Caching in Dataflow-Based Environments

Eli Steenput, Yves Rolain
Dept ELEC
Vrije Universiteit Brussel
Pleinlaan 2
1050 Brussel, Belgium
esteenpu@vub.ac.be, yrolain@vub.ac.be

ABSTRACT

Caching can result in significant time savings in applications where repeated executions with slightly altered settings or algorithms are frequent. 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 caching schemes are proposed, and methods are presented to determine the caching scheme best suited for a particular algorithm, based on 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.

To prevent the redundant execution of an operation, its previous arguments and the corresponding results can be stored in a data cache, so that when the current combination of argument values is found in the cache, the corresponding result can be retrieved instead of being recalculated.

Cache lookup will create some overhead, and because data can be quite large in a measurement or modelling environment, caching all intermediate results is not always feasible. Data dependency information can be used to reduce the necessary cache size and cache lookup overhead. In measurement environments based on virtual instruments, the dataflow information of the algorithm is readily available. There are several advantages to caching integrated 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.

[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).

2. Caching schemes

Three different caching schemes will be presented, each comparing new values to cached results for a different number of nodes.

2.1 The MinCache scheme

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. This can be explained as follows:

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 exactly one thread (or constitutes a thread by itself).

Nodes in the same thread all depend on the same 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 caching. This permits a reduction of the number of caches, without reduction of caching efficiency. The number of threads in a graph will determine the number of results that require caching, and the required cache space. The name of the scheme refers 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 caching, is calculated as follows:

If a node depends on p input terminals, and assuming all input states are equally likely, the probability that a node is affected by a change to the input terminals equals $1-2^{-p}$. This is easily derived: suppose the number of input terminals of the graph is i. The number of possible input states is $\sum_{k=0}^{i}\binom{i}{k}=2^{i}$ (with k the number of changed inputs). If a node depends on p input terminals, the number of possible input states where none of the p inputs changes is $\sum_{k=0}^{i-p}\binom{i-p}{k}=2^{i-p}$. The probability that one of the node’s inputs changed is thus $\frac{2^{i}-2^{i-p}}{2^{i}}=1-2^{-p}$.

The MinCache calculation cost can be obtained by multiplying the cost of each node with the probability of its execution, and summing the obtained values. The cost without caching is simply the sum of the costs of all nodes. The only overhead consists of comparing new input settings (data fetching etc. are performed anyway in a dataflow environment).

2.2 The MaxCache scheme

If nodes sometimes produce the same output for different argument values, it is useful to compare a newly calculated result with the previous value in the cache.

Suppose each node (or input terminal) has an associated cache to store its last value. An executing node can compare its new result first with the cached value. If the new result is the same as the previous value in the cache, the result is marked “unchanged” by a tag that is added to the data. A node that has only unchanged arguments does not have to execute. Its cached result is still valid and can be re-issued, with an 'unchanged' tag added to it.

This scheme stops execution on paths depending on a changed input where some of the intermediate operations yield the same results (relational operators for example often produce the same result for different input values).

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.

To calculate the MaxCache cost, for each node n the probability of producing the same value for changed input arguments must be known. This probability will be called r_n. To determine the calculation cost of an interaction, the cost of a node n is multiplied by the probability P_n that the node will be executed (provided it depends on a changed input). Several cases must be distinguished in the MaxCache scheme:

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 terminal i, $r_i = 0$, $P_i = 0$ if the input is unchanged and $P_i = 1$ for a changed input.

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 that is not an input terminal:

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

This is no longer a simple calculation, and the probabilities r will have to be estimated. Also, P_n depends on the structure of the subgraph leading to n.

2.3 The MedCache scheme

This is a hybrid scheme, that caches only the end nodes of threads just 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.

3. Performance of caching schemes

Because only a limited number of graphs were available for analysis, performance measure were obtained by simulations on random generated graphs. Modifications to the graph were not simulated, only changes to the input settings.

To compare different graphs and draw general conclusions, some quantitative properties, representative of each graph, must be derived. These properties should be easy to determine and promote understanding of the factors affecting performance.

