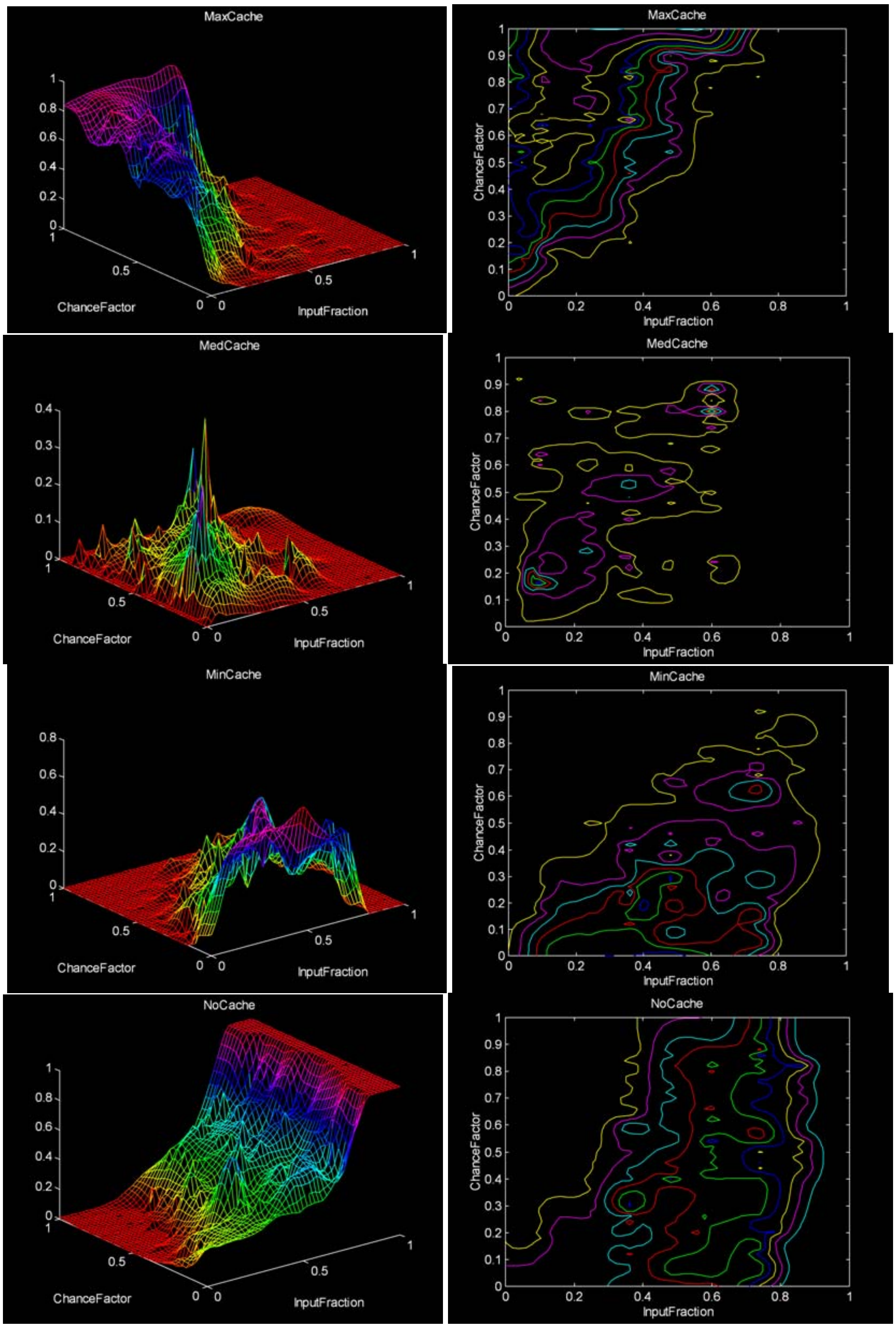

Figure 4 Areas of method effectiveness

It is clear from this example that regions of scheme effectiveness can be defined for certain graph properties. The regions will of course also depend on the ChangedFraction and the overhead importance relative to the cost (both 50% in Figure 4), but a full treatment of these factors falls outside the scope of the present paper. For several sample algorithms, the performance was calculated mathematically, and the exact results for these algorithms conform reasonably well to the simulations.

5. CONCLUSION

For some conditions caching can result in significant time savings in applications that are often and repeatedly executed with largely identical settings. When dataflow information about the algorithm is available to the environment, as it is in dataflow-based virtual instrumentation environments for example, it can be used to reduce the cache overhead and to limit caching to relevant operations.

Depending on the nature of the dataflow graph or the user interaction, different caching schemes are appropriate, yet none of the proposed caching schemes consistently outperforms the others in all situations. Whether caching is appropriate or whether using the data dependency information will be beneficial for a given problem must be determined case by case.

However, based on simulation results, the caching scheme most likely to be effective for a given algorithm can be estimated from general properties of the algorithm's dataflow diagram. Mathematical methods can also be used to determine the amount of performance benefits possible for simple algorithms.

References

[1] 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.

[2] 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.

[3] 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.

[4] E. Steenput, Y. Rolain, "Caching of intermediate results in dataflow environments", Proceedings of the IEEE Workshop on Emergent Technologies & Virtual Systems for Instrumentation and Measurement, Minnesota Club, St. Paul, MN, USA, May 15-16, 1998, pp 138-147.