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 number of inputs and the graph connectivity), but also on the unpredictable sequence of user interactions.

The graph properties that affect performance can be divided into node properties and structural properties.

The node properties are:

1. the node's calculation cost,
2. the probability to yield the same result for different inputs (the average value for the graph is called the ChanceFactor from now on),
3. the overhead required to compare this node's result with the cache.

Simulated cost and result lookup overhead are normalised relative to the average node cost and node data size of the graph. The absolute value of these parameters has no influence on the effectiveness of the caching scheme. This leaves the ChanceFactor as the only (averaged) node property that influences the caching performance.

The structural properties of dataflow graphs can be represented by various parameters. The calculation cost will of course increase with graph size (=number of nodes), but if all size-dependent values are normalised for graph size, normalised performance measures appear to be almost size-independent. The following parameters were selected:

1. the InputFraction or number of inputs relative to graph size,
2. the BranchFactor or average number of output arcs per node,
3. the ThreadFraction or number of threads relative to number of nodes.

The interaction will affect the performance through the number of input terminals that are changed in the interaction. The fraction of changed inputs is called the ChangedFraction.

A dataflow execution system can be data driven or demand driven, but it was found that the execution system doesn't significantly influence the relative performance of the caching schemes (it does influence overall performance).

4. Influence of graph structure on calculation cost

4.1 Cost dependency on the InputFraction

The higher the InputFraction (number of input nodes relative to graph size), the higher the cost. Note that the simulated results assume a uniform distribution of input terminals. Equally-sized graphs with fewer inputs have fewer and longer threads than those with many inputs, so the effect of one unchanged input on the calculation cost is more pronounced. In the MaxCache scheme, detecting nodes that produce the same results for changed inputs is more effective for graphs with long threads. For very high InputFractions, the majority of nodes are input terminals and the costs are virtually identical for all caching schemes.

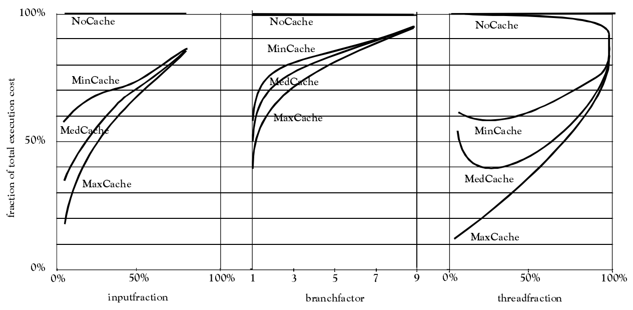


Figure 1 Effect of graph properties on calculation cost

4.2 Cost dependency on the BranchFactor

Increasing the BranchFactor will cause an increase in graph interconnection, and the graph will consist of more, shorter threads. For the maximum BranchFactor value, all nodes are interconnected and each node depends on all input terminals. Caching is almost useless in such a graph, since a change to any one input always requires recalculation of the entire graph.

4.3 Cost dependency on ThreadFraction

Caching is most effective in graphs consisting of few, large threads, where each thread depends on a small fraction of the input terminals. For very high BranchFactor values, the ThreadFraction decreases with the BranchFactor while the cost continues to increase until every node depends on all input terminals. All nodes then belong to one single thread. However, these unrealistically high BranchFactors are of little relevance to measurement environments.

The plots of cost vs. InputFraction will change considerably when the BranchFactor is varied, and the cost vs. BranchFactor plots will similarly depend on the InputFraction. In contrast, the behaviour of the calculation costs as a function of the ThreadFraction shows a great similarity for a very wide variation of graph structures. Despite its behaviour for high BranchFactor values, the ThreadFraction parameter has a high potential to explain the influence of graph structure on performance.

5. Influence of node and interaction properties on calculation cost

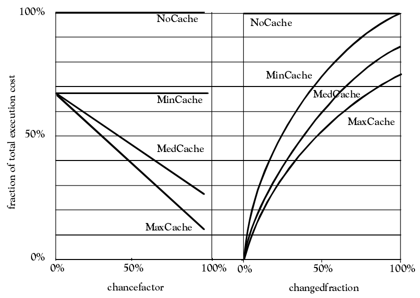


Figure 2 Effect of node and interaction properties on calculation cost

5.1 Cost dependency on the ChanceFactor

After normalisation, the only node property influencing the caching scheme is the ChanceFactor. If the ChanceFactor is zero, the three caching scheme should result in identical calculation costs. A low ChanceFactor (corresponding to most executed nodes producing changed values) will yield a MaxCache cost as high as the MinCache cost. The difference between MaxCache cost and MinCache cost increases with increasing ChanceFactor. This makes sense, since the MaxCache scheme detects the nodes that produce the same results for changed inputs. As expected, the MedCache scheme results in a calculation cost somewhere in between those of MinCache and MaxCache.

5.2 Cost dependency on ChangedFraction

The only interaction parameter affecting the cost is the fraction of changed input values. The MinCache cost varies from 0% of the NoCache cost when no inputs have changed to 100% when all inputs have changed. The MaxCache and MedCache costs can be smaller than NoCache cost even when all inputs have changed. This can be attributed to the skipping of parts of the graph due to nodes producing unchanged results upon execution.

6. Influence of graph structure on caching overhead

The caching overhead is expressed relative to the overhead for comparing all intermediate results with their previous values. The chosen reference assumes each intermediate result is compared exactly once to a cached value. This is the absolute minimum needed to make caching possible without data dependency information. The simulated overheads are always lower than the chosen reference, and the relative measure allows the comparison of caching overhead for different graphs.

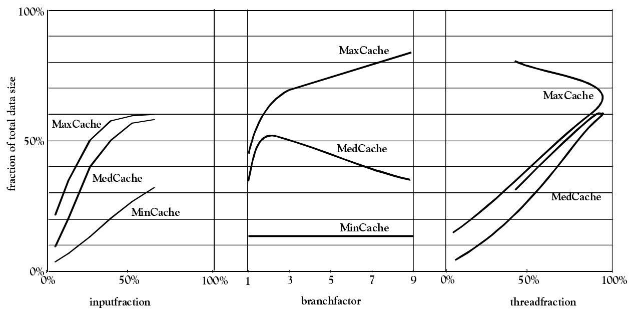


Figure 3 Effect of graph properties on caching overhead

6.1 Overhead dependency on InputFraction

The MinCache overhead is caused by comparing values for the input terminals, so naturally this overhead is proportional to the InputFraction. The MedCache and MaxCache scheme compare values for executed nodes as well as for the input terminals, and these overheads will be larger as more nodes are executed.

6.2 Overhead dependency on BranchFactor

The MinCache overhead remains unaffected by variation of the BranchFactor. MaxCache overhead increases with the BranchFactor for the same reason as the calculation cost; more executed nodes require more comparisons. The MedCache scheme compares only values for executed nodes at the end of threads. Since there are fewer threads for a high BranchFactor, a maximum in the MedCache overhead occurs.

6.3 Overhead dependency on ThreadFraction

The overhead for the MaxCache and MedCache scheme increases with the ThreadFraction about the same way as the calculation cost. For very high, increasing BranchFactor values, the MaxCache overhead will increase with the decreasing ThreadFraction (just like the calculation cost).

The MedCache overhead will decrease with the decreasing ThreadFraction at a level somewhat above the overhead for normal BranchFactor values. Plotting MinCache overhead as a function of ThreadFraction doesn't make much sense, as it depends only on the number of input terminals.

7. Influence of node and interaction properties on caching overhead

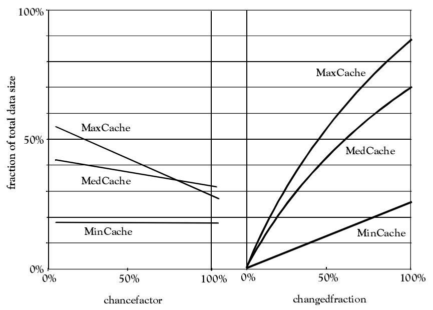


Figure 4 Effect of node and interaction properties on caching overhead

7.1 Overhead dependency on ChanceFactor

The MinCache overhead is independent of the ChanceFactor. A high ChanceFactor reduces the number of nodes to be executed and thereby also reduces the number of results to be compared to cached values for the MaxCache and MinCache scheme. The effectiveness of comparing extra results increases with the ChanceFactor, and for very high ChanceFactor values the MaxCache scheme actually performs less comparisons than the MedCache scheme.

7.2 Overhead dependency on ChangedFraction

If the system knows which input settings were updated by the user, only an updated setting will be checked for change, and the MinCache overhead is proportional to the changed fraction. The additional overhead in the MedCache and MaxCache schemes is proportional to the number of executed nodes, so it will increase with increasing ChangedFraction. The rate of increase will depend on the other graph properties.

8. Cost and overhead combined

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 hardware platform running the environment.

In Figure 5, the overhead/cost proportion is linearly increased. As can be seen, no scheme is always superior to all others, and general conclusions are not straightforward. The curve for caching without the use of dataflow information uses the minimum overhead for this scheme (as the exact value depends on the unknown user interaction history), and the same cost as the MaxCache scheme is assumed.

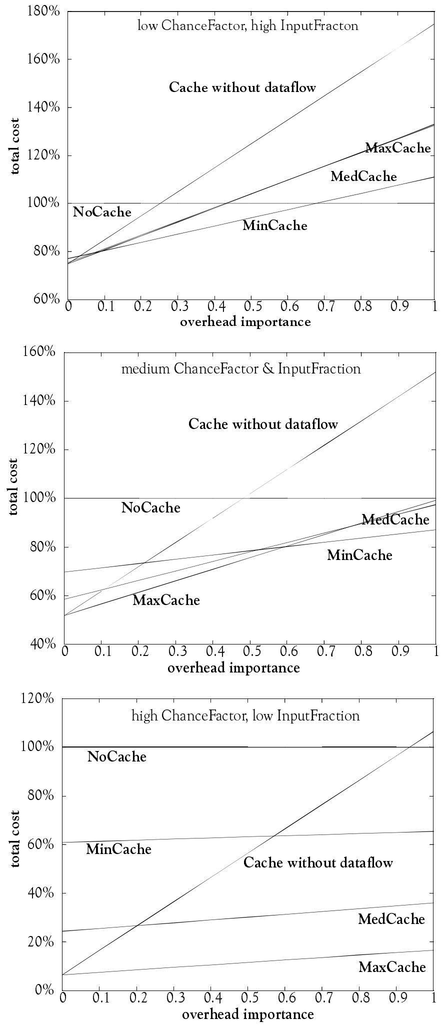


Figure 5 Total Cost (calculation + overhead) for increasing relative comparison cost.

The figure on top shows the total cost for graphs with low ChanceFactor and high InputFraction. The benefit of caching is small, and it is hard to say which caching scheme is superior. Not using caching is more effective than any caching scheme as soon as the overhead rises above 70% of the cost.

The middle figure is for medium ChanceFactor and InputFraction values (medium relative to the average of the simulations, not necessarily for realistic graphs). It appears the MedCache scheme is only cheaper than the MaxCache when the MinCache scheme outperforms both.

For high ChanceFactor, low InputFraction (bottom figure), very different results are obtained. Caching is always effective, and there is no question that the MaxCache scheme is superior, and that it is very effective even when the cost of caching without using dataflow information exceeds the total calculation cost without caching.

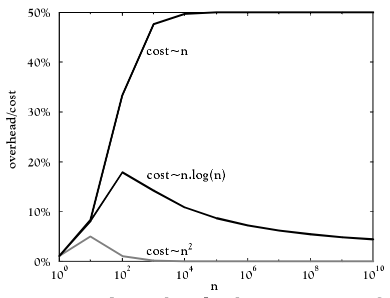


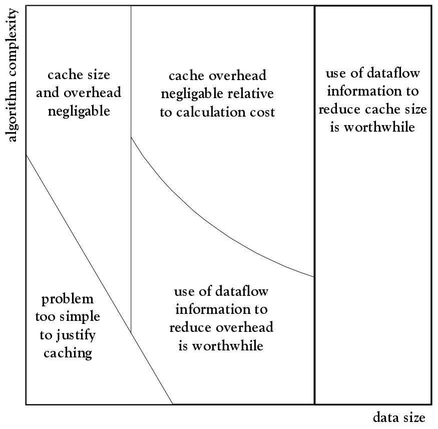
Figure 6 Overhead relative to cost for some typical complexities

Most likely the overhead is lower than the calculation cost, depending on the complexity of the operations and the data size. As Figure 6 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 just 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 cache size, even for complex algorithms where the caching overhead is insignificant.

The resulting areas are summarised in Figure 7 (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 7 Effect of data size and algorithm complexity on the usefulness of caching

8.1 Areas of caching scheme effectiveness

Because none of the caching schemes is consistently superior, we will attempt to identify areas of effectiveness for the various schemes. 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 was found to have a strong influence on the effectiveness of caching. The InputFraction, ThreadFraction and BranchFactor are not independent but interrelated. From these last three the ThreadFraction is the most general and most capable of explaining the variations in cost and overhead, however the InputFraction produces somewhat more easily readable figures.

Figure 8 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 number of graphs for which the scheme produces the lowest cost, relative to the total number of samples in the local area. To improve clarity in the figures, a surface is interpolated for grid points where no values were available.

It is clear that regions of effectiveness can be defined, even though the schemes may be preferable for less than 100% of the graphs in their regions. It must be noted that the very high InputFractions, where the graph contains far more input terminals than nodes, and high ChanceFactors, are included only to make the global cost behaviour more apparent, not because such graphs would be typical for the structure of any real application (a graph with an InputFraction of 0 or 1 is impossible).

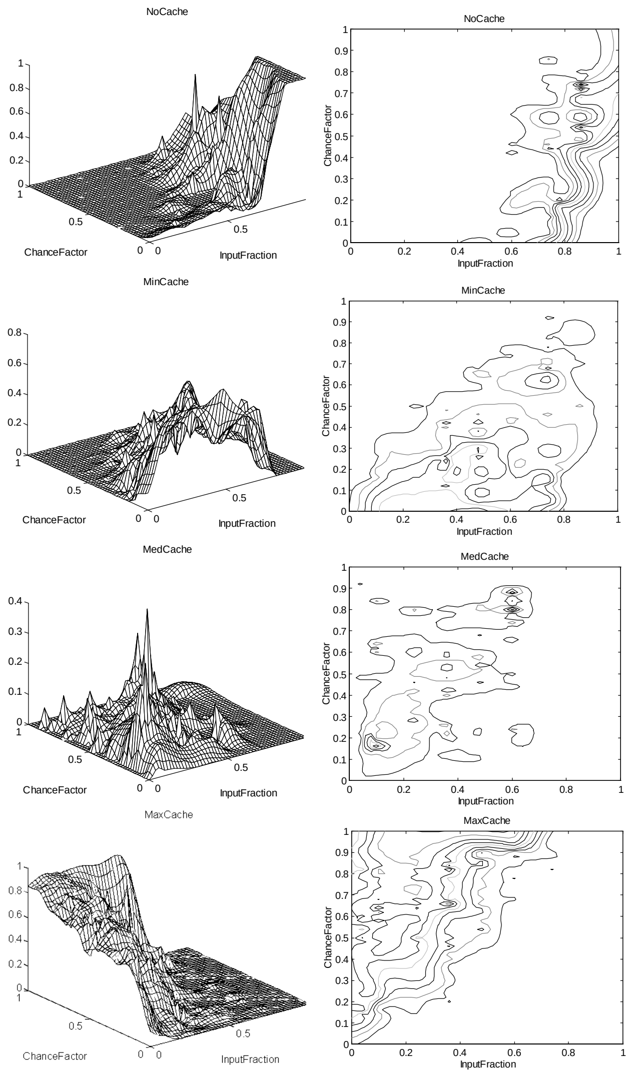


Figure 8 Areas of method effectiveness

For a high InputFraction and a low ChanceFactor, the difference in overhead between the MinCache scheme and the other caching schemes is the largest. For a higher ChanceFactor, the other caching schemes become more effective, whereas the performance of the MinCache scheme doesn't depend on the ChanceFactor.

The MedCache scheme seems to be most effective for a minority of graphs only, but spread over a large area. For a high InputFraction, the MedCache overhead will become important, while the MaxCache scheme will benefit more from a high ChanceFactor. Note that there are other factors besides performance, such as the required cache space, which were not taken into account to determine these areas of method effectiveness.

The MaxCache scheme appears to be preferable for the majority of graphs with low InputFraction, especially when the ChanceFactor is high as well. For a very high InputFraction, most nodes will have at least one input terminal as argument. In such a graph the caching scheme is not able to exploit the high ChanceFactor. For a zero or very low ChanceFactor, the MinCache scheme performs best.

Figure 9 shows the influence of the overhead importance on the areas of caching effectiveness. In the lightest area the MaxCache is the most effective scheme, darkest means caching is ineffective. Dark grey represents the MinCache scheme, light grey the MedCache scheme. Figure 9 also shows ChanceFactor/ThreadFraction graphs.

Note the areas of effectiveness for a overhead importance of 100%, which corresponds to a total caching overhead that equals the total calculation cost. In an environment that doesn't make use of dataflow information, such conditions would completely exclude caching as a measure to increase performance. However, Figure 9 clearly indicates that a caching scheme assisted by dataflow information can still perform effectively under these conditions.

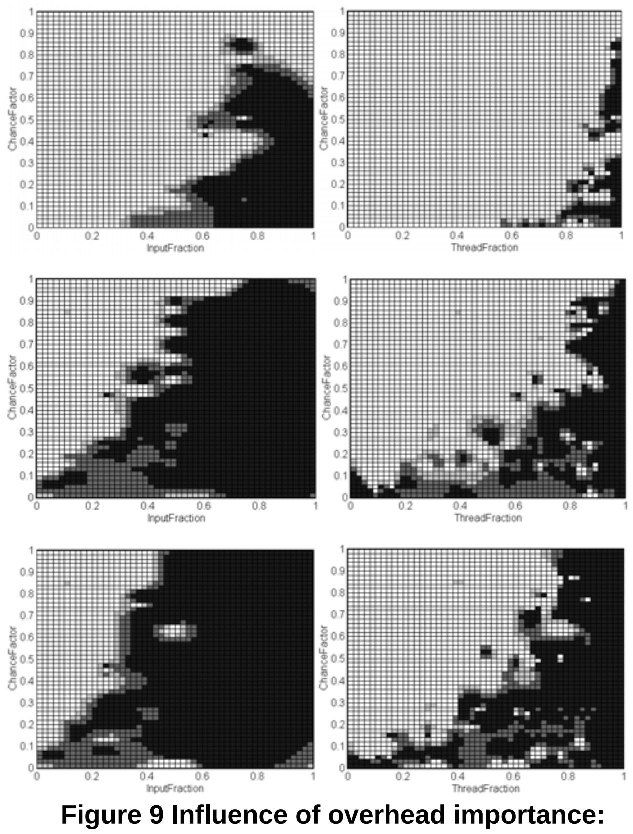


Figure 9 Influence of overhead importance: 1%, 50% and 100%

The areas of effectiveness in the previous figures were constructed assuming all input states are equally likely, which corresponds to an average ChangedFraction of 50%. Changing only a small fraction of the input terminals during each interaction will benefit caching, while leaving only a few input terminals unchanged during each interaction will have an adverse effect on the effectiveness of caching.

Figure 10 show the (interpolated) areas of method effectiveness for average ChangedFractions of 25%, 50%, and 75% (overhead importance 50%).

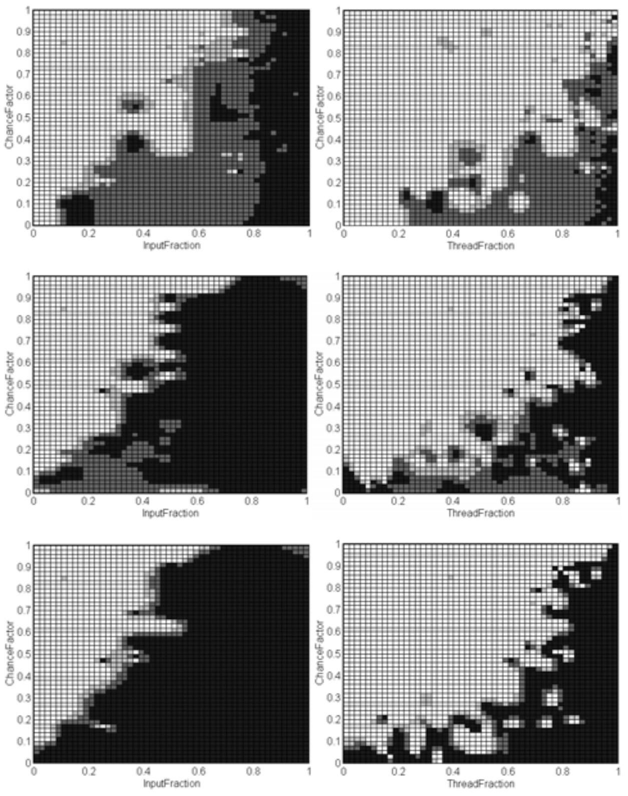


Figure 10 Influence of the ChangedFraction: 25%, 50% and 75%

Note that the area of best performance for the MaxCache scheme changes little. The ChangedFraction has more effect on the other schemes.

For a low ChangedFraction, the calculation cost is almost identical for the three caching schemes, but the lower overhead of the MedCache and MinCache schemes can make these schemes even more effective than the MaxCache scheme. For a high ChangedFraction, the cost difference between the caching schemes is more pronounced.

For a zero ChanceFactor, the MedCache and MaxCache scheme reduce the calculation cost only as much as the MinCache scheme, but at the cost of a much higher overhead. As a result, the MinCache scheme is the most effective scheme for low ChanceFactor values, unless the ChangedFraction is too high to make caching profitable (for zero ChanceFactor).

9. Conclusion

Caching can result in significant time savings in applications that are often and repeatedly executed with largely identical values. Dataflow information, as available in dataflow-based environments, can be used to reduce cache overhead and to limit the memory space required by caching. Different caching schemes were suggested, appropriate for different types of dataflow graph or user interaction. Methods (mathematical and based on simulations) were presented to determine the caching scheme best suited for a particular algorithm, and to estimate the possible savings.

References

[1] E. C. Baroth, C. Hartsough, “Experience report: Visual Programming in the Real World”, *Visual Object Oriented Programming*, M. M. Burnett, A. Goldberg, T. G. Lewis (editors), Manning Publications, Prentice Hall, 1995, pp 21-22.

[2] B. W. Char, *Maple User’s Guide*. Waterloo, Ontario, WATCOM Publications, 1988.

[3] V. K. Chaudri, R. Greiner, “A Formal Analysis of Solution Caching”, *Proceedings of the Canadian Artificial Intelligence Conference*, Vancouver, 11-15 May, 1992.

[4] *IBM Visualisation Data Explorer User’s Guide 3.1.4*, IBM Corporation, May 1997.

[5] F. M. Rijnders, *A visual programming environment for scientific applications: possibilities and limitations*, academisch proefschrift ter verkrijging van de graad van doctor aan de Vrije Universiteit te Amsterdam, 29 juni 1995.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] A. Woodruff, M. Stonebraker, “Buffering of Intermediate Results in Dataflow Diagrams”, *Proceedings of the IEEE Symposium on Visual Languages*, Darmstadt, Germany, September 1995